

# OpenSplice DDS

Version 5.x

## Java Reference Guide





# OpenSplice DDS

## JAVA REFERENCE GUIDE



Part Number: OS-JREFG

Doc Issue 31, 6 January 2011

## Copyright Notice

© 2011 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 Java Reference Guide .....	xxi
Contacts .....	xxiii

## Introduction

About the Java Reference Guide .....	3
Operations .....	4

## API Reference

<b>Chapter 1</b>	<b>DCPS API General Description</b>	<b>7</b>
1.1	IDL Mapping to Java .....	8
1.1.1	IDL-Constant Mapping .....	8
1.1.2	IDL-Sequence Mapping .....	9
1.2	Thread Safety .....	9
1.3	Signal Handling .....	10
1.4	Memory Management .....	11
1.5	Parameter Passing .....	11
1.6	Casting of Objects .....	13
1.7	Listeners Interfaces .....	14
1.8	Inheritance of Abstract Operations .....	15
<b>Chapter 2</b>	<b>DCPS Modules</b>	<b>17</b>
2.1	Functionality .....	17
2.2	Infrastructure Module .....	18
2.3	Domain Module .....	19
2.4	Topic-Definition Module .....	21
2.5	Publication Module .....	22
2.6	Subscription Module .....	23
<b>Chapter 3</b>	<b>DCPS Classes and Operations</b>	<b>27</b>
3.1	Infrastructure Module .....	28
3.1.1	Interface Entity .....	28
3.1.1.1	enable .....	29
3.1.1.2	get_instance_handle .....	31
3.1.1.3	get_listener (abstract) .....	32
3.1.1.4	get_qos (abstract) .....	32
3.1.1.5	get_status_changes .....	32
3.1.1.6	get_statuscondition .....	34

3.1.1.7 set_listener (abstract) . . . . .	34
3.1.1.8 set_qos (abstract) . . . . .	35
3.1.2 Class DomainEntity . . . . .	35
3.1.3 Class QosPolicy . . . . .	35
3.1.3.1 DeadlineQosPolicy . . . . .	44
3.1.3.2 DestinationOrderQosPolicy . . . . .	46
3.1.3.3 DurabilityQosPolicy . . . . .	48
3.1.3.4 DurabilityServiceQosPolicy . . . . .	52
3.1.3.5 EntityFactoryQosPolicy . . . . .	54
3.1.3.6 GroupDataQosPolicy . . . . .	55
3.1.3.7 HistoryQosPolicy . . . . .	56
3.1.3.8 LatencyBudgetQosPolicy . . . . .	58
3.1.3.9 LifespanQosPolicy . . . . .	60
3.1.3.10 LivelinessQosPolicy . . . . .	61
3.1.3.11 OwnershipQosPolicy . . . . .	63
3.1.3.12 OwnershipStrengthQosPolicy . . . . .	66
3.1.3.13 PartitionQosPolicy . . . . .	66
3.1.3.14 PresentationQosPolicy . . . . .	67
3.1.3.15 ReaderDataLifecycleQosPolicy . . . . .	73
3.1.3.16 ReliabilityQosPolicy . . . . .	74
3.1.3.17 ResourceLimitsQosPolicy . . . . .	77
3.1.3.18 SchedulingQosPolicy . . . . .	78
3.1.3.19 TimeBasedFilterQosPolicy . . . . .	80
3.1.3.20 TopicDataQosPolicy . . . . .	80
3.1.3.21 TransportPriorityQosPolicy . . . . .	81
3.1.3.22 UserDataQosPolicy . . . . .	82
3.1.3.23 WriterDataLifecycleQosPolicy . . . . .	82
3.1.4 Listener Interface . . . . .	83
3.1.5 Class Status . . . . .	85
3.1.5.1 InconsistentTopicStatus . . . . .	89
3.1.5.2 LivelinessChangedStatus . . . . .	90
3.1.5.3 LivelinessLostStatus . . . . .	91
3.1.5.4 OfferedDeadlineMissedStatus . . . . .	92
3.1.5.5 OfferedIncompatibleQosStatus . . . . .	93
3.1.5.6 PublicationMatchedStatus . . . . .	96
3.1.5.7 RequestedDeadlineMissedStatus . . . . .	96
3.1.5.8 RequestedIncompatibleQosStatus . . . . .	97
3.1.5.9 SampleLostStatus . . . . .	99
3.1.5.10 SampleRejectedStatus . . . . .	100
3.1.5.11 SubscriptionMatchedStatus . . . . .	101
3.1.6 Class WaitSet . . . . .	102
3.1.6.1 attach_condition . . . . .	103



3.1.6.2	detach_condition	104
3.1.6.3	get_conditions	105
3.1.6.4	wait	106
3.1.7	Class Condition	107
3.1.7.1	get_trigger_value	108
3.1.8	Class GuardCondition	109
3.1.8.1	get_trigger_value (inherited)	110
3.1.8.2	set_trigger_value	110
3.1.9	Class StatusCondition	111
3.1.9.1	get_enabled_statuses	112
3.1.9.2	get_entity	114
3.1.9.3	get_trigger_value (inherited)	114
3.1.9.4	set_enabled_statuses	115
3.2	<b>Domain Module</b>	<b>117</b>
3.2.1	Class DomainParticipant	117
3.2.1.1	assert_liveliness	121
3.2.1.2	contains_entity	122
3.2.1.3	create_contentfilteredtopic	123
3.2.1.4	create_multitopic	124
3.2.1.5	create_publisher	126
3.2.1.6	create_subscriber	129
3.2.1.7	create_topic	131
3.2.1.8	delete_contained_entities	134
3.2.1.9	delete_contentfilteredtopic	136
3.2.1.10	delete_multitopic	137
3.2.1.11	delete_publisher	138
3.2.1.12	delete_subscriber	139
3.2.1.13	delete_topic	140
3.2.1.14	enable (inherited)	142
3.2.1.15	find_topic	142
3.2.1.16	get_builtin_subscriber	143
3.2.1.17	get_current_time	144
3.2.1.18	get_default_publisher_qos	145
3.2.1.19	get_default_subscriber_qos	146
3.2.1.20	get_default_topic_qos	147
3.2.1.21	get_discovered_participants	149
3.2.1.22	get_discovered_participant_data	149
3.2.1.23	get_discovered_topics	149
3.2.1.24	get_discovered_topic_data	149
3.2.1.25	get_domain_id	150
3.2.1.26	get_listener	151
3.2.1.27	get_qos	151

3.2.1.28	get_status_changes (inherited) . . . . .	152
3.2.1.29	get_statuscondition (inherited) . . . . .	152
3.2.1.30	ignore_participant . . . . .	153
3.2.1.31	ignore_publication . . . . .	153
3.2.1.32	ignore_subscription . . . . .	153
3.2.1.33	ignore_topic . . . . .	154
3.2.1.34	lookup_topicdescription . . . . .	154
3.2.1.35	set_default_publisher_qos . . . . .	155
3.2.1.36	set_default_subscriber_qos . . . . .	156
3.2.1.37	set_default_topic_qos . . . . .	157
3.2.1.38	set_listener . . . . .	158
3.2.1.39	set_qos . . . . .	161
3.2.2	Class DomainParticipantFactory . . . . .	162
3.2.2.1	create_participant . . . . .	163
3.2.2.2	delete_participant . . . . .	167
3.2.2.3	get_default_participant_qos . . . . .	168
3.2.2.4	get_instance . . . . .	169
3.2.2.5	get_qos . . . . .	169
3.2.2.6	lookup_participant . . . . .	170
3.2.2.7	set_default_participant_qos . . . . .	171
3.2.2.8	set_qos . . . . .	173
3.2.2.9	delete_domain . . . . .	174
3.2.2.10	lookup_domain . . . . .	174
3.2.2.11	delete_contained_entities . . . . .	175
3.2.3	Class Domain . . . . .	176
3.2.3.1	create_persistent_snapshot . . . . .	177
3.2.4	DomainParticipantListener Interface . . . . .	178
3.2.4.1	on_data_available (inherited, abstract) . . . . .	180
3.2.4.2	on_data_on_readers (inherited, abstract) . . . . .	181
3.2.4.3	on_inconsistent_topic (inherited, abstract) . . . . .	181
3.2.4.4	on_liveliness_changed (inherited, abstract) . . . . .	181
3.2.4.5	on_liveliness_lost (inherited, abstract) . . . . .	181
3.2.4.6	on_offered_deadline_missed (inherited, abstract) . . . . .	182
3.2.4.7	on_offered_incompatible_qos (inherited, abstract) . . . . .	182
3.2.4.8	on_publication_matched (inherited, abstract) . . . . .	182
3.2.4.9	on_requested_deadline_missed (inherited, abstract) . . . . .	182
3.2.4.10	on_requested_incompatible_qos (inherited, abstract) . . . . .	183
3.2.4.11	on_sample_lost (inherited, abstract) . . . . .	183
3.2.4.12	on_sample_rejected (inherited, abstract) . . . . .	183
3.2.4.13	on_subscription_matched (inherited, abstract) . . . . .	183
3.2.5	ExtDomainParticipantListener interface . . . . .	184
3.2.5.1	on_all_data_disposed (inherited, abstract) . . . . .	185

3.2.5.2	on_data_available (inherited, abstract) . . . . .	186
3.2.5.3	on_data_on_readers (inherited, abstract) . . . . .	186
3.2.5.4	on_inconsistent_topic (inherited, abstract) . . . . .	186
3.2.5.5	on_liveliness_changed (inherited, abstract) . . . . .	186
3.2.5.6	on_liveliness_lost (inherited, abstract) . . . . .	187
3.2.5.7	on_offered_deadline_missed (inherited, abstract) . . . . .	187
3.2.5.8	on_offered_incompatible_qos (inherited, abstract) . . . . .	187
3.2.5.9	on_publication_matched (inherited, abstract) . . . . .	187
3.2.5.10	on_requested_deadline_missed (inherited, abstract) . . . . .	188
3.2.5.11	on_requested_incompatible_qos (inherited, abstract) . . . . .	188
3.2.5.12	on_sample_lost (inherited, abstract) . . . . .	188
3.2.5.13	on_sample_rejected (inherited, abstract) . . . . .	188
3.2.5.14	on_subscription_matched (inherited, abstract) . . . . .	189
3.3	<b>Topic-Definition Module</b> . . . . .	<b>189</b>
3.3.1	Interface TopicDescription . . . . .	190
3.3.1.1	get_name . . . . .	191
3.3.1.2	get_participant . . . . .	192
3.3.1.3	get_type_name . . . . .	192
3.3.2	Interface Topic . . . . .	193
3.3.2.1	enable (inherited) . . . . .	194
3.3.2.2	get_inconsistent_topic_status . . . . .	194
3.3.2.3	dispose_all_data . . . . .	195
3.3.2.4	get_listener . . . . .	197
3.3.2.5	get_name (inherited) . . . . .	197
3.3.2.6	get_participant (inherited) . . . . .	197
3.3.2.7	get_qos . . . . .	198
3.3.2.8	get_status_changes (inherited) . . . . .	199
3.3.2.9	get_statuscondition (inherited) . . . . .	199
3.3.2.10	get_type_name (inherited) . . . . .	199
3.3.2.11	set_listener . . . . .	199
3.3.2.12	set_qos . . . . .	201
3.3.3	Interface ContentFilteredTopic . . . . .	202
3.3.3.1	get_expression_parameters . . . . .	204
3.3.3.2	get_filter_expression . . . . .	205
3.3.3.3	get_name (inherited) . . . . .	205
3.3.3.4	get_participant (inherited) . . . . .	206
3.3.3.5	get_related_topic . . . . .	206
3.3.3.6	get_type_name (inherited) . . . . .	207
3.3.3.7	set_expression_parameters . . . . .	207
3.3.4	Interface MultiTopic . . . . .	208
3.3.4.1	get_expression_parameters . . . . .	209
3.3.4.2	get_name (inherited) . . . . .	210

3.3.4.3	get_participant (inherited)	210
3.3.4.4	get_subscription_expression	211
3.3.4.5	get_type_name (inherited)	211
3.3.4.6	set_expression_parameters	212
3.3.5	TopicListener interface	213
3.3.5.1	on_inconsistent_topic (abstract)	214
3.3.6	ExtTopicListener interface	214
3.3.6.1	on_all_data_disposed (abstract)	215
3.3.7	Topic-Definition type specific interfaces	216
3.3.7.1	Interface TypeSupport	216
3.3.7.2	get_type_name	217
3.3.7.3	register_type	217
3.3.7.4	Class FooTypeSupport	217
3.3.7.5	get_type_name	218
3.3.7.6	register_type	219
3.3.8	on_all_data_disposed (abstract)	220
3.4	<b>Publication Module.</b>	<b>222</b>
3.4.1	Interface Publisher.	223
3.4.1.1	begin_coherent_changes	225
3.4.1.2	copy_from_topic_qos	226
3.4.1.3	create_datawriter	228
3.4.1.4	delete_contained_entities	230
3.4.1.5	delete_datawriter	232
3.4.1.6	enable (inherited)	233
3.4.1.7	end_coherent_changes	233
3.4.1.8	get_default_datawriter_qos	234
3.4.1.9	get_listener	235
3.4.1.10	get_participant	236
3.4.1.11	get_qos	236
3.4.1.12	get_status_changes (inherited)	237
3.4.1.13	get_statuscondition (inherited)	237
3.4.1.14	lookup_datawriter	238
3.4.1.15	resume_publications	238
3.4.1.16	set_default_datawriter_qos	239
3.4.1.17	set_listener	240
3.4.1.18	set_qos	243
3.4.1.19	suspend_publications	244
3.4.1.20	wait_for_acknowledgments	245
3.4.2	Publication Type Specific Classes.	247
3.4.2.1	Interface DataWriter	247
3.4.2.2	assert_liveliness	250
3.4.2.3	dispose (abstract)	252

3.4.2.4	dispose_w_timestamp (abstract) . . . . .	252
3.4.2.5	enable (inherited) . . . . .	252
3.4.2.6	get_key_value (abstract) . . . . .	253
3.4.2.7	get_listener . . . . .	253
3.4.2.8	get_liveliness_lost_status . . . . .	254
3.4.2.9	get_matched_subscription_data . . . . .	255
3.4.2.10	get_matched_subscriptions . . . . .	256
3.4.2.11	get_offered_deadline_missed_status . . . . .	258
3.4.2.12	get_offered_incompatible_qos_status . . . . .	259
3.4.2.13	get_publication_matched_status . . . . .	260
3.4.2.14	get_publisher . . . . .	261
3.4.2.15	get_qos . . . . .	262
3.4.2.16	get_status_changes (inherited) . . . . .	263
3.4.2.17	get_statuscondition (inherited) . . . . .	263
3.4.2.18	get_topic . . . . .	263
3.4.2.19	lookup_instance (abstract) . . . . .	264
3.4.2.20	register_instance (abstract) . . . . .	264
3.4.2.21	register_instance_w_timestamp (abstract) . . . . .	265
3.4.2.22	set_listener . . . . .	265
3.4.2.23	set_qos . . . . .	267
3.4.2.24	unregister_instance (abstract) . . . . .	269
3.4.2.25	unregister_instance_w_timestamp (abstract) . . . . .	269
3.4.2.26	wait_for_acknowledgments . . . . .	269
3.4.2.27	write (abstract) . . . . .	271
3.4.2.28	write_w_timestamp (abstract) . . . . .	271
3.4.2.29	writedispose (abstract) . . . . .	272
3.4.2.30	writedispose_w_timestamp (abstract) . . . . .	272
3.4.2.31	Interface FooDataWriter . . . . .	272
3.4.2.32	assert_liveliness (inherited) . . . . .	275
3.4.2.33	dispose . . . . .	276
3.4.2.34	dispose_w_timestamp . . . . .	279
3.4.2.35	enable (inherited) . . . . .	280
3.4.2.36	get_key_value . . . . .	281
3.4.2.37	get_listener (inherited) . . . . .	282
3.4.2.38	get_liveliness_lost_status (inherited) . . . . .	282
3.4.2.39	get_matched_subscription_data (inherited) . . . . .	282
3.4.2.40	get_matched_subscriptions (inherited) . . . . .	282
3.4.2.41	get_offered_deadline_missed_status (inherited) . . . . .	283
3.4.2.42	get_offered_incompatible_qos_status (inherited) . . . . .	283
3.4.2.43	get_publication_matched_status (inherited) . . . . .	283
3.4.2.44	get_publisher (inherited) . . . . .	283
3.4.2.45	get_qos (inherited) . . . . .	284

3.4.2.46	get_status_changes (inherited)	284
3.4.2.47	get_statuscondition (inherited)	284
3.4.2.48	get_topic (inherited)	284
3.4.2.49	lookup_instance	284
3.4.2.50	register_instance	286
3.4.2.51	register_instance_w_timestamp	288
3.4.2.52	set_listener (inherited)	288
3.4.2.53	set_qos (inherited)	289
3.4.2.54	unregister_instance	289
3.4.2.55	unregister_instance_w_timestamp	292
3.4.2.56	wait_for_acknowledgments (inherited)	294
3.4.2.57	write	294
3.4.2.58	write_w_timestamp	296
3.4.2.59	writedispose	298
3.4.2.60	writedispose_w_timestamp	301
3.4.3	PublisherListener interface	303
3.4.3.1	on_liveliness_lost (inherited, abstract)	304
3.4.3.2	on_offered_deadline_missed (inherited, abstract)	304
3.4.3.3	on_offered_incompatible_qos (inherited, abstract)	304
3.4.3.4	on_publication_matched (inherited, abstract)	304
3.4.4	DataWriterListener interface	305
3.4.4.1	on_liveliness_lost	306
3.4.4.2	on_offered_deadline_missed	307
3.4.4.3	on_offered_incompatible_qos	308
3.4.4.4	on_publication_matched	309
3.5	<b>Subscription Module</b>	<b>311</b>
3.5.1	Interface Subscriber	312
3.5.1.1	begin_access	314
3.5.1.2	copy_from_topic_qos	314
3.5.1.3	create_datareader	316
3.5.1.4	delete_contained_entities	319
3.5.1.5	delete_datareader	320
3.5.1.6	enable (inherited)	321
3.5.1.7	end_access	322
3.5.1.8	get_datareaders	322
3.5.1.9	get_default_datareader_qos	322
3.5.1.10	get_listener	323
3.5.1.11	get_participant	324
3.5.1.12	get_qos	325
3.5.1.13	get_status_changes (inherited)	326
3.5.1.14	get_statuscondition (inherited)	326
3.5.1.15	lookup_datareader	326

3.5.1.16	notify_datareaders. . . . .	327
3.5.1.17	set_default_datareader_qos. . . . .	328
3.5.1.18	set_listener . . . . .	329
3.5.1.19	set_qos . . . . .	332
3.5.2	Subscription Type Specific Classes. . . . .	333
3.5.2.1	Interface DataReader . . . . .	334
3.5.2.2	create_querycondition. . . . .	338
3.5.2.3	create_readcondition. . . . .	340
3.5.2.4	delete_contained_entities . . . . .	341
3.5.2.5	delete_readcondition. . . . .	342
3.5.2.6	enable (inherited) . . . . .	343
3.5.2.7	get_key_value (abstract). . . . .	344
3.5.2.8	get_listener . . . . .	344
3.5.2.9	get_liveliness_changed_status . . . . .	345
3.5.2.10	get_matched_publication_data . . . . .	346
3.5.2.11	get_matched_publications . . . . .	347
3.5.2.12	get_qos . . . . .	349
3.5.2.13	get_requested_deadline_missed_status . . . . .	350
3.5.2.14	get_requested_incompatible_qos_status. . . . .	351
3.5.2.15	get_sample_lost_status . . . . .	352
3.5.2.16	get_sample_rejected_status . . . . .	353
3.5.2.17	get_status_changes (inherited) . . . . .	354
3.5.2.18	get_statuscondition (inherited) . . . . .	354
3.5.2.19	get_subscriber. . . . .	354
3.5.2.20	get_subscription_matched_status . . . . .	355
3.5.2.21	get_topicdescription . . . . .	356
3.5.2.22	lookup_instance (abstract) . . . . .	357
3.5.2.23	read (abstract) . . . . .	357
3.5.2.24	read_instance (abstract) . . . . .	358
3.5.2.25	read_next_instance (abstract) . . . . .	358
3.5.2.26	read_next_instance_w_condition (abstract) . . . . .	358
3.5.2.27	read_next_sample (abstract). . . . .	359
3.5.2.28	read_w_condition (abstract) . . . . .	359
3.5.2.29	return_loan (abstract) . . . . .	359
3.5.2.30	set_listener . . . . .	360
3.5.2.31	set_qos . . . . .	362
3.5.2.32	take (abstract) . . . . .	364
3.5.2.33	take_instance (abstract) . . . . .	364
3.5.2.34	take_next_instance (abstract) . . . . .	364
3.5.2.35	take_next_instance_w_condition (abstract) . . . . .	365
3.5.2.36	take_next_sample (abstract) . . . . .	365
3.5.2.37	take_w_condition (abstract) . . . . .	365

3.5.2.38	wait_for_historical_data . . . . .	366
3.5.2.39	Interface FooDataReader . . . . .	367
3.5.2.40	create_querycondition (inherited). . . . .	372
3.5.2.41	create_readcondition (inherited). . . . .	372
3.5.2.42	delete_contained_entities (inherited) . . . . .	372
3.5.2.43	delete_readcondition (inherited). . . . .	372
3.5.2.44	enable (inherited) . . . . .	373
3.5.2.45	get_key_value. . . . .	373
3.5.2.46	get_listener (inherited) . . . . .	374
3.5.2.47	get_liveliness_changed_status (inherited) . . . . .	374
3.5.2.48	get_matched_publication_data (inherited) . . . . .	375
3.5.2.49	get_matched_publications (inherited) . . . . .	375
3.5.2.50	get_qos (inherited) . . . . .	375
3.5.2.51	get_requested_deadline_missed_status (inherited) . . . . .	375
3.5.2.52	get_requested_incompatible_qos_status (inherited). . . . .	375
3.5.2.53	get_sample_lost_status (inherited). . . . .	376
3.5.2.54	get_sample_rejected_status (inherited) . . . . .	376
3.5.2.55	get_status_changes (inherited) . . . . .	376
3.5.2.56	get_statuscondition (inherited) . . . . .	376
3.5.2.57	get_subscriber (inherited). . . . .	377
3.5.2.58	get_subscription_match_status (inherited). . . . .	377
3.5.2.59	get_topicdescription (inherited) . . . . .	377
3.5.2.60	lookup_instance . . . . .	377
3.5.2.61	read . . . . .	378
3.5.2.62	read_instance . . . . .	382
3.5.2.63	read_next_instance . . . . .	384
3.5.2.64	read_next_instance_w_condition . . . . .	386
3.5.2.65	read_next_sample . . . . .	388
3.5.2.66	read_w_condition . . . . .	388
3.5.2.67	return_loan . . . . .	390
3.5.2.68	set_listener (inherited) . . . . .	392
3.5.2.69	set_qos (inherited) . . . . .	392
3.5.2.70	take . . . . .	392
3.5.2.71	take_instance . . . . .	394
3.5.2.72	take_next_instance . . . . .	395
3.5.2.73	take_next_instance_w_condition . . . . .	397
3.5.2.74	take_next_sample . . . . .	399
3.5.2.75	take_w_condition . . . . .	399
3.5.2.76	wait_for_historical_data (inherited). . . . .	400
3.5.3	Class DataSample . . . . .	401
3.5.4	Class SampleInfo. . . . .	401
3.5.4.1	SampleInfo . . . . .	401



3.5.5	SubscriberListener Interface	405
3.5.5.1	on_data_available (inherited)	406
3.5.5.2	on_data_on_readers	406
3.5.5.3	on_liveliness_changed (inherited)	408
3.5.5.4	on_requested_deadline_missed (inherited)	408
3.5.5.5	on_requested_incompatible_qos (inherited)	408
3.5.5.6	on_sample_lost (inherited)	408
3.5.5.7	on_sample_rejected (inherited)	409
3.5.5.8	on_subscription_matched (inherited,)	409
3.5.6	DataReaderListener interface	409
3.5.6.1	on_data_available	410
3.5.6.2	on_liveliness_changed	412
3.5.6.3	on_requested_deadline_missed	413
3.5.6.4	on_requested_incompatible_qos	414
3.5.6.5	on_sample_lost	415
3.5.6.6	on_sample_rejected	415
3.5.6.7	on_subscription_matched (abstract)	416
3.5.7	Interface ReadCondition	417
3.5.7.1	get_datareader	418
3.5.7.2	get_instance_state_mask	419
3.5.7.3	get_sample_state_mask	419
3.5.7.4	get_trigger_value (inherited)	420
3.5.7.5	get_view_state_mask	420
3.5.8	Interface QueryCondition	421
3.5.8.1	get_datareader (inherited)	422
3.5.8.2	get_instance_state_mask (inherited)	423
3.5.8.3	get_query_parameters	423
3.5.8.4	get_query_expression	424
3.5.8.5	get_sample_state_mask (inherited)	425
3.5.8.6	get_trigger_value (inherited)	425
3.5.8.7	get_view_state_mask (inherited)	425
3.5.8.8	set_query_parameters	425

## **Appendix A Quality Of Service 429**

Affected Entities	429
Basic Usage	429
DataReaderQos	431
DataWriterQos	434
DomainParticipantFactoryQos	437
DomainParticipantQos	438
PublisherQos	439
SubscriberQos	440

TopicQos . . . . .	442
<b>Appendix B API Constants and Types</b>	<b>445</b>
Duration and Time. . . . .	445
Status to Support Listeners and Conditions. . . . .	447
States . . . . .	449
QosPolicy. . . . .	450
<b>Appendix C Platform Specific Model IDL Interface</b>	<b>455</b>
dds_dcps.idl . . . . .	455
Foo.idl . . . . .	484
<b>Appendix D SampleStates, ViewStates and InstanceStates</b>	<b>489</b>
SampleInfo Class. . . . .	489
sample_state. . . . .	489
instance_state. . . . .	490
view_state . . . . .	492
State Masks . . . . .	494
Operations Concerning States . . . . .	495
<b>Appendix E Interface Inheritance</b>	<b>499</b>
<b>Appendix F Listeners, Conditions and Waitsets</b>	<b>501</b>
Communication Status Event . . . . .	503
Listeners. . . . .	506
Conditions and Waitsets . . . . .	508
StatusCondition Trigger State . . . . .	510
ReadCondition and QueryCondition Trigger State . . . . .	511
GuardCondition Trigger State . . . . .	511
<b>Appendix G Topic Definitions</b>	<b>513</b>
Topic Definition Example . . . . .	513
Complex Topics. . . . .	514
IDL Preprocessor. . . . .	514
<b>Appendix H DCPS Queries and Filters</b>	<b>519</b>
SQL Grammar in BNF . . . . .	519
SQL Token Expression . . . . .	520
SQL Examples. . . . .	521
<b>Bibliography</b>	<b>525</b>
<b>Glossary</b>	<b>529</b>
<b>Index</b>	<b>533</b>

# List of Figures

Figure 1	Java Reference Guide Document Structure	3
Figure 2	DCPS Module Composition	17
Figure 3	DCPS Infrastructure Module's Class Model	18
Figure 4	DCPS Domain Module's Class Model	20
Figure 5	DCPS Topic-Definition Module's Class Model	21
Figure 6	Typed Classes for Data Type "Foo" Pre-processor Generation	22
Figure 7	DCPS Publication Module Class Model	23
Figure 8	DCPS Subscription Module's Class Model	24
Figure 9	DCPS Infrastructure Module's Class Model	28
Figure 10	QosPolicy Settings	36
Figure 11	DCPS Listeners	84
Figure 12	DCPS Status Values	87
Figure 13	DCPS WaitSets	102
Figure 14	DCPS Conditions	108
Figure 15	DCPS Domain Module's Class Model	117
Figure 16	DCPS Topic-Definition Module's Class Model	189
Figure 17	Pre-processor Generation of the Typed Classes for Data Type "Foo"	190
Figure 18	DCPS Publication Module's Class Model	222
Figure 19	DCPS Subscription Module's Class Model	311
Figure 20:	sample_state for a Single Sample State Chart	490
Figure 21:	State Chart of the instance_state for a Single Instance	492
Figure 22:	view_state for a Single Instance State Chart	493
Figure 23	DCPS Inheritance	499
Figure 24:	Plain Communication Status State Chart	504
Figure 25:	Read Communication Status DataReader Statecraft	505
Figure 26:	Subscriber Statecraft for a Read Communication Status	505
Figure 27:	DCPS Listeners	507
Figure 28:	DCPS WaitSets	508
Figure 29	DCPS Conditions	509
Figure 30:	Blocking Behaviour of a Waitset State Chart	510



# Preface

## About the Java Reference Guide

The *Java Reference Guide* provides details of the OpenSplice DDS (*Subscription Paradigm for the Logical Interconnection of Concurrent Engines*) Application Programming Interfaces for the Java language.

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

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

## Organisation

The *Java Reference Guide* is organised into the following topics.

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

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, *Interface 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 Java Reference Guide.



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



Item contains helpful hint or special information.



Information applies to Windows (*e.g.* XP, 2003, Windows 7) only.



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



C language specific



C++ language specific



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 xxiii, are 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](mailto:crc@prismtech.com) (Customer Response Center)

Sales enquiries:

[sales@prismtech.com](mailto:sales@prismtech.com)





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 geometric pattern. The lighting is soft, and the overall color palette is muted, with a slight purple or blue tint. The word "INTRODUCTION" is centered in the upper half of the image in a dark blue, serif font.

# INTRODUCTION



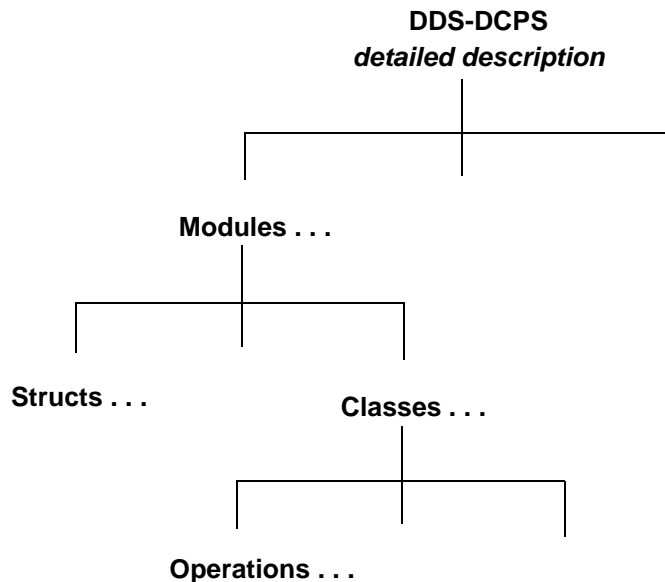
# About the Java Reference Guide

## Document Structure

The Java 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*).

- 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 Java Reference Guide Document Structure**

## Operations

Operations are described in detail in the class they are implemented in. This means inherited operations, only refer to the operation in the class they are inherited from. Abstract operations only refer to the type specific implementations in their respective derived class. An exception is made for the abstract operations (implemented as an interface), which must be implemented in the application. Abstract interfaces are described in detail in this manual. In the API description header file, the inherited and abstract operations are commented out since they are not implemented in this class.

The background of the page is a close-up, low-angle photograph of a computer keyboard. The keys are white and slightly worn, with some characters visible like 'I', 'J', and 'P'. A white grid of thin lines is overlaid on the entire image, creating a perspective effect that makes the grid lines converge towards the top right. The overall color palette is a mix of light and dark purples and blues, giving it a modern, tech-oriented feel.

# API REFERENCE



# 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 `int`, which possible value is defined in the next table. See Section 1.1, IDL Mapping to Java, on page 8 for an explanation of IDL constants.*

**Table 1 Return Codes**

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>&lt;Entity&gt;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

**Table 1 Return Codes**

Return Code	Description
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 QoS Policies that control the behaviour of the object in question). There is no precondition that could be changed to make the operation succeed.  In Java, this code can never be returned!
RETCODE_NO_DATA	Indicates a situation where the operation did not return any data

Return Codes are located in the package `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. All constants and types are listed in Appendix B, *API Constants and Types*.

The return code `RETCODE_ILLEGAL_OPERATION` can never be returned in Java: it indicates that you try to invoke an operation on the wrong class, which in a language like Java, that natively supports Object Orientation by supporting the concept of classes with internal operations, is never possible.

## 1.1 IDL Mapping to Java

### 1.1.1 IDL-Constant Mapping

Constants, declared outside the scope of a IDL interface, are mapped to a public interface with the same name as the constant and containing a field, named `value`, that holds the constant's value. For example the IDL constant:

```
module DDS {
    const ReturnCode_t RETCODE_OK = 0;
}
```

is mapped to the following interface class:

```
Package DDS;
```



```
public interface RETCODE_OK {
    public static final int value = (int)(0);
}
```

The value of the constant can be retrieved with:

```
RETCODE_OK.value
```

A few examples of the usage of constants:

- use the *OR* operation to compose a value:

```
condition.set_enabled_statuses (
    INCONSISTENT_TOPIC_STATUS.value |
    SAMPLE_REJECTED_STATUS.value);
```

- use the *AND* operation to check a value:

```
if ((status & INCONSISTENT_TOPIC_STATUS.value) ==
    INCONSISTENT_TOPIC_STATUS.value) {
    error = true;
}
```

### 1.1.2 IDL-Sequence Mapping

An IDL sequence is mapped to a Java array with the same name. The holder class, see section 1.5 on page 11, for the sequence is generated too. For example the IDL sequence:

```
typedef sequence<long> Data
```

is mapped to the following holder class:

```
public final class DataHolder
{
    public int[] value = null;
    public DataHolder () { }
    public DataHolder (int[] initialValue)
    {
        value = initialValue;
    }
}
```

The array can be initiated using the constructor:

```
int[] data = new int[10];
DataHolder dataHolder = new DataHolder(data);
```

To reference to an element of the array use for example:

```
int x = dataHolder.value[2];
```

The length of the array is found using:

```
int length = dataHolder.value.length;
```

## 1.2 Thread Safety

All operations are thread safe.

## 1.3 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 service will conditionally set the signal handlers when creating the `DomainParticipantFactory`, which is the first call to `DDS.DomainParticipantFactory.get_instance()` for Java.

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

The Java Virtual Machine uses its own signal handlers internally. Which signals are caught by it, depends on the platform it is running on. The Splice-DDS signal handlers conflict with the ones used internally by the Java Virtual Machine. To avoid this problem the signal-chaining facility of the virtual machine must be used. The signal-chaining facility is installed by loading the shared library `libjsig.so` before `libc`, `libthead` and/or `libpthread`. The `libjsig.so` library is provided with the Java virtual machine. This library ensures that calls such as `signal()`, `sigset()` and `sigaction()` are intercepted so that they do not actually replace the Java virtual machine signal handlers. Instead, signal handlers of the application are “chained” behind the handlers of the Java virtual machine. When any of these signals are raised and found not to be targeted at the Java virtual machine, the signals invoke the signal handlers installed by the application. When just the Data Distribution System is used without any application signal handlers, no action needs to be taken.

The library `libjsig.so` can be loaded before the Java application by using the `LD_PRELOAD` environment variable, for example:

```
LD_PRELOAD = <libjvm directory>/libjsig.so
export LD_PRELOAD
```

## 1.4 Memory Management

When objects are being created, they will occupy memory resources. Release of the memory resources is the responsibility of the Java garbage collector. The memory of an object is released, after all references to this object have run out of scope or have explicitly been removed (set to null).

## 1.5 Parameter Passing

Support for passing out and inout parameter in a function call requires the use of additional “holder” classes. These classes are available for all of the data types in the DDS package and are generated for all user-defined IDL types, except those defined by typedefs. For user-defined IDL types, the holder class name is constructed by appending “Holder” to the mapped Java name of the type, i.e. FooHolder for user-defined type Foo.

Each holder class has a default constructor and has a public instance member, named value, which is the typed value. The default constructor sets the value field to null for object references.

The Holder class for the type DataReaderSeq is shown below:

```
public final class DataReaderSeqHolder
{
    public DDS.DataReader value[] = null;
    public DataReaderSeqHolder ()
    {
    }
}
```

Example of usage of the DataReaderSeqHolder class:

```
DDS.DataReader reader;
DDS.DataReaderSeqHolder readersholder = new
DDS.DataReaderSeqHolder;
// Get data readers objects
subscriber.get_datareaders (readersholder, DDS.ANY_STATE.value,
DDS.ANY_VIEW.value,
DDS.ANY_INSTANCE_STATE.value);
// Get first data reader object from the sequence
if(readersholder.value[0] != null)
    reader = readersholder.value[0];
```

In this example a holder is declared for the type DataReaderSeq. After the call to the function subscriber.get\_datareaders, the readersholder contains a reference to an array of DataReader objects. A reader is retrieved in the last line of the example code.

The Holder classes defined in DCPS are listed in Table 2:

**Table 2 DCPS Holder Classes**

BuiltinTopicKey_tHolder	PresentationQosPolicyAccessScopeKindHolder
ConditionHolder	PresentationQosPolicyHolder
ConditionSeqHolder	PublicationBuiltinTopicDataHolder
ContentFilteredTopicHolder	PublicationMatchStatusHolder
DataReaderHolder	PublisherHolder
DataReaderListenerHolder	PublisherListenerHolder
DataReaderQosHolder	PublisherQosHolder
DataReaderSeqHolder	QosPolicyCountHolder
DataWriterHolder	QosPolicyCountSeqHolder
DataWriterListenerHolder	QueryConditionHolder
DataWriterQosHolder	ReadConditionHolder
DeadlineQosPolicyHolder	ReaderDataLifecycleQosPolicyHolder
DestinationOrderQosPolicyHolder	ReliabilityQosPolicyHolder
DestinationOrderQosPolicyKindHolder	ReliabilityQosPolicyKindHolder
DomainParticipantFactoryHolder	RequestedDeadlineMissedStatusHolder
DomainParticipantHolder	RequestedIncompatibleQosStatusHolder
DomainParticipantListenerHolder	ResourceLimitsQosPolicyHolder
DomainParticipantQosHolder	SampleInfoHolder
DurabilityQosPolicyHolder	SampleInfoSeqHolder
DurabilityQosPolicyKindHolder	SampleLostStatusHolder
Duration_tHolder	SampleRejectedStatusHolder
EntityFactoryQosPolicyHolder	SampleRejectedStatusKindHolder
EntityHolder	SampleStateSeqHolder
GroupDataQosPolicyHolder	StatusConditionHolder
GuardConditionHolder	StringSeqHolder
HistoryQosPolicyHolder	SubscriberHolder
HistoryQosPolicyKindHolder	SubscriberListenerHolder
InconsistentTopicStatusHolder	SubscriberQosHolder
InstanceHandleSeqHolder	SubscriptionBuiltinTopicDataHolder
InstanceStateSeqHolder	SubscriptionMatchStatusHolder
LatencyBudgetQosPolicyHolder	Time_tHolder
LifespanQosPolicyHolder	TimeBasedFilterQosPolicyHolder
ListenerHolder	TopicBuiltinTopicDataHolder
LivelinessChangedStatusHolder	TopicDataQosPolicyHolder

**Table 2 DCPS Holder Classes (Continued)**

LivelinessLostStatusHolder	TopicDescriptionHolder
LivelinessQosPolicyHolder	TopicHolder
LivelinessQosPolicyKindHolder	TopicListenerHolder
MultiTopicHolder	TopicQosHolder
OfferedDeadlineMissedStatusHolder	TopicSeqHolder
OfferedIncompatibleQosStatusHolder	TransportPriorityQosPolicyHolder
OwnershipQosPolicyHolder	TypeSupportHolder
OwnershipQosPolicyKindHolder	UserDataQosPolicyHolder
OwnershipStrengthQosPolicyHolder	ViewStateSeqHolder
ParticipantBuiltinTopicDataHolder	WaitSetHolder
PartitionQosPolicyHolder	WriterDataLifecycleQosPolicyHolder

## 1.6 Casting of Objects

Instances of classes can also be cast to instances of other classes, with one restriction: the class of the object you're casting and the class you're casting it to must be related by inheritance. So a class can be cast to a subclass, not to any random class.

For any user-defined interface a Helper class is generated with the method narrow() to cast to the derived class type. The following Java code is generated:

```
abstract public class <typename> Helper {
    public static <typename> narrow(java.lang.Object obj);
}
```

The static method narrow(), defined in the Helper class, is a method to cast a superclass object to the <typename> object.

For the user-defined structure Foo, the following Helper classes are generated:

- FooDataWriterHelper
- FooDataReaderHelper
- FooTypeSupportHelper

The example below demonstrates the casting of an DataReader type to a FooDataReader type by calling the function narrow:

```
// Create the DomainParticipant participant first
...
// The participant creates a new subscriber
DDS.Subscriber subscriber =
    participant.create_subscriber(subqos, sublistener);
// The subscriber creates a new FooDataReader
```

```
DS.DataReader reader = subscriber.create_datareader(topic,
    drqos, drlistener);
// The DataReader is cast to a FooDataReader,
// using the static method narrow() in the Helper class
FooDataReader foo_reader =
    FooDataReaderHelper.narrow(reader);
```

## 1.7 Listeners Interfaces

The `Listener` 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 `Listener` is related to changes in communication status.

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. 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`
- `ExtDomainParticipantListener`
- `TopicListener`
- `ExtTopicListener`
- `PublisherListener`
- `DataWriterListener`
- `SubscriberListener`
- `DataReaderListener`

### *Example*

For example, 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`. This interface must be implemented by the application. 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.

## 1.8 Inheritance of Abstract Operations

This manual is based on:

- PIM part of the DDS-DCPS specification (to describe the modules);
- PSM part of the DDS-DCPS specification (to describe the classes and operations).

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

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.



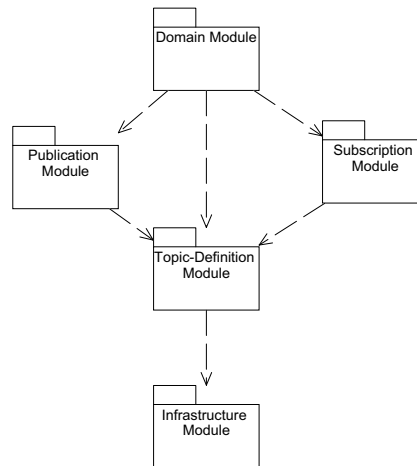


## CHAPTER

# 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 and operations are described in the next chapter.*



**Figure 2 DCPS Module Composition**

## 2.1 Functionality

The modules have the following function in the Data Distribution Service:

**Infrastructure 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** - Contains the DomainParticipant class, which is the entry point of the application and DomainParticipantListener interface;

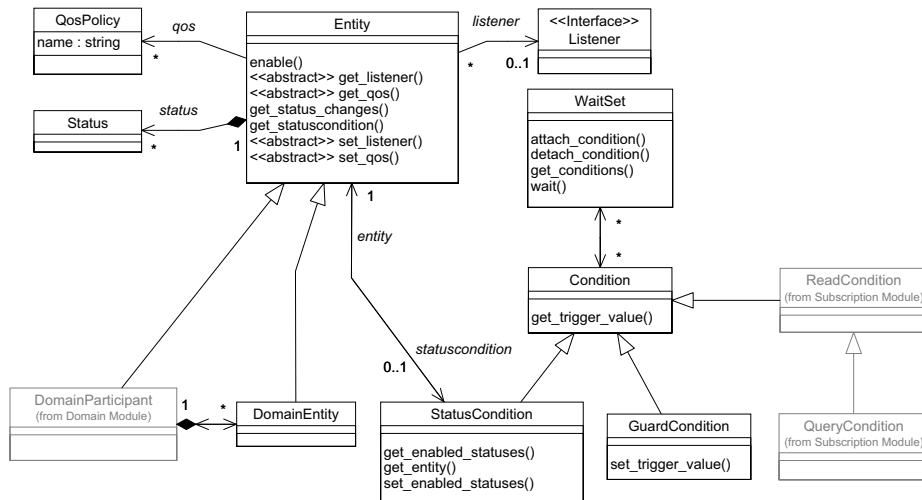
**Topic-Definition Module** - Contains the Topic, ContentFilteredTopic and MultiTopic classes. It also contains the TopicListener interface and all support to define Topic objects and assign QosPolicy settings to them;

**Publication Module** - Contains the Publisher and DataWriter classes. It also contains the PublisherListener and DataWriterListener interfaces;

**Subscription 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, class)
- Listener (interface)

- `Status` (abstract, class)
- `WaitSet`
- `Condition`
- `GuardCondition`
- `StatusCondition`

## 2.3 Domain Module

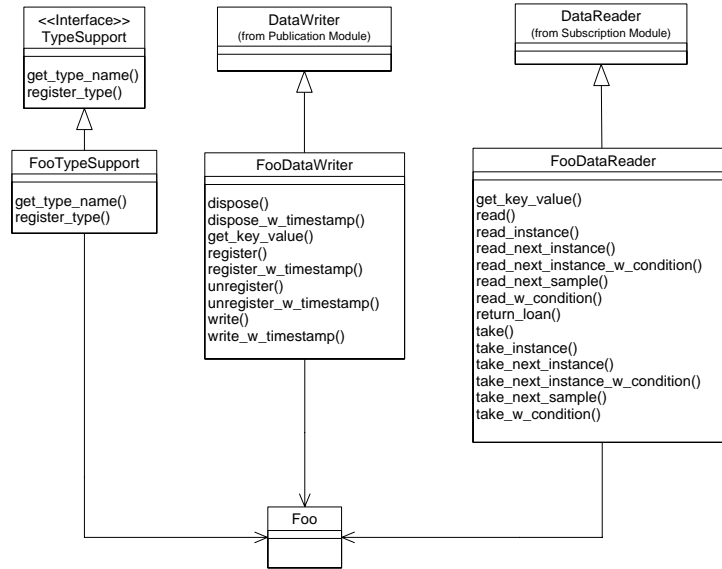
This module contains the class `DomainParticipant`, which acts as an entry point of the Data Distribution Service and acts as a factory for many of the classes. The `DomainParticipant` also acts as a container for the other objects that make up the Data Distribution Service. It isolates applications 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.

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



“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:

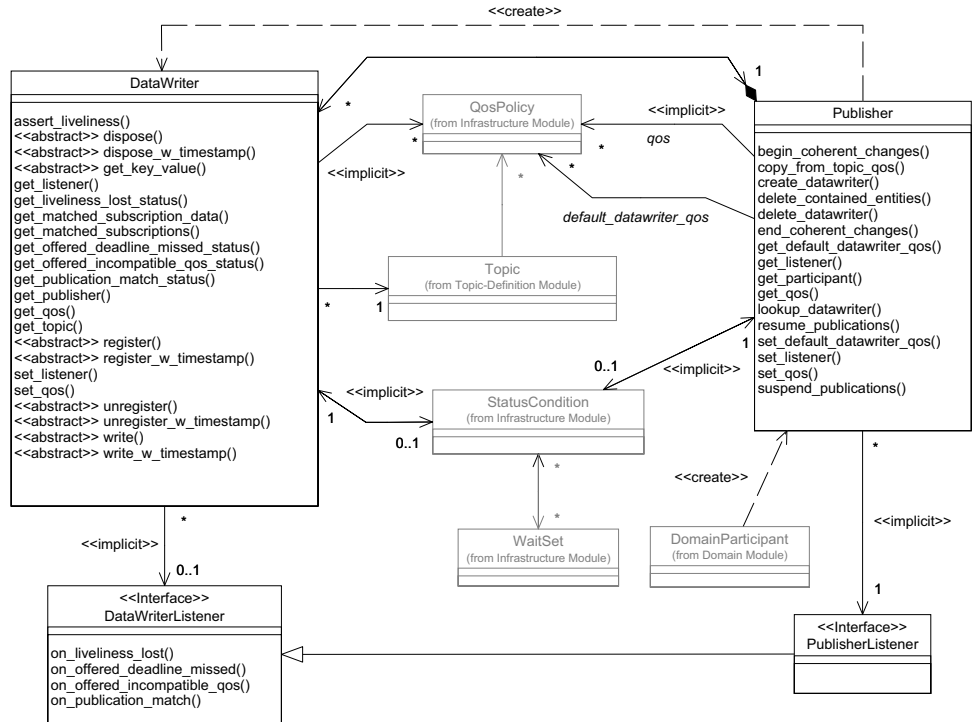
- `TypeSupport` (abstract)
- `FooTypeSupport`



**Figure 6 Typed Classes for Data Type “Foo” 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 Class Model**

This module contains the following classes:

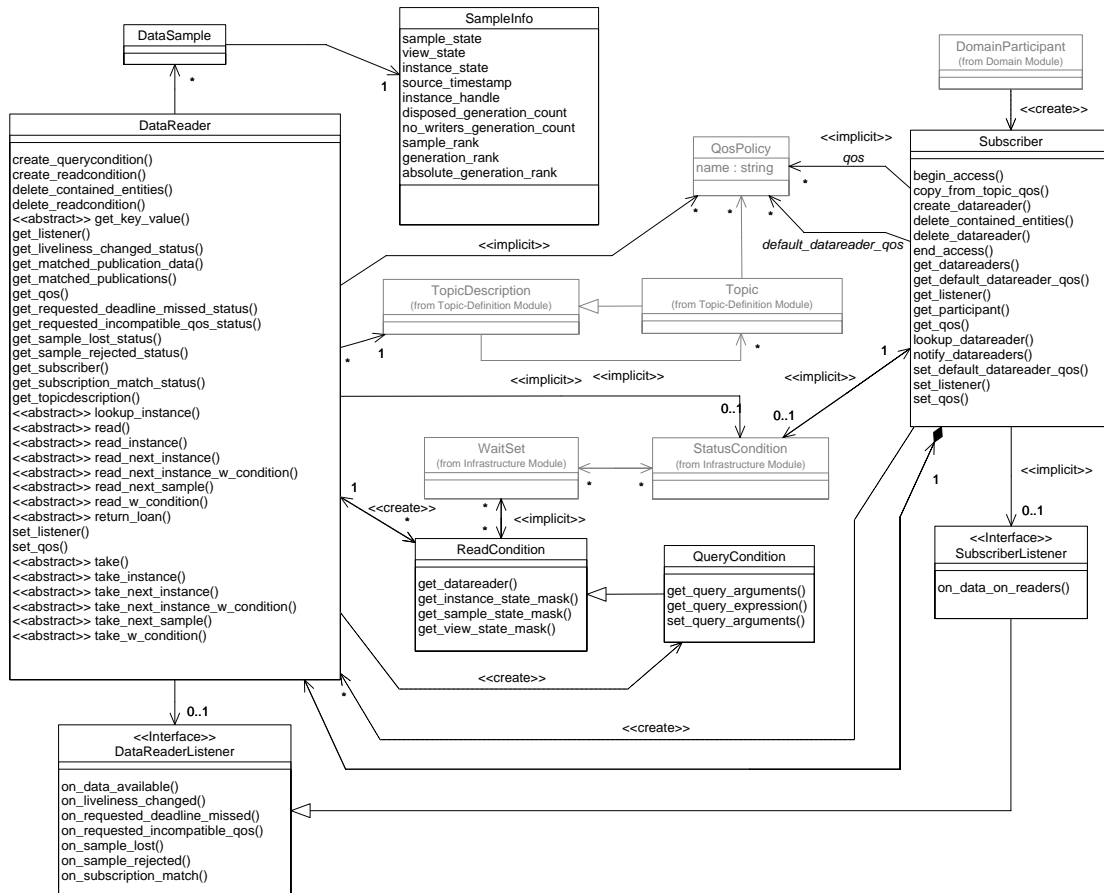
- Publisher
- Publication type specific classes
- PublisherListener (interface)
- DataWriterListener (interface)

“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 (class)
- 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

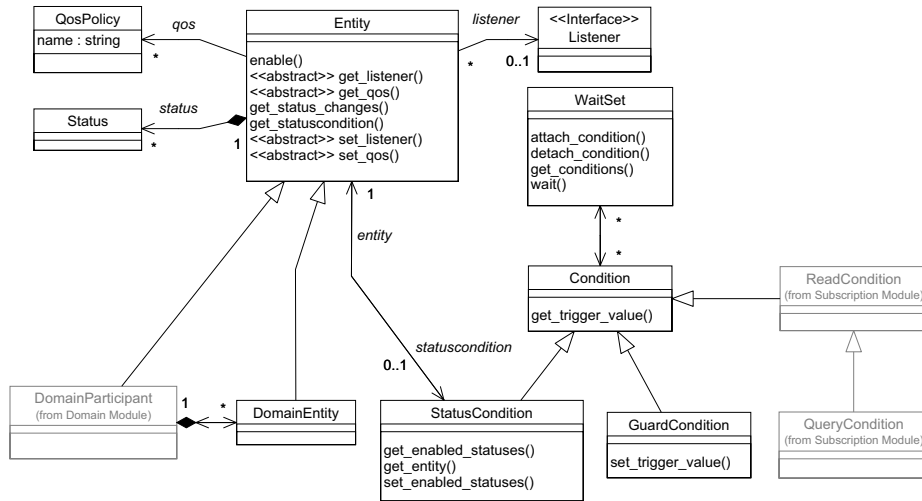
*This chapter describes, for each module, its classes and operations in detail. Each module consists of several classes as defined at PIM level in the DDS-DCPS specification. Some of the classes are implemented as a struct in the PSM. Some of the other classes are abstract, which means they contain some 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 (interface)
- DomainEntity (abstract)
- QosPolicy (abstract, class)
- Listener (interface)
- Status (abstract, class)
- WaitSet
- Condition
- GuardCondition
- StatusCondition

### 3.1.1 Interface Entity

Entity is the interface for all the DCPS objects. It acts as a generic interface for Entity objects.

The interface description is as follows:

```

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

```

// public int
//     set_qos
//         (EntityQos qos);
// public int
//     get_qos
//         (EntityQosHolder qos);
// public int
//     set_listener
//         (Listener a_listener,
//          int mask);
// public Listener
//     get_listener
//         (void);
//
// implemented API operations
//
    public int
        enable
            (void);
    public StatusCondition
        get_statuscondition
            (void);
    public int
        get_status_changes
            (void);
    public long
        get_instance_handle
            (void);

};

```

The following 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

```

import DDS.*;
public int
    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

*int* - 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 lookup<sup>1</sup> 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

```
import DDS.*;
public long
    get_instance_handle
        (void);
```

### Description

This operation returns the `instance_handle` of the built-in topic sample that represents the specified `Entity`.

### Parameters

<none>

### Return Value

`long` - Result value is the `instance_handle` of the built-in topic sample that represents the state of this `Entity`.

## Detailed Description

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

Some `Entities` (i.e. `Publisher` and `Subscriber`) do not have a corresponding built-in 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 122.)

### 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

```
import DDS.*;
public Listener
    get_listener
        (void);
```

### 3.1.1.4 `get_qos` (abstract)

This abstract operation is defined as a generic operation to access an object 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

```
import DDS.*;
public int
    get_qos
        (EntityQosHolder qos);
```

### 3.1.1.5 `get_status_changes`

#### Scope

```
DDS.Entity
```



## Synopsis

```
import DDS.*;
public int
    get_status_changes
        (void);
```

## Description

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

## Parameters

<none>

## Return Value

*int* - 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 resulting status mask. 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

```
import DDS.*;
public StatusCondition
    get_statuscondition
        (void);
```

#### Description

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

#### Parameters

<none>

#### Return Value

*StatusCondition* - Result value is 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

```
import DDS.*;
public int
    set_listener
```

```
(Listener a_listener,
 int mask);
```

### 3.1.1.8 set\_qos (abstract)

This abstract operation is defined as a generic operation to modify an object 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

```
import DDS.*;
public int
    set_qos
        (EntityQos qos);
```

### 3.1.2 Class DomainEntity

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.

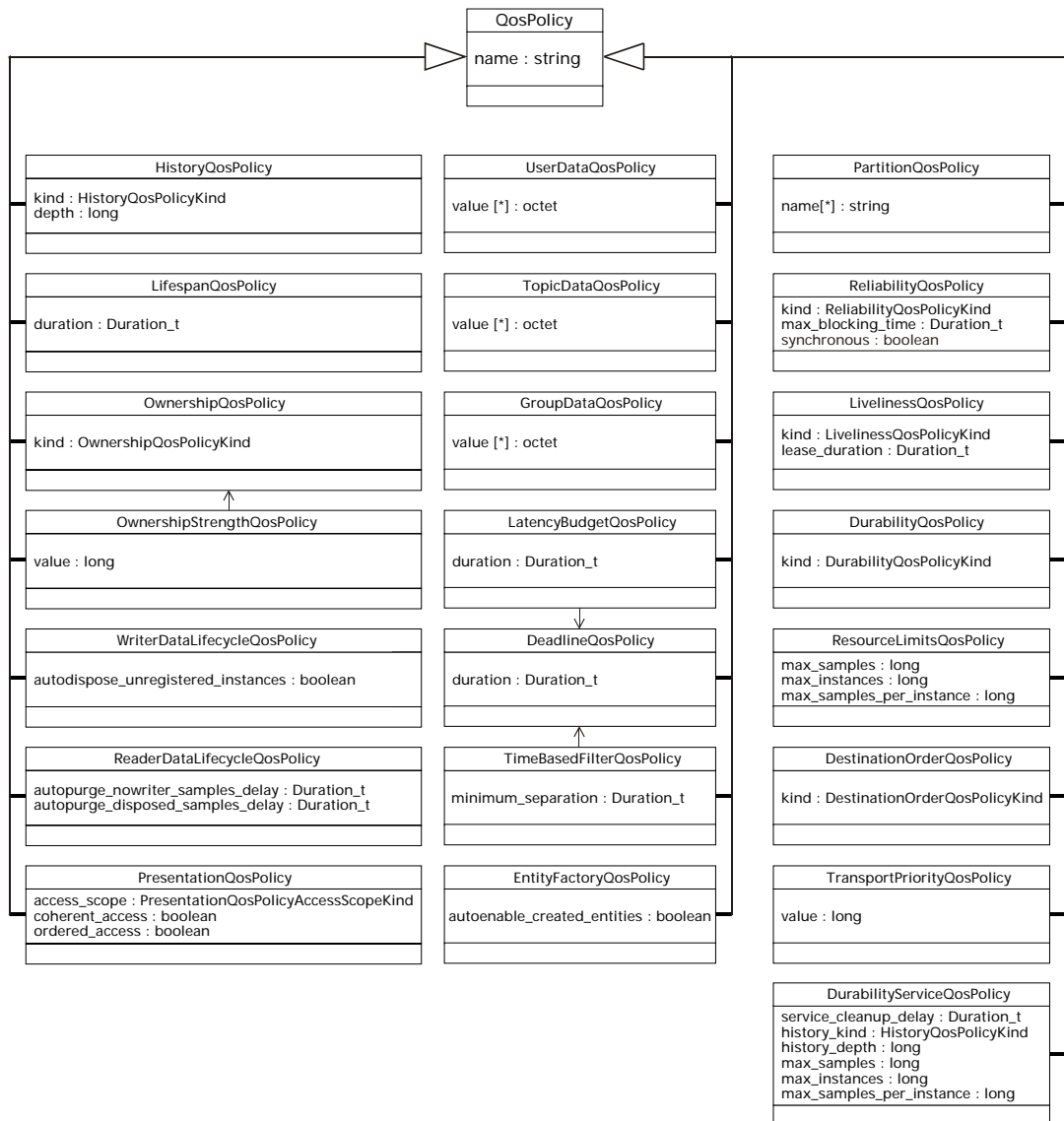
### 3.1.3 Class QosPolicy

Each `Entity` provides an `<Entity>Qos` class that implements the basic mechanism for an application to specify Quality of Service attributes. This class consists of `Entity` specific `QosPolicy` attributes. `QosPolicy` attributes are class 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 class that is set atomically so that it represents a coherent set of `QosPolicy` attributes.

Compound classes 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 that 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. In 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 the `QosPolicy` classes is as follows:

```
// class <Entity>Qos
//
package DDS;
//
// class <name>QosPolicy
//
public final class UserDataQosPolicy
{ public byte value[]; };

public final class TopicDataQosPolicy
{ public byte value[]; };

public final class GroupDataQosPolicy
{ public byte value[]; };

public final class TransportPriorityQosPolicy
{ public int value; };

public final class LifespanQosPolicy
{ public Duration_t duration; };

public class DurabilityQosPolicyKind
{ public static final DurabilityQosPolicyKind
    VOLATILE_DURABILITY_QOS;
```

```

        public static final DurabilityQosPolicyKind
            TRANSIENT_LOCAL_DURABILITY_QOS;
        public static final DurabilityQosPolicyKind
            TRANSIENT_DURABILITY_QOS;
        public static final DurabilityQosPolicyKind
            PERSISTENT_DURABILITY_QOS; };

public final class DurabilityQosPolicy
{ public DurabilityQosPolicyKind kind; };

public class PresentationQosPolicyAccessScopeKind
{ public static final PresentationQosPolicyAccessScopeKind
    INSTANCE_PRESENTATION_QOS;
    public static final PresentationQosPolicyAccessScopeKind
    TOPIC_PRESENTATION_QOS;
    public static final PresentationQosPolicyAccessScopeKind
    GROUP_PRESENTATION_QOS; };

public final class PresentationQosPolicy
{ public PresentationQosPolicyAccessScopeKind access_scope;
    public boolean coherent_access;
    public boolean ordered_access; };

public final class DeadlineQosPolicy
{ public Duration_t period; };

public final class LatencyBudgetQosPolicy
{ public Duration_t duration; };

public class OwnershipQosPolicyKind
{ public static final OwnershipQosPolicyKind
    SHARED_OWNERSHIP_QOS;
    public static final OwnershipQosPolicyKind
    EXCLUSIVE_OWNERSHIP_QOS; };

public final class OwnershipQosPolicy
{ public OwnershipQosPolicyKind kind; };

public final class OwnershipStrengthQosPolicy
{ public int value; };

public class LivelinessQosPolicyKind
{ public static final LivelinessQosPolicyKind
    AUTOMATIC_LIVELINESS_QOS;
    public static final LivelinessQosPolicyKind
    MANUAL_BY_PARTICIPANT_LIVELINESS_QOS;
    public static final LivelinessQosPolicyKind
    MANUAL_BY_TOPIC_LIVELINESS_QOS; };

public final class LivelinessQosPolicy

```

```

    { public LivelinessQosPolicyKind kind;
      public Duration_t lease_duration; };

public final class TimeBasedFilterQosPolicy
{ public Duration_t minimum_separation; };

public final class PartitionQosPolicy
{ public String name[]; };

public class ReliabilityQosPolicyKind
{ public static final ReliabilityQosPolicyKind
    BEST_EFFORT_RELIABILITY_QOS;
  public static final ReliabilityQosPolicyKind
    RELIABLE_RELIABILITY_QOS; };

public final class ReliabilityQosPolicy
{ public ReliabilityQosPolicyKind kind;
  public Duration_t max_blocking_time;
  public boolean synchronous; };

public class DestinationOrderQosPolicyKind
{ public static final DestinationOrderQosPolicyKind
    BY_RECEPTION_TIMESTAMP_DESTINATIONORDER_QOS;
  public static final DestinationOrderQosPolicyKind
    BY_SOURCE_TIMESTAMP_DESTINATIONORDER_QOS; };

public final class DestinationOrderQosPolicy
{ public DestinationOrderQosPolicyKind kind; };

public class HistoryQosPolicyKind
{ public static final HistoryQosPolicyKind KEEP_LAST_HISTORY_QOS;
  public static final
    HistoryQosPolicyKind KEEP_ALL_HISTORY_QOS; };

public final class HistoryQosPolicy
{ public HistoryQosPolicyKind kind;
  public int depth; };

public final class ResourceLimitsQosPolicy
{ public int max_samples;
  public int max_instances;
  public int max_samples_per_instance; };

public final class EntityFactoryQosPolicy
{ public boolean autoenable_created_entities; };

public final class WriterDataLifecycleQosPolicy
{ public boolean autodispose_unregistered_instances; };

public final class ReaderDataLifecycleQosPolicy

```

```

{ public Duration_t autopurge_nowriter_samples_delay;
  public Duration_t autopurge_disposed_samples_delay; };

public final class DurabilityServiceQosPolicy
{ public Duration_t service_cleanup_delay;
  public HistoryQosPolicyKind history_kind;
  public int history_depth;
  public int max_samples;
  public int max_instances;
  public int max_samples_per_instance; };

public class SchedulingClassQosPolicyKind
{ public static final SchedulingClassQosPolicyKind
  SCHEDULE_DEFAULT;
  public static final SchedulingClassQosPolicyKind
  SCHEDULE_TIMESHARING;
  public static final SchedulingClassQosPolicyKind
  SCHEDULE_REALTIME; };

public class SchedulingClassQosPolicy
{ public SchedulingClassQosPolicyKind kind; };

public class SchedulingPriorityQosPolicyKind
{ public static final SchedulingPriorityQosPolicyKind
  PRIORITY_RELATIVE;
  public static final SchedulingPriorityQosPolicyKind
  PRIORITY_ABSOLUTE; };

public class SchedulingPriorityQosPolicy
{ public SchedulingPriorityQosPolicyKind kind; };

public class SchedulingQosPolicy
{ public SchedulingClassQosPolicy scheduling_class;
  public SchedulingPriorityQosPolicy scheduling_priority_kind;
  public int scheduling_priority; };

```

### Default Attributes

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

**Table 3 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



**Table 3 QosPolicy Default Attributes (Continued)**

<b>QosPolicy</b>	<b>Attribute</b>	<b>Value</b>
lifespan	duration	DURATION_INFINITE
durability	kind	VOLATILE_DURABILITY_QOS
presentation	access_scope	INSTANCE_PRESENTATION_QOS
	coherent_access	false
	ordered_access	false
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

**Table 3 QosPolicy Default Attributes (Continued)**

QosPolicy	Attribute	Value
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
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 4, *QosPolicy Basics*, on page 43:

- 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 4 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

**Table 4 QosPolicy Basics (Continued)**

<b>QosPolicy</b>	<b>Concerns Entity</b>	<b>RxO</b>	<b>Changeable After Enabling</b>
liveliness	Topic DataReader DataWriter	Yes	No
time_based_filter	DataReader	Not applicable	Yes
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 following paragraphs describe each <name>QosPolicy class.

### 3.1.3.1 DeadlineQosPolicy

#### Scope

DDS

#### Synopsis

```
import DDS.*;
```

```
public final class DeadlineQosPolicy
{ public Duration_t period; }
```

## Description

This `QosPolicy` defines the period within which a new sample is expected by the `DDS_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 5 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, *Class 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

```
import DDS.*
public class DestinationOrderQosPolicyKind
{ public static final DestinationOrderQosPolicyKind
    BY_RECEPTION_TIMESTAMP_DESTINATIONORDER_QOS;
    public static final DestinationOrderQosPolicyKind
        BY_SOURCE_TIMESTAMP_DESTINATIONORDER_QOS;
}
public class DestinationOrderQosPolicy
{ public 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 6 Requested/Offered DestinationOrderQoSPolicy**

<div>Requested Offered</div>	BY_RECEPTION_TIMESTAMP	BY_SOURCE_TIMESTAMP
BY_RECEPTION_TIMESTAMP	compatible	Incompatible
BY_SOURCE_TIMESTAMP	compatible	compatible

Whether communication is established, it 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**

```
import DDS.*;
public class DurabilityQoSPolicyKind
{
    public static final DurabilityQoSPolicyKind
        VOLATILE_DURABILITY_QOS;
    public static final DurabilityQoSPolicyKind
        TRANSIENT_LOCAL_DURABILITY_QOS;
    public static final DurabilityQoSPolicyKind
        TRANSIENT_DURABILITY_QOS;
    public static final DurabilityQoSPolicyKind
        PERSISTENT_DURABILITY_QOS;
```



```
}  
public class DurabilityQosPolicy  
{  
    public 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` - currently behaves identically to the `TRANSIENT_DURABILITY_QOS`, except for its `RxO` properties. 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` and the `service_cleanup_delay` set to 0 on the durability service. This is because for `TRANSIENT_LOCAL`, the data should only remain available for late-joining readers during the lifetime of its source writer, so it is not required to survive after its source writer has been deleted. Since the deletion of a writer implicitly

unregisters all its instances, an `autodispose_unregistered_instances` value of `TRUE` will also dispose the affected data from the durability store, and thus prevent it from remaining available to late joining readers.

- **`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 7 Requested/Offered DurabilityQosPolicy**

Requested Offered	<b>VOLATILE</b>	<b>TRANSIENT_ LOCAL</b>	<b>TRANSIENT</b>	<b>PERSISTENT</b>
<b>VOLATILE</b>	compatible	INcompatible	INcompatible	INcompatible
<b>TRANSIENT_LOCAL</b>	compatible	compatible	INcompatible	INcompatible
<b>TRANSIENT</b>	compatible	compatible	compatible	INcompatible
<b>PERSISTENT</b>	compatible	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_LOCAL_DURABILITY_QOS`, `DDS_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

```
import DDS.*;
public final class DurabilityServiceQosPolicy
{
    public Duration_t service_cleanup_delay;
    public HistoryQosPolicyKind history_kind;
    public int history_depth;
    public int max_samples;
    public int max_instances;
    public int 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.

*int 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.

*int 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.

*int max\_instances* - specifies the maximum number of instances the durability service - manage for the transient or persistent topic data-instances.

*int 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 Section 3.1.3.7, *HistoryQosPolicy*, on page 56 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

```
import DDS.*;
public final class EntityFactoryQosPolicy
{ public 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 a 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 paragraph 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

```
import DDS.*;
public final class GroupDataQosPolicy
{ public byte 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

`byte value[]` - a sequence of bytes 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

```
import DDS.*;
public class HistoryQosPolicyKind
{
    public static final HistoryQosPolicyKind
        KEEP_LAST_HISTORY_QOS;
    public static final HistoryQosPolicyKind
        KEEP_ALL_HISTORY_QOS;
}
public final class HistoryQosPolicy
{
    public HistoryQosPolicyKind kind;
    public int 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`.

*int 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

```
import DDS.*;
public final class LatencyBudgetQosPolicy
{
    public 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 optimize 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 8 LatencyBudgetQosPolicy**

Duration	Compatibility
offered duration < requested duration	compatible
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, *Class QosPolicy*, the paragraph entitled *Requested/Offered*.) Consider a writer with budget `Bw` and a reader with budget `Br`, where  $Bw \leq Br$ . In this case a connection between that reader and that writer is established. Now suppose `Bw` is changed so that  $Bw > Br$ , 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

```
import DDS.*;
public final class LifespanQosPolicy
{
    public 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

```
import DDS.*;
public class LivelinessQosPolicyKind
{
    public static final LivelinessQosPolicyKind
        AUTOMATIC_LIVELINESS_QOS;
    public static final LivelinessQosPolicyKind
        MANUAL_BY_PARTICIPANT_LIVELINESS_QOS;
    public static final LivelinessQosPolicyKind
        MANUAL_BY_TOPIC_LIVELINESS_QOS;
}
public final class LivelinessQosPolicy
{ public LivelinessQosPolicyKind kind;
  public Duration_t lease_duration;}
```

#### Description

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

## Attributes

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

*Duration\_t lease\_duration* - the duration of the interval within which the liveliness must be determined.

## 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 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 9 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

```
import DDS.*;
public class OwnershipQosPolicyKind
{
    public static final OwnershipQosPolicyKind SHARED_OWNERSHIP_QOS;
    public static final OwnershipQosPolicyKind
        EXCLUSIVE_OWNERSHIP_QOS;
}
public final class OwnershipQosPolicy
{ public 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 10 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

```
import DDS.*;
public final class OwnershipStrengthQosPolicy
{   public int value; }
```

##### Description

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

##### Attributes

*int 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

```
import DDS.*;
public final class PartitionQosPolicy
{
```

```
    public String name[];
}
```

## Description

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

## Attributes

*String name[]* - the array 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 that 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

```
import DDS.*;
public class PresentationQosPolicyAccessScopeKind
{
    public static final PresentationQosPolicyAccessScopeKind
        INSTANCE_PRESENTATION_QOS;
    public static final PresentationQosPolicyAccessScopeKind
        TOPIC_PRESENTATION_QOS;
    public static final PresentationQosPolicyAccessScopeKind
        GROUP_PRESENTATION_QOS;
}
public final class PresentationQosPolicy
{
    public PresentationQosPolicyAccessScopeKind access_scope;
    public boolean coherent_access;
    public boolean ordered_access; }

```

## Description

This `QosPolicy` controls the extent to which 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.

## Attributes

*PresentationQosPolicyAccessScopeKind access\_scope* - specifies the granularity of the changes that needs to be preserved when communicating a set of samples. Currently only the `INSTANCE_PRESENTATION_QOS` and `TOPIC_PRESENTATION_QOS` scopes are supported.

*boolean coherent\_access* - controls whether the Data Distribution Service will preserve the groupings of changes, as indicated by the `access_scope`, made by a publishing application by means of the operations `begin_coherent_change` and `end_coherent_change`.

*boolean ordered\_access* - controls whether the Data Distribution Service will preserve the order of the changes, as indicated by the `access_scope`. Currently only an `ordered_access` setting of `false` is supported.

## Detailed Description

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 erroneously interpret that the aircraft is on a collision course.

Basically this `QosPolicy` allows a Publisher to group a number of samples by enclosing them within calls to `begin_coherent_change` and `end_coherent_change` and treat them as if they are to be communicated as a single message. 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 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.

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

### Attributes

The `PresentationQosPolicy` is applicable to both Publisher and Subscriber, but behaves differently on the publishing side and the subscribing side. The setting of `coherent_access` on a Publisher controls whether that Publisher will preserve the coherency of changes (enclosed by calls to `begin_coherent_change` and `end_coherent_change`), as indicated by its `access_scope` and as made available by its embedded `DataWriters`. However, the Subscriber settings determine whether a coherent set of samples will actually be delivered to the subscribing application in a coherent way.

- If a Publisher or Subscriber sets `coherent_access` to `false`, it indicates that it does not want to maintain coherency between the different samples in a set: a Subscriber that receives only a part of this set may still deliver this partial set of samples to its embedded `DataReaders`.
- If both Publisher and Subscriber set `coherent_access` to `true`, they indicate that they want to maintain coherency between the different samples in a set: a Subscriber that receives only a part of this set may not deliver this partial set of samples to its embedded `DataReaders`; it needs to wait for the set to become complete, and it will flush this partial set when it concludes that it will never be able to complete it.

Coherency is implemented on top of a transaction mechanism between individual `DataWriters` and `DataReaders`; completeness of a coherent set is determined by the successful completion of each of its participating transactions. The value of the `access_scope` attribute determines which combination of transactions constitute the contents of a coherent set.

The setting of `ordered_access` has no impact on the way in which a Publisher transmits its samples (although it does influence the RxO properties of this Publisher), but basically it determines whether a Subscriber will preserve the ordering of samples when the subscribing application uses its embedded `DataReaders` to read or take samples:

- If a Subscriber sets `ordered_access` to `false`, it indicates that it does not want to maintain ordering between the different samples it receives: a subscribing application that reads or takes samples will receive these samples ordered by their key-values, which does probably not resemble the order they were written in.
- If a Subscriber sets `ordered_access` to `true` (currently not supported), it indicates that it does want to maintain ordering within the specified `access_scope` between the different samples it receives: a subscribing application that reads or takes samples will receive these samples sorted by the order in which they were written..

The `access_scope` determines the maximum extent of coherent and/or ordered changes:

- If `access_scope` is set to `INSTANCE_PRESENTATION_QOS` and `coherent_access` is set to `true`, then the Subscriber will behave, with respect to maintaining coherency, in a way similar to an `access_scope` that is set to `TOPIC_PRESENTATION_QOS`. This is caused by the fact that coherency is defined as the successful completion of all participating transactions. If a `DataWriter` writes a transaction containing samples from different instances, and a connected `DataReader` misses one of these samples, then the transaction failed and the coherent set is considered incomplete by the receiving `DataReader`. It doesn't matter that all the other instances have received their samples successfully; an unsuccessful transaction by definition results in an incomplete coherent set. In that respect the DDS can offer no granularity that is more fine-grained with respect to coherency than that described by the `TOPIC_PRESENTATION_QOS`.

If `access_scope` is set to `INSTANCE_PRESENTATION_QOS` and `ordered_access` is set to `true`, then the subscriber will maintain ordering between samples belonging to the same instance. Samples belonging to different instances will still be grouped by their key-values instead of by the order in which they were received.

- If `access_scope` is set to `TOPIC_PRESENTATION_QOS` and `coherent_access` is set to `true`, then the DDS will define the scope of a coherent set on individual transactions. So a coherent set that spans samples coming from multiple `DataWriters` (indicated by its enclosure within calls to `begin_coherent_change` and `end_coherent_change` on their shared Publisher), is chopped up into separate and disjunct transactions (one for each participating `DataWriter`), where each transaction is processed separately. On the

subscribing side this may result in the successful completion of some of these transactions, and the unsuccessful completion of some others. In such cases all DataReaders that received successful transactions will deliver the embedded content to their applications, without waiting for the completion of other transactions in other DataReaders connected to the same Subscriber.

If `access_scope` is set to `TOPIC_PRESENTATION_QOS` and `ordered_access` is set to `true` (currently not supported), then the subscriber will maintain ordering between samples belonging to the same DataReader. This means that samples belonging to the same instance in the same DataReader may no longer be received consecutively if samples belonging to different instances were written in between.

- If `access_scope` is set to `GROUP_PRESENTATION_QOS` (currently not supported) and `coherent_access` is set to `true`, then the DDS will define the scope of a coherent set on the sum of all participating transactions. So a coherent set that spans samples coming from multiple DataWriters (indicated by its enclosure within calls to `begin_coherent_change` and `end_coherent_change` on their shared Publisher), is chopped up into separate and disjunct transactions (one for each participating DataWriter), where each transactions is processed separately. On the subscribing side this may result in the successful completion of some of these transactions, and the unsuccessful completion of some others. However, each DataReader is only allowed to deliver the embedded content when all participating transactions completed successfully. This means that DataReaders that received successful transactions will need to wait for all other DataReaders attached to the same Subscriber to also complete their transactions successfully. If one or more DataReaders conclude that they will not be able to complete their transactions successfully, then all DataReaders that participate in the original coherent set will flush the content of their transactions.

If `access_scope` is set to `GROUP_PRESENTATION_QOS` (currently not supported) and `ordered_access` is set to `true` (currently not supported), then ordering is maintained between samples that are written by DataWriters attached to a common Publisher and received by DataReaders attached to a common Subscriber. This way the subscribing application can access the changes as a unit and/or in the proper order. However, this does not necessarily imply that the subscribing application will indeed access the changes as a unit and/or in the correct order. For that to occur, the subscribing application must use the proper logic in accessing its datareaders:

- Upon notification by the callback operation `on_data_on_readers` of the SubscriberListener or when triggered by the similar `DATA_ON_READERS` status of the Subscriber's `StatusCondition`, the application uses `begin_access` on the Subscriber to indicate it will be accessing data through the Subscriber. This will lock the embedded datareaders for any incoming messages during the coherent data access.

- Then it calls `get_datareaders` on the Subscriber to get the list of `DataReader` objects where data samples are available. Note that when `ordered_access` is `TRUE`, then the list of `DataReaders` may contain the same reader several times. In this manner the correct sample order can be maintained among samples in different `DataReader` objects.
- Following this it calls `read` or `take` on each `DataReader` in the same order returned to access all the relevant changes in the `DataReader`. Note that when **`ordered_access`** is `true`, you should only read or take *one* sample at a time.
- Once it has called `read` or `take` on all the readers, it calls `end_access` on the Subscriber. This will unlock the embedded datareaders again.

### 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 PresentationQosPolicy**

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

The value offered is considered compatible with the value requested if and only if the following conditions are met:

1. The inequality “`offered access_scope >= requested access_scope`” evaluates to ‘`true`’. For the purposes of this inequality, the values of `PRESENTATION access_scope` are considered ordered such that `INSTANCE < TOPIC < GROUP`.
2. Requested `coherent_access` is `false`, or else both offered and requested `coherent_access` are `true`.
3. Requested `ordered_access` is `false`, or else both offered and requested `ordered_access` are `true`.

In case the quality offered by the Publisher is better than the value requested by the Subscriber, the subscriber’s values determine the resulting behaviour for the subscribing application. In other words, the quality specified at the Subscriber site overrules the corresponding value at the Publisher site.

Consider the following scenario:



1. A Publisher publishes coherent sets with `access_scope` is `GROUP` and `coherent_access` is `true`.
2. A Subscriber subscribes to these coherent sets with `access_scope` is `TOPIC` and `coherent_access` is `true`.
3. The Publisher writes a coherent set consisting of two samples of Topic A, and two samples of Topic B.
4. During transmission, the first sample of Topic B gets lost.

According to the `access_scope` of the Publisher, the coherent set is incomplete and can therefore not be delivered. However, according to the `access_scope` of the Subscriber, coherency needs to be maintained on a per Reader/Writer pair basis so the samples for Topic A will be delivered upon arrival, but the samples for Topic B will not.

Basically, when both `coherent_access` and `ordered_access` are set to `false`, then the `access_scope` serves no other purpose than to determine connectivity between Publishers and Subscribers.

An `access_scope` value of `GROUP_PRESENTATION_QOS` and/or an `ordered_access` value of `true` are not yet supported. Setting any of these values in your `PresentationQosPolicy` will result in a `RETCODE_NOT_SUPPORTED`.

### 3.1.3.15 ReaderDataLifecycleQosPolicy

#### Scope

DDS

#### Synopsis

```
import DDS.*;
public final class ReaderDataLifecycleQosPolicy
{
    public Duration_t autopurge_nowriter_samples_delay;
    public 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

```
import DDS.*;
public class ReliabilityQosPolicyKind
{
```

```

    public static final ReliabilityQosPolicyKind
        BEST_EFFORT_RELIABILITY_QOS;
    public static final ReliabilityQosPolicyKind
        RELIABLE_RELIABILITY_QOS;
}
public final class ReliabilityQosPolicy
{
    public ReliabilityQosPolicyKind kind;
    public Duration_t max_blocking_time;
    public 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

- *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 DDS\_HistoryQosPolicy and DDS\_ResourceLimitsQosPolicy and/or the synchronous setting of the DDS\_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 12 Requested/Offered ReliabilityQosPolicy**

<div>Requested</div> <div>Offered</div>	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

```
import DDS.*;
public final class ResourceLimitsQosPolicy
{
    public int max_samples;
    public int max_instances;
    public int max_samples_per_instance;
}
```

#### Description

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

#### Attributes

*int max\_samples* - the maximum number of data samples for all instances for any single DataWriter (or DataReader). By default, LENGTH\_UNLIMITED.

*int max\_instances* - the maximum number of instances for any single DataWriter (or DataReader). By default, LENGTH\_UNLIMITED.

*int 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

```
import DDS.*;
public class SchedulingClassQosPolicyKind
{
    public static final SchedulingClassQosPolicyKind
        SCHEDULE_DEFAULT;
    public static final SchedulingClassQosPolicyKind
        SCHEDULE_TIMESHARING;
    public static final SchedulingClassQosPolicyKind
        SCHEDULE_REALTIME;
}
public class SchedulingClassQosPolicy
{
    public SchedulingClassQosPolicyKind kind;
}
public class SchedulingPriorityQosPolicyKind
{
    public static final SchedulingPriorityQosPolicyKind
        PRIORITY_RELATIVE;
    public static final SchedulingPriorityQosPolicyKind
        PRIORITY_ABSOLUTE;
}
public class SchedulingPriorityQosPolicy
```

```

{
    public SchedulingPriorityQosPolicyKind kind;
}
public class SchedulingQosPolicy
{
    public SchedulingClassQosPolicy scheduling_class;
    public SchedulingPriorityQosPolicy scheduling_priority_kind;
    public int 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`.

*int 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

```
import DDS.*;
public final class TimeBasedFilterQosPolicy
{
    public 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

```
import DDS.*;
public final class TopicDataQosPolicy
{
    public byte value[];
}
```

#### Description

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

#### Attributes

*byte value[]* - a sequence of bytes 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

```
import DDS.*;
public final class TransportPriorityQosPolicy
{
    public int value;
}
```

#### Description

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

#### Attributes

*int 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

```
import DDS.*;
public final class UserDataQosPolicy
{
    public byte 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

*byte value[]* - a sequence of bytes 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

```
import DDS.*;
public final class WriterDataLifecycleQosPolicy
{
    public boolean autodispose_unregistered_instances;
}
```

## 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, the 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 286).

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 liveliness of a connected `DataWriter`, then the `autodispose_unregistered_instances` flag determines whether the concerned instances are automatically disposed (`true`) or not (`false`).

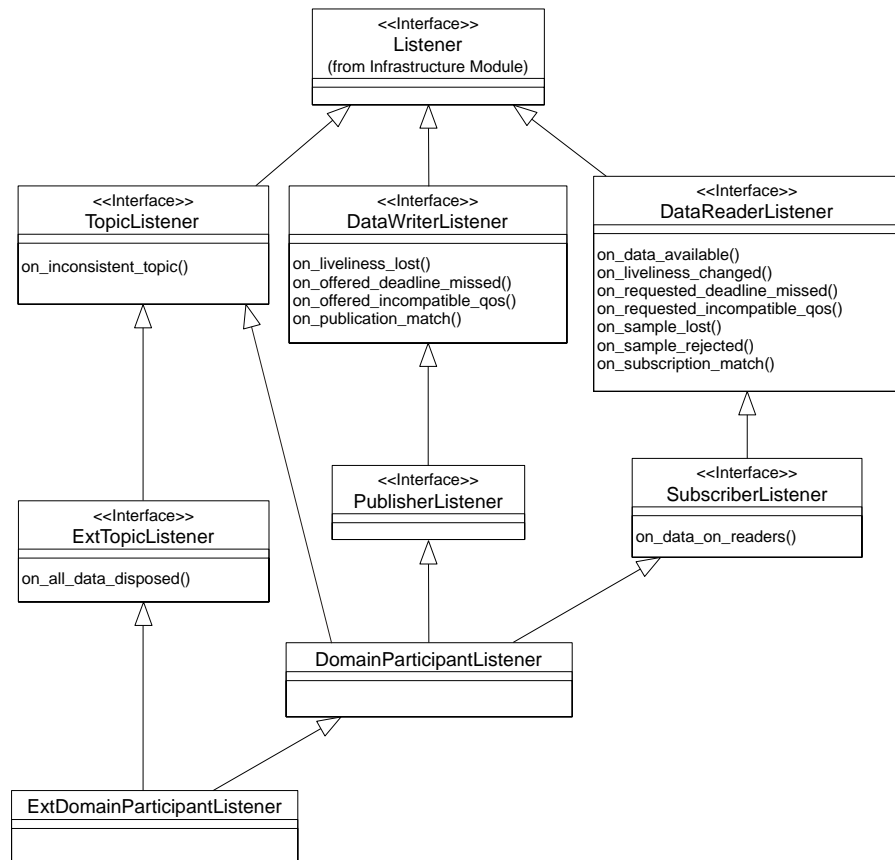
The default value for the `autodispose_unregistered_instances` flag is `true`. For `TRANSIENT` and `PERSISTENT` topics this means that all instances that are not explicitly unregistered by the application will by default be removed from the Transient and Persistent stores when the `DataWriter` is deleted or when a loss of its liveliness 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
- ExtDomainParticipantListener
- TopicListener
- ExtTopicListener
- 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 Class 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 13 Status Description Per Entity**

Entity	Status Name	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.
DataReader	SAMPLE_REJECTED_STATUS	A (received) sample has been rejected.
	LIVELINESS_CHANGED_STATUS	The liveliness of one or more <code>DataWriter</code> objects that were writing instances read through the <code>DataReader</code> has changed. Some <code>DataWriter</code> have become “alive” or “not alive”.
	REQUESTED_DEADLINE_MISSED_STATUS	The deadline that the <code>DataReader</code> was expecting through its <code>DeadlineQosPolicy</code> was not respected for a specific instance.
	REQUESTED_INCOMPATIBLE_QOS_STATUS	A <code>QosPolicy</code> 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 <code>DataReader</code> has found a <code>DataWriter</code> that matches the <code>Topic</code> and has compatible QoS.

**Table 13 Status Description Per Entity (Continued)**

Entity	Status Name	Meaning
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.

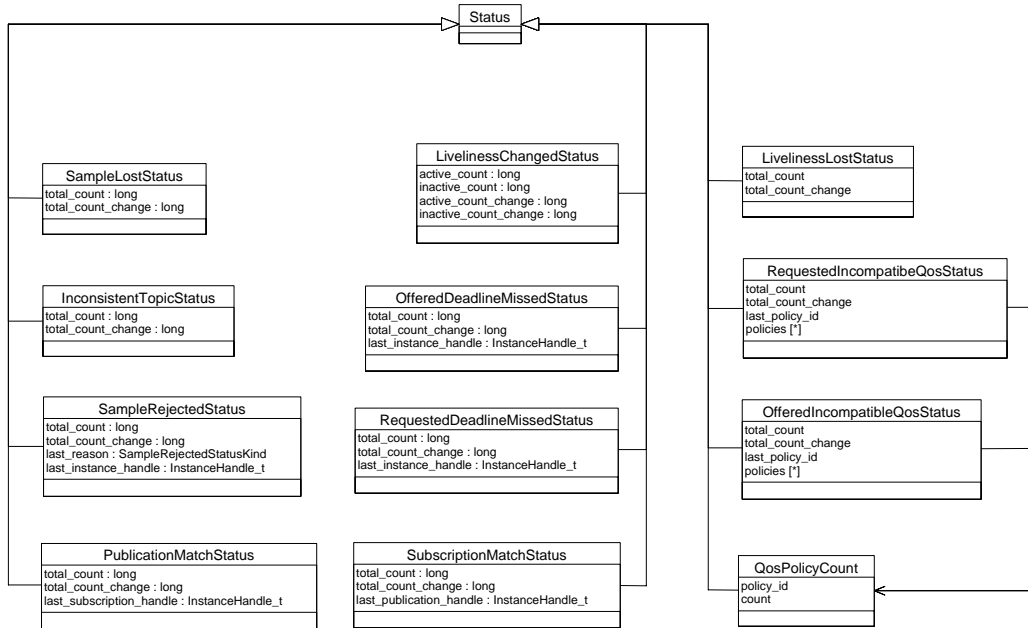


Figure 12 DCPS Status Values

Each Status attribute is implemented as a class without any operations. The interface description of these class is as follows:

```

// public class <name>Status
//
public final class InconsistentTopicStatus
{
    public int total_count;
    public int total_count_change;
};
public final class SampleLostStatus
{
    public int total_count;
    public int total_count_change;
};
public final class SampleRejectedStatusKind
{
    public static final SampleRejectedStatusKind
        NOT_REJECTED;
    public static final SampleRejectedStatusKind
        REJECTED_BY_INSTANCES_LIMIT;
    public static final SampleRejectedStatusKind
        REJECTED_BY_SAMPLES_LIMIT;
    public static final SampleRejectedStatusKind
        REJECTED_BY_SAMPLES_PER_INSTANCE_LIMIT;
};
  
```

```

};
public final class SampleRejectedStatus
{
    public int total_count;
    public int total_count_change;
    public SampleRejectedStatusKind last_reason;
    public long last_instance_handle;
};
public final class LivelinessLostStatus
{
    public int total_count;
    public int total_count_change;
};
public final class LivelinessChangedStatus
{
    public int alive_count;
    public int not_alive_count;
    public int alive_count_change;
    public int not_alive_count_change;
    public long last_publication_handle;};
public final class OfferedDeadlineMissedStatus
{
    public int total_count;
    public int total_count_change;
    public long last_instance_handle;
};
public final class RequestedDeadlineMissedStatus
{
    public int total_count;
    public int total_count_change;
    public long last_instance_handle;
};
public final class OfferedIncompatibleQosStatus
{
    public int total_count;
    public int total_count_change;
    public int last_policy_id;
    public QosPolicyCount policies[];
};
public final class RequestedIncompatibleQosStatus
{
    public int total_count;
    public int total_count_change;
    public int last_policy_id;
    public QosPolicyCount policies[];
};
public final class PublicationMatchedException
{
    public int total_count;

```



```

        public int total_count_change;
        public int current_count;
        public int current_count_change;
        public long last_subscription_handle;
    };
    public final class SubscriptionMatchedStatus
    {
        public int total_count;
        public int total_count_change;
        public int current_count;
        public int current_count_change;
        public long last_publication_handle;
    };
    //
    // implemented API operations
    // <no operations>
    //

```

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

### 3.1.5.1 InconsistentTopicStatus

#### Scope

DDS.Topic

#### Synopsis

```

import DDS.*;
public final class InconsistentTopicStatus
{
    public int total_count;
    public int total_count_change;
};

```

#### Description

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

#### Attributes

*int 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.

*int total\_count\_change* - the change in total\_count since the last time the Listener was called or the Status was read.

## Detailed Description

This class contains the statistics about attempts to create other `Topic`s 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.Topic`

#### Synopsis

```
import DDS.*;
public final class LivelinessChangedStatus
{
    public int alive_count;
    public int not_alive_count;
    public int alive_count_change;
    public int not_alive_count_change;
    public long last_publication_handle;
};
```

#### Description

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

#### Attributes

*int alive\_count* - the total count of currently alive `DataWriter` objects that write the topic read by the `DataReader` to which this `Status` is attached.

*int 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.

*int alive\_count\_change* - the change in `alive_count` since the last time the `Listener` was called or the `Status` was read.

*int not\_alive\_count\_change* - the change in `not_alive_count` since the last time the `Listener` was called or the `Status` was read.

*long last\_publication\_handle* - handle to the last `DataWriter` whose change in liveliness caused this status to change.

## Detailed Description

This class 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.Topic`

## Synopsis

```
import DDS.*;
public final class LivelinessLostStatus
{
    public int total_count;
    public int total_count_change;
};
```

## Description

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

## Attributes

*int 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.

*int total\_count\_change* - the change in `total_count` since the last time the `Listener` was called or the `Status` was read.

## Detailed Description

This class 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.Topic
```

#### Synopsis

```
import DDS.*;
```

```
public final class OfferedDeadlineMissedStatus
{
    public int total_count;
    public int total_count_change;
    public long last_instance_handle;
};
```

## Description

This class 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

*int total\_count* - the total cumulative count of times the `DataWriter` to which this `Status` is attached failed to write within its offered deadline.

*int total\_count\_change* - the change in `total_count` since the last time the `Listener` was called or the `Status` was read.

*long 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 class 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.Topic`

## Synopsis

```
import DDS.*;
public final class OfferedIncompatibleQosStatus
{
    public int total_count;
    public int total_count_change;
    public int last_policy_id;
    public QosPolicyCount policies[];
};
```

## Description

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

## Attributes

*int 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.

*int total\_count\_change* - the change in total\_count since the last time the Listener was called or the Status was read.

*int 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.

*QosPolicyCount 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 class 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* for an overview of all these constants.

**Table 14 Overview of All Named QosPolicy Indexes**

Index name	Index Value
<code>INVALID_QOS_POLICY_ID</code>	0
<code>USERDATA_QOS_POLICY_ID</code>	1
<code>DURABILITY_QOS_POLICY_ID</code>	2
<code>PRESENTATION_QOS_POLICY_ID</code>	3
<code>DEADLINE_QOS_POLICY_ID</code>	4
<code>LATENCYBUDGET_QOS_POLICY_ID</code>	5
<code>OWNERSHIP_QOS_POLICY_ID</code>	6
<code>OWNERSHIPSTRENGTH_QOS_POLICY_ID</code>	7
<code>LIVELINESS_QOS_POLICY_ID</code>	8
<code>TIMEBASEDFILTER_QOS_POLICY_ID</code>	9

**Table 14 Overview of All Named QosPolicy Indexes (Continued)**

Index name	Index Value
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.Topic

#### Synopsis

```
import DDS.*;
public final class PublicationMatchedStatus
{
    public int total_count;
    public int total_count_change;
    public int current_count;
    public int current_count_change;
    public long last_subscription_handle;
};
```

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

### 3.1.5.7 RequestedDeadlineMissedStatus

#### Scope

DDS.Topic



## Synopsis

```
import DDS.*;
public final class RequestedDeadlineMissedStatus
{
    public int total_count;
    public int total_count_change;
    public long last_instance_handle;
};
```

## Description

This class 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

*int total\_count* - the total cumulative count of the missed deadlines detected for any instance read by the `DataReader` to which this `Status` is attached.

*int total\_count\_change* - the change in *total\_count* since the last time the `Listener` was called or the `Status` was read.

*long 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 class 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.Topic`

## Synopsis

```
import DDS.*;
public final class RequestedIncompatibleQosStatus
{
    public int total_count;
    public int total_count_change;
    public int last_policy_id;
    public QosPolicyCount policies[];
};
```

## Description

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

## Attributes

*int 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.

*int total\_count\_change* - the change in total\_count since the last time the Listener was called or the Status was read.

*int 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.

*QosPolicyCount 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 class 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 14, *Overview of All Named QosPolicy Indexes*, on page 95 for an overview of all these constants.

### 3.1.5.9 SampleLostStatus

#### Scope

`DDS.Topic`

#### Synopsis

```
import DDS.*;
public final class SampleLostStatus
{
    public int total_count;
    public int total_count_change;
};
```

## Description

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

## Attributes

*int total\_count* - the total cumulative count of all samples lost across all instances of data published under the `Topic`.

*int total\_count\_change* - the change in `total_count` since the last time the `Listener` was called or the `Status` was read.

## Detailed Description

This class 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.Topic`

## Synopsis

```
import DDS.*;
public final class SampleRejectedStatusKind
{
    public static final SampleRejectedStatusKind
        NOT_REJECTED;
    public static final SampleRejectedStatusKind
        REJECTED_BY_INSTANCES_LIMIT;
    public static final SampleRejectedStatusKind
        REJECTED_BY_SAMPLES_LIMIT;
    public static final SampleRejectedStatusKind
        REJECTED_BY_SAMPLES_PER_INSTANCE_LIMIT;
}
public final class SampleRejectedStatus
{
    public int total_count;
    public int total_count_change;
    public SampleRejectedStatusKind last_reason;
    public long last_instance_handle;
};
```

## Description

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

## Attributes

*int total\_count* - the total cumulative count of samples rejected by the `DataReader` to which this `DDS_Status` is attached.

*int total\_count\_change* - the change in `total_count` since the last time the `Listener` was called or the `DDS_Status` was read.

*SampleRejectedStatusKind last\_reason* - the reason for rejecting the last sample.

*long last\_instance\_handle* - the handle to the instance which would have been updated by the last sample that was rejected.

## Detailed Description

This class 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 `DDS_Status` is attached.

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

## Synopsis

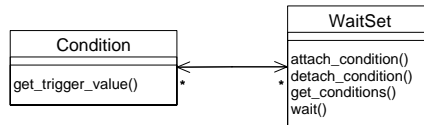
```
import DDS.*;
public final class SubscriptionMatchedExceptionStatus
{
    public int total_count;
    public int total_count_change;
    public int current_count;
    public int current_count_change;
    public long last_publication_handle;
};
```

**NOTE:** The functionality behind the SubscriptionMatchedExceptionStatus 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:

```
public class WaitSet
{
    //
    // implemented API operations
    //
    public int
        _wait
        (ConditionSeqHolder active_conditions,
         Duration_t timeout);
    public int
        attach_condition
        (Condition cond);
    public int
        detach_condition
        (Condition cond);
    public int
        get_conditions
        (ConditionSeqHolder attached_conditions);
};
```

The following paragraphs describe the usage of all WaitSet operations.

### 3.1.6.1 attach\_condition

#### Scope

DDS.WaitSet

#### Synopsis

```
import DDS.*;
public int
    attach_condition
        (Condition cond);
```

#### Description

This operation attaches a Condition to the WaitSet.

#### Parameters

*in Condition cond* - a reference to a Condition.

#### Return Value

*int* - 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 Java 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 reference.
- `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

```
import DDS.*;
public int
    detach_condition
        (Condition cond);
```

#### Description

This operation detaches a Condition from the WaitSet.

#### Parameters

*in Condition cond* - a reference to a Condition in the WaitSet.

#### Return Value

*int* - 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 reference.
- `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

```
import DDS.*;
public int
    get_conditions
        (ConditionSeqHolder attached_conditions);
```

#### Description

This operation retrieves the list of attached conditions.

#### Parameters

*inout ConditionSeqHolder attached\_conditions* - a Holder to a sequence which is used to pass the list of attached conditions.

#### Return Value

*int* - 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 Holder to a sequence which afterwards will refer to the sequence of attached conditions. The array inside the `attached_conditions` Holder may be pre-allocated by the application and can be re-used in a subsequent invocation of the `get_conditions` operation. If the pre-allocated array is not big enough to hold the the number of attached 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 Java 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

```
import DDS.*;
public int
    _wait
        (ConditionSeqHolder active_conditions,
         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 ConditionSeqHolder active\_conditions* - parameter `active_conditions` is a `Holder` to a sequence, which is used to pass the list of all the attached conditions that have a `trigger_value` of `true`.

*in 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

*int* - 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 `Holder` for a sequence, which will contain the list of all the attached conditions that have a `trigger_value` of `true`. The array inside the `active_conditions` `Holder` may be pre-allocated by the application and can be re-used in a subsequent invocation of the `_wait` operation. If the pre-allocated array is not big enough to hold the the number of triggered Conditions, the array 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 have 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

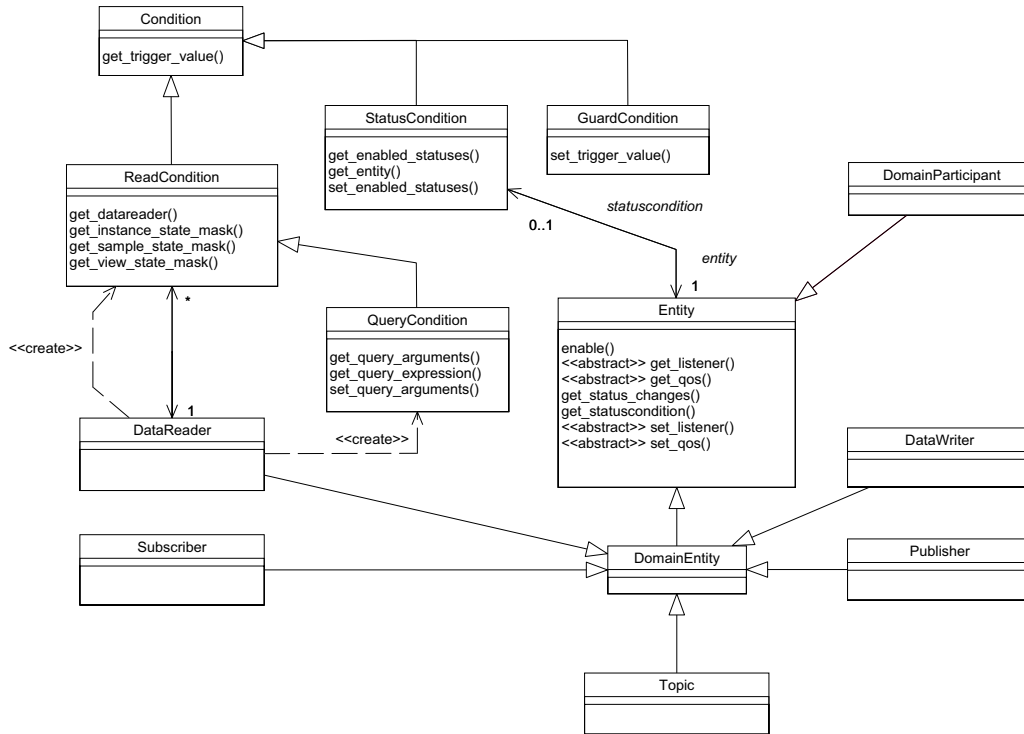
When the operation returns:

- `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 is as follows:

```

public interface Condition
{
    //
    // implemented API operations
    //
    public 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

```
import DDS.*;
public 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:

```
public interface GuardCondition
{
    //
    // extends interface Condition
    //
    // public boolean
    //     get_trigger_value
    //         (void);
    //
    // implemented API operations
    //
    public int
```

```

        set_trigger_value
            (boolean value);
    };

```

The following 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

```

import DDS.*;
public boolean
    get_trigger_value
        (void);

```

### 3.1.8.2 `set_trigger_value`

#### Scope

`DDS.GuardCondition`

#### Synopsis

```

import DDS.*;
public int
    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

*int* - 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 15 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_MATCH_STATUS
DataWriter	LIVELINESS_LOST_STATUS
	OFFERED_DEADLINE_MISSED_STATUS
	OFFERED_INCOMPATIBLE_QOS_STATUS
	PUBLICATION_MATCH_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:

```
public interface StatusCondition
{
    //
    // extends interface Condition
    //
    // boolean
    //     get_trigger_value
    //         (void);
    //
    // implemented API operations
    //
    public int
        get_enabled_statuses
            (void);

    public int
        set_enabled_statuses
            (int mask);

    public Entity
        get_entity
            (void);
};
```

The following 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

```
import DDS.*;
public int
```



```
get_enabled_statuses  
(void);
```

## Description

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

## Parameters

<none>

## Return Value

*int* - Result is 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 `int`.

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

```
import DDS.*;
public Entity
    get_entity
        (void);
```

#### Description

This operation returns the `Entity` associated with the `StatusCondition` or the null reference.

#### Parameters

<none>

#### Return Value

*Entity* - Result value is a reference to the `Entity` associated with the `StatusCondition` or the null reference.

#### 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 reference 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

```
import DDS.*;
public boolean
    get_trigger_value
        (void);
```

### 3.1.9.4 set\_enabled\_statuses

#### Scope

DDS.StatusCondition

#### Synopsis

```
import DDS.*;
public int
    set_enabled_statuses
        (int 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 int mask* - a bit mask in which each bit sets the status which is taken into account for the `StatusCondition`.

#### Return Value

*int* - 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 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 OR operation to set the status bit in the parameter `mask` of type `int`. The constants are implemented as an interface.

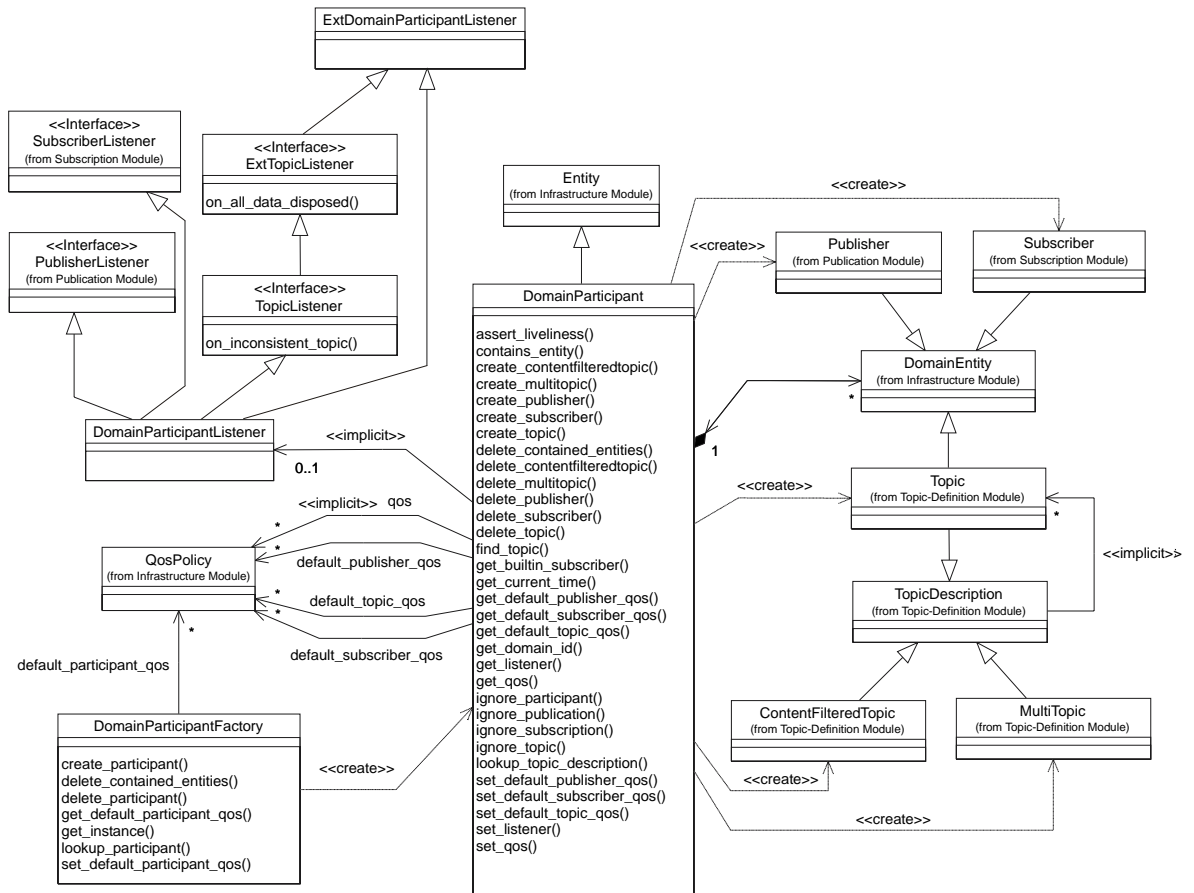
Not all statuses are relevant to all `Entity` objects. See the respective `Listener` objects for each `Entity` for more information.

#### 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.

### 3.2 Domain Module



### Figure 15 DCPS Domain Module's Class Model

This module contains the following classes:

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

### 3.2.1 Class DomainParticipant

All the DCPS Entity objects are attached to a DomainParticipant.

A `DomainParticipant` represents the local membership of the application in a `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 Publishers and the Subscribers attached to the same Domain can interact.

This class implements several functions:

- It acts as a container for all other Entity objects
- 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:

```
public interface DomainParticipant
{
//
// extends interface Entity
//
// public StatusCondition
//   get_statuscondition
//   (void);
// public int
//   get_status_changes
//   (void);
// public int
//   enable
//   (void);
//
// implemented API operations
//
    public Publisher
        create_publisher
        (PublisherQos qos,
         PublisherListener a_listener,
         int mask);
    public int
        delete_publisher
        (Publisher p);
    public Subscriber
```

```

        create_subscriber
            (SubscriberQos qos,
             SubscriberListener a_listener,
             int mask);
public int
    delete_subscriber
        (Subscriber s);
public Subscriber
    get_builtin_subscriber
        (void);
public Topic
    create_topic
        (String topic_name,
         String type_name,
         TopicQos qos,
         TopicListener a_listener,
         int mask);
public int
    delete_topic
        (Topic a_topic);
public Topic
    find_topic
        (String topic_name,
         Duration_t timeout);
public TopicDescription
    lookup_topicdescription
        (String name);
public ContentFilteredTopic
    create_contentfilteredtopic
        (String name,
         Topic related_topic,
         String filter_expression,
         String[] expression_parameters);
public int
    delete_contentfilteredtopic
        (ContentFilteredTopic a_contentfilteredtopic);
public MultiTopic
    create_multitopic
        (String name,
         String type_name,
         String subscription_expression,
         String[] expression_parameters);
public int
    delete_multitopic
        (MultiTopic a_multitopic);
public int
    delete_contained_entities
        (void);
public int
    set_qos

```

```

        (DomainParticipantQos qos);
    public int
        get_qos
        (DomainParticipantQosHolder qos);
    public int
        set_listener
        (DomainParticipantListener a_listener,
         int mask);
    public DomainParticipantListener
        get_listener
        (void);
    public int
        ignore_participant
        (long handle);
    public int
        ignore_topic
        (long handle);
    public int
        ignore_publication
        (long handle);
    public int
        ignore_subscription
        (long handle);
    public String
        get_domain_id
        (void);
    public int
        get_discovered_participants
        (InstanceHandleSeqHolder participant_handles);
    public int
        get_discovered_participant_data
        (long ParticipantBuiltinTopicDataHolder participant_data,
         handle);
    public int
        get_discovered_topics
        (InstanceHandleSeqHolder topic_handles);
    public int
        get_discovered_topic_data
        (long TopicBuiltinTopicDataHolder topic_data,
         handle);
    public int
        assert_liveliness
        (void);
    public int
        set_default_publisher_qos
        (PublisherQos qos);
    public int
        get_default_publisher_qos
        (PublisherQosHolder qos);
    public int

```



```

        set_default_subscriber_qos
            (SubscriberQos qos);
    public int
        get_default_subscriber_qos
            (SubscriberQosHolder qos);
    public int
        set_default_topic_qos
            (TopicQos qos);
    public int
        get_default_topic_qos
            (TopicQosHolder qos);
    public boolean
        contains_entity
            (long a_handle);
    public int
        get_current_time
            (Time_tHolder current_time);
};

```

The following 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

```

import DDS.*;
public int
    assert_liveliness
        (void);

```

#### Description

This operation asserts the liveliness for the `DomainParticipant`.

#### Parameters

<none>

#### Return Value

*int* - 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 `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 liveness of those `DataWriters`.

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

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

### Return Code

When the operation returns:

- `RETCODE_OK` - the liveness 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

```
import DDS.*;
public boolean
    contains_entity
        (long 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 long 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

```
import DDS.*;
public ContentFilteredTopic
    create_contentfilteredtopic
        (String name,
         Topic related_topic,
         String filter_expression,
         String[] 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 String name* - the name of the ContentFilteredTopic.

*in Topic related\_topic* - the reference to the base topic on which the filtering will be applied. Therefore, a filtered topic is based on an existing Topic.

*in String filter\_expression* - the SQL expression (subset of SQL), which defines the filtering.

*in String[] 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* - Return value is the reference to the newly created *ContentFilteredTopic*. In case of an error, a null reference 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

```
import DDS.*;
public MultiTopic
```

```
create_multitopic  
(String name,  
    String type_name,  
    String subscription_expression,  
    String[] 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 String name* - the name of the multi topic.

*in String 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 String subscription\_expression* - the SQL expression (subset of SQL), which defines the selection, filtering, combining and re-arranging of the sample data.

*in String[] 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* - Return value is the reference to the newly created `MultiTopic`. In case of an error, a null reference is returned.

## Detailed Description

This operation creates a multiple topic 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

```
import DDS.*;
public Publisher
    create_publisher
        (PublisherQos qos,
         PublisherListener a_listener,
         int mask);
```

#### Description

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

## Parameters

*in 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 a\_listener* - a reference 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 int mask* - a bit-mask in which each bit enables the invocation of the PublisherListener for a certain status.

## Return Value

*Publisher* - Return value is a reference to the newly created Publisher. In case of an error, the null reference 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 reference 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 reference is returned. The null reference 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

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.

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)*.



Be aware that the `PUBLICATION_MATCHED_STATUS` is not applicable when the infrastructure does not have the information available to determine connectivity. This is the case when OpenSplice is configured not to maintain discovery information in the Networking Service. (See the description for the `NetworkingService/Discovery/enabled` property in the Deployment Manual for more information about this subject.) In this case the operation will return `null`.

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 applicable 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

```
import DDS.*;
public Subscriber
    create_subscriber
        (SubscriberQos qos,
         SubscriberListener a_listener,
         int mask);
```

#### Description

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

#### Parameters

*in 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 a\_listener* - a reference 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 int mask* - a bit-mask in which each bit enables the invocation of the SubscriberListener for a certain status.

#### Return Value

*Subscriber* - Return value is a reference to the newly created Subscriber. In case of an error, the null reference 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 reference 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 reference is returned. The null reference 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 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.

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`.



Be aware that the `SUBSCRIPTION_MATCHED_STATUS` is not applicable when the infrastructure does not have the information available to determine connectivity. This is the case when `OpenSplice` is configured not to maintain discovery information in the `Networking Service`. (See the description for the `NetworkingService/Discovery/enabled` property in the `Deployment Manual` for more information about this subject.) In this case the operation will return `null`.

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 applicable 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

```
import DDS.*
public Topic
    create_topic
        (String topic_name,
         String type_name,
         TopicQos qos,
         TopicListener a_listener,
         int 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 String 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 String type\_name* - a local alias of the data type, which must have been registered before creating the `Topic`.

*in 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 a\_listener* - a reference 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 int mask* - a bit-mask in which each bit enables the invocation of the `TopicListener` for a certain status.

## Return Value

*Topic* - Return value is a reference to the new or existing `Topic`. In case of an error, the `null` reference 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` reference must be supplied instead. In case the specified `QosPolicy` settings are not consistent, no `Topic` is created and the `null` reference 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 reference 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 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.

The following statuses are applicable to the `TopicListener`:

- `INCONSISTENT_TOPIC_STATUS`

The following statuses are applicable to the `ExtTopicListener`:

- `ON_ALL_DATA_DISPOSED_TOPIC_STATUS`

**NOTE:** The `DDS.STATUS_MASK_ANY_V1_2` mask does not include the `ON_ALL_DATA_DISPOSED_TOPIC_STATUS` bit, because this is an OpenSplice extension.

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

```
import DDS.*;
public int
    delete_contained_entities
        (void);
```

### Description

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

### Parameters

<none>

## Return Value

*int* - 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

```
import DDS.*;
public int
    delete_contentfilteredtopic
        (ContentFilteredTopic a_contentfilteredtopic);
```

#### Description

This operation deletes a ContentFilteredTopic.

#### Parameters

*in ContentFilteredTopic a\_contentfilteredtopic* - a reference to the ContentFilteredTopic, which is to be deleted.

#### Return Value

*int* - 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 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* 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\_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

```
import DDS.*;
public int
    delete_multitopic
        (MultiTopic 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 a\_multitopic* - a reference to the *MultiTopic*, which is to be deleted.

#### Return Value

*int* - 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 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 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_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

```
import DDS.*;
public int
    delete_publisher
        (Publisher p);
```

#### Description

This operation deletes a Publisher.

#### Parameters

*in Publisher p* - a reference to the Publisher, which is to be deleted.

## Return Value

*int* - 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 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_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

```
import DDS.*
public int
    delete_subscriber
        (Subscriber s);
```

#### Description

This operation deletes a Subscriber.

## Parameters

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

## Return Value

*int* - 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 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_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

```
import DDS.*;
public int
    delete_topic
        (Topic a_topic);
```

## Description

This operation deletes a `Topic`.

## Parameters

*in Topic a\_topic* - a reference to the `Topic`, which is to be deleted.

## Return Value

*int* - 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` 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_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

```
import DDS.*;
public int
    enable
        (void);
```

#### 3.2.1.15 `find_topic`

##### Scope

`DDS.DomainParticipant`

##### Synopsis

```
import DDS.*;
public Topic
    find_topic
        (String topic_name,
         Duration_t timeout);
```

##### Description

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

##### Parameters

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

*in* `Duration_t timeout` - the maximum duration to block for the `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* - a reference 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 reference 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

```
import DDS.*;
public Subscriber
    get_builtin_subscriber
        (void);
```

**Description**

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

**Parameters**

<none>

**Return Value**

*Subscriber* - Result value is a reference 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**

```
import DDS.*;
public int
    get_current_time
        (Time_tHolder 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\_tHolder 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**

*int* - 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

```
import DDS.*;
public int
    get_default_publisher_qos
        (PublisherQosHolder qos);
```

#### Description

This operation gets an object with the default `Publisher QosPolicy` settings of the `DomainParticipant`.

#### Parameters

*inout* `PublisherQosHolder qos` - a reference to the destination `PublisherQosHolder` object in which the default `QosPolicy` settings for the `Publisher` are written.

#### Return Value

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

## Detailed Description

This operation gets an object of the class `PublisherQos` 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` object 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 object referenced to by `qos`. Any settings in the object 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, *DCPS Holder Classes*, on page 12.

### Return Code

When the operation returns:

- `RETCODE_OK` - the default `Publisher QosPolicy` settings of this `DomainParticipant` have successfully been copied into the specified `PublisherQosHolder` 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

```
import DDS.*;
public int
    get_default_subscriber_qos
        (SubscriberQosHolder qos);
```

#### Description

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

## Parameters

*inout SubscriberQosHolder qos* - a reference to the destination SubscriberQosHolder object in which the default QosPolicy settings for the Subscriber are written.

## Return Value

*int* - Possible return codes of the operation are: RETCODE\_OK, RETCODE\_ERROR, RETCODE\_ALREADY\_DELETED or RETCODE\_OUT\_OF\_RESOURCES.

## Detailed Description

This operation gets an object of the class SubscriberQos with the default Subscriber QosPolicy 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 object in which the policy can be stored and pass the qos reference to the operation. The operation writes the default QosPolicy to the object referenced to by qos. Any settings in the object 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 QosPolicy as defined in Table 2, *DCPS Holder Classes*, on page 12.

### Return Code

When the operation returns:

- *RETCODE\_OK* - the default Subscriber QosPolicy settings of this DomainParticipant have successfully been copied into the specified SubscriberQosHolder 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

```
import DDS.*;
public int
    get_default_topic_qos
        (TopicQosHolder qos);
```

## Description

This operation gets an object with the default `Topic QosPolicy` settings of the `DomainParticipant`.

## Parameters

*inout TopicQosHolder qos* - a reference to the destination `TopicQosHolder` object in which the default `QosPolicy` settings for the `Topic` are written.

## Return Value

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

## Detailed Description

This operation gets an object of the class `TopicQos` 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 an object of the `TopicQos` class in which the policy can be stored and pass the `qos` reference to the operation. The operation writes the default `QosPolicy` settings to the object referenced to by `qos`. Any settings in the object 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` as defined in Table 2, *DCPS Holder Classes*, on page 12.

### Return Code

When the operation returns:

- `RETCODE_OK` - the default `Topic QosPolicy` settings of this `DomainParticipant` have successfully been copied into the specified `TopicQosHolder` 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

```
import DDS.*;
public int
    get_discovered_participants
        (InstanceHandleSeqHolder 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

```
import DDS.*;
public int
    get_discovered_participant_data
        (long ParticipantBuiltinTopicDataHolder participant_data,
         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

```
import DDS.*;
public int
    get_discovered_topics
        (InstanceHandleSeqHolder 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

```
import DDS.*;
public int
    get_discovered_topic_data
        (long TopicBuiltinTopicDataHolder topic_data,
         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

```
import DDS.*;
public String
    get_domain_id
        (void);
```

## Description

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

## Parameters

<none>

## Return Value

*String* - 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 reference: 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

```
import DDS.*;
public DomainParticipantListener
    get_listener
        (void);
```

#### Description

This operation allows access to a `DomainParticipantListener`.

#### Parameters

<none>

#### Return Value

*DomainParticipantListener* - result is a reference 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 reference is returned.

### 3.2.1.27 `get_qos`

#### Scope

`DDS.DomainParticipant`

#### Synopsis

```
import DDS.*;
public int
    get_qos
        (DomainParticipantQosHolder qos);
```

#### Description

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

## Parameters

*inout DomainParticipantQosHolder qos* - a reference to the destination *DomainParticipantQosHolder* object in which the *QosPolicy* settings will be copied.

## Return Value

*int* - Possible return codes of the operation are: *RETCODE\_OK*, *RETCODE\_ERROR*, *RETCODE\_ALREADY\_DELETED* or *RETCODE\_OUT\_OF\_RESOURCES*.

## 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 *DomainParticipantQosHolder* 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.

## 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 referenced by the *qos* parameter.

### 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

```
import DDS.*;
public int
    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**

```
import DDS.*;
public StatusCondition
    get_statuscondition
        (void);
```

**3.2.1.30 ignore\_participant****Scope**

DDS.DomainParticipant

**Synopsis**

```
import DDS.*
public int
    ignore_participant
        (long 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**

```
import DDS.*;
public int
    ignore_publication
        (long 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**

```
import DDS.*;
public int
    ignore_subscription
        (long 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

```
import DDS.*;
public int
    ignore_topic
        (long 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

```
import DDS.*;
public TopicDescription
    lookup_topicdescription
        (String name);
```

#### Description

This operation gives access to a locally-created `TopicDescription`, with a matching name.

#### Parameters

*in String name* - the name of the `TopicDescription` to look for.

#### Return Value

*TopicDescription* - Return value is a reference to the `TopicDescription` found. When no such `TopicDescription` is found, the null reference 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, *find\_topic*, on page 142) of the same name already exist, a reference 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 reference. 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

```
import DDS.*;
public int
    set_default_publisher_qos
        (PublisherQos qos);
```

#### Description

This operation sets the default `PublisherQos` of the `DomainParticipant`.

#### Parameters

*in* `PublisherQos qos` - a collection of `QosPolicy` settings, which contains the new default `QosPolicy` settings for the newly created Publishers.

#### Return Value

*int* - 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 compound class 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 that 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.
- `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

```
import DDS.*;
public int
    set_default_subscriber_qos
        (SubscriberQos qos);
```

#### Description

This operation sets the default SubscriberQos of the DomainParticipant.

#### Parameters

*in* `SubscriberQos qos` - a collection of QosPolicy settings, which contains the new default QosPolicy settings for the newly created Subscribers.

#### Return Value

*int* - 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 compound class 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 that 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`.
- `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

```
import DDS.*;
public int
    set_default_topic_qos
        (TopicQos qos);
```

#### Description

This operation sets the default `TopicQos` of the `DomainParticipant`.

#### Parameters

*in* `TopicQos qos` - a collection of `QosPolicy` settings, which contains the new default `QosPolicy` settings for the newly created `Topics`.

#### Return Value

*int* - 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 compound class 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.
- `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

```
import DDS.*;
public int
    set_listener
        (DomainParticipantListener a_listener,
         int mask);
```

## Description

This operation attaches a `DomainParticipantListener` to the `DomainParticipant`.

## Parameters

*in* `DomainParticipantListener a_listener` - a reference to the `DomainParticipantListener` instance, which will be attached to the `DomainParticipant`.

*in* `int mask` - a bit mask in which each bit enables the invocation of the `DomainParticipantListener` for a certain status.

## Return Value

*int* - Possible return codes of the operation are: `RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_UNSUPPORTED`, `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 reference, it represents a listener that is treated as a NOOP<sup>1</sup> for all statuses activated in the bitmask.

### 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 performs nothing at all.

- 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).*



Be aware that the `PUBLICATION_MATCHED_STATUS` and `SUBSCRIPTION_MATCHED_STATUS` are not applicable when the infrastructure does not have the information available to determine connectivity. This is the case when OpenSplice is configured not to maintain discovery information in the Networking Service. (See the description for the `NetworkingService/Discovery/enabled` property in the Deployment Manual for more information about this subject.) In this case the operation will return `RETCODE_UNSUPPORTED`.

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 applicable 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_UNSUPPORTED` - a status was selected that cannot be supported because the infrastructure does not maintain the required connectivity information.
- `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

```
import DDS.*;
public int
    set_qos
        (DomainParticipantQos qos);
```

#### Description

This operation replaces the existing set of `QosPolicy` settings for a `DomainParticipant`.

#### Parameters

*in* `DomainParticipantQos qos` - the new set of `QosPolicy` settings for the `DomainParticipant`.

#### Return Value

*int* - 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 object with the `QosPolicy` settings which is checked for self-consistency.

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

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:

```
public class DomainParticipantFactory
{
    //
    // implemented API operations
    //
    public static DomainParticipantFactory
        get_instance
        (void);
    public DomainParticipant
        create_participant
        (String domainId,
         DomainParticipantQos qos,
         DomainParticipantListener a_listener,
         int mask);
    public int
        delete_participant
        (DomainParticipant a_participant);
```

```

    public DomainParticipant
        lookup_participant
            (String domainId);
    public int
        set_default_participant_qos
            (DomainParticipantQos qos);
    public int
        get_default_participant_qos
            (DomainParticipantQosHolder qos);
    public int
        set_qos
            (DomainParticipantFactoryQos qos);
    public int
        get_qos
            (DomainParticipantFactoryQosHolder qos);
    public int
        delete_domain
            (Domain a_domain);
    public Domain
        lookup_domain
            (String domainId);
    public int
        delete_contained_entities
            (void);
};

```

The following paragraphs describe the usage of all DomainParticipantFactory operations.

### 3.2.2.1 create\_participant

#### Scope

DDS.DomainParticipantFactory

#### Synopsis

```

import DDS.*;
public DomainParticipant
    create_participant
        (String domainId,
         DomainParticipantQos qos,
         DomainParticipantListener a_listener,
         int 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 String 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 DomainParticipantQos qos* - a DomainParticipantQos for the new DomainParticipant. When this set of QosPolicy settings is inconsistent, no DomainParticipant is created.
- in DomainParticipantListener a\_listener* - a reference 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 int mask* - a bit-mask in which each bit enables the invocation of the DomainParticipantListener for a certain status.

## Return Value

*DomainParticipant* - Return value is a reference to the newly created DomainParticipant. In case of an error, the null reference 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 reference. 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 reference 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 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.

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)*.



Be aware that the `PUBLICATION_MATCHED_STATUS` and `SUBSCRIPTION_MATCHED_STATUS` are not applicable when the infrastructure does not have the information available to determine connectivity. This is the case when OpenSplice is configured not to maintain discovery information in the Networking Service. (See the description for the `NetworkingService/Discovery/enabled` property in the Deployment Manual for more information about this subject.) In this case the operation will return `null`.

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 applicable 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

```
import DDS.*;
public int
    delete_participant
        (DomainParticipant a_participant);
```

#### Description

This operation deletes a DomainParticipant.

#### Parameters

*in DomainParticipant a\_participant* - a reference to the DomainParticipant, which is to be deleted.

#### Return Value

*int* - 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 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 reference.
- *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

```
import DDS.*;
public int
    get_default_participant_qos
        (DomainParticipantQosHolder qos);
```

#### Description

This operation gets the default `DomainParticipantQos` of the `DomainParticipantFactory`.

#### Parameters

*inout* `DomainParticipantQosHolder qos` - a reference to the destination `DomainParticipantQosHolder` object in which the default `DomainParticipantQos` for the `DomainParticipantFactory` is written.

#### Return Value

*int* - 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 object 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 an object of the `DomainParticipantQos` class in which the `QosPolicy` settings can be stored and provide a reference to the object. The operation writes the default `QosPolicy` settings to the object referenced to by `qos`. Any settings in the object 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 `QosPolicy` values as defined in Table 2, *DCPS Holder Classes*, on page 12.

#### Return Code

When the operation returns:



- *RETCODE\_OK* - the default DomainParticipant QosPolicy settings of this DomainParticipantFactory have successfully been copied into the specified DomainParticipantQosHolder 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

```
import DDS.*;
public static DomainParticipantFactory
    get_instance
        (void);
```

#### Description

This operation returns the DomainParticipantFactory singleton.

#### Parameters

<none>

#### Return Value

*DomainParticipantFactory* - return value is a reference 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

```
import DDS.*;
public int
    get_qos
        (DomainParticipantFactoryQosHolder qos);
```

## Description

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

## Parameters

*inout* `DomainParticipantFactoryQosHolder qos` - a reference to the destination `DomainParticipantFactoryQosHolder` object in which the `QosPolicy` settings will be copied.

## Return Value

*int* - 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 `DomainParticipantFactoryQosHolder` 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

```
import DDS.*;
```

```
public DomainParticipant
    lookup_participant
        (String domainId);
```

## Description

This operation retrieves a previously created `DomainParticipant` belonging to the specified `domainId`.

## Parameters

*in String 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* - Return value is a reference to the `DomainParticipant` retrieved. When no such `DomainParticipant` is found, the null reference 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. 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

```
import DDS.*;
int
    set_default_participant_qos
    (DomainParticipantQos qos);
```

## Description

This operation sets the default `DomainParticipantQos` of the `DomainParticipantFactory`.

## Parameters

*in* `DomainParticipantQos qos` - an object of the `DomainParticipantQos` class, which contains the new default `DomainParticipantQos` for the newly created `DomainParticipants`.

## Return Value

*int* - 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 object 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 that 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

```
import DDS.*;
public int
    set_qos
        (DomainParticipantFactoryQos qos);
```

#### Description

This operation replaces the existing set of `QosPolicy` settings for a `DomainParticipantFactory`.

#### Parameters

*in* `DomainParticipantFactoryQos qos` - must contain the new set of `QosPolicy` settings for the `DomainParticipantFactory`.

#### Return Value

*int* - 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 object 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

#### Scope

DDS.DomainParticipantFactory

#### Synopsis

```
import DDS.*;
public int
    delete_domain
        (Domain a_domain);
```

#### Description

This operation deletes a Domain proxy.

#### Parameters

*in Domain a\_domain* - a pointer to the Domain proxy, which is to be deleted.

#### Return Value

*int* - Possible return codes of the operation are:

RETCODE\_OK, RETCODE\_ERROR, DDS\_RETCODE\_BAD\_PARAMETER or  
DDS\_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

#### Scope

DDS.DomainParticipantFactory

#### Synopsis

```
import DDS.*;
```

```
public Domain
    lookup_domain
        (String 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 String 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* - Return value is a pointer to the `Domain` proxy retrieved. When no such `Domain` proxy is found or could be created, the null reference 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 reference.

The `domainId` used to search for a specific `Domain` proxy must not be a null reference, as a null reference will not be resolved on this level and in effect has no meaning.

## 3.2.2.11 delete\_contained\_entities

### Scope

```
DDS.DomainParticipantFactory
```

### Synopsis

```
import DDS.*;
public int
    delete_contained_entities
        (void);
```

### Description

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

## Parameters

<none>

## Return Value

*int* - Possible return codes of the operation are:

RETCODE\_OK, RETCODE\_ERROR, RETCODE\_OUT\_OF\_RESOURCES or  
RETCODE\_PRECONDITION\_NOT\_MET.

## Detailed Description

This operation deletes all of the `Entity` objects that were created on the `DomainParticipantFactory` (it deletes all contained `DomainParticipant` objects). Prior to deleting each contained `Entity`, this operation regressively calls the `delete_contained_entities` operation on each `Participant`. In other words, this operation cleans up *all* `Entity` objects in the process.



---

**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* - all contained `Entity` objects are deleted.

*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* - one or more of the contained entities are in a state where they cannot be deleted.

### 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 `Publishers` and the `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
 */
public class Domain {
    public int
        create_persistent_snapshot(
            String partition_expression,
            String topic_expression,
            String URI);
};
```

The following sections describe the usage of all Domain operations.

### 3.2.3.1 create\_persistent\_snapshot

#### Scope

DDS.Domain

#### Synopsis

```
public int
    create_persistent_snapshot(
        String partition_expression,
        String topic_expression,
        String 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 String partition\_expression* - The expression of all partitions involved in the snapshot; this may contain wildcards.

*in String topic\_expression* - The expression of all topics involved in the snapshot; this may contain wildcards.

*in String uri* - The location where to store the snapshot. Currently only directories are supported.

**Return Value**

*int* - Possible return codes of the operation are:

RETCODE\_OK, RETCODE\_ERROR, RETCODE\_BAD\_PARAMETER, RETCODE\_ALREADY\_DELETED or DDS\_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 a null reference.
- *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:

```
public interface DomainParticipantListener
{
//
// extends interface TopicListener
//
// void
//     on_inconsistent_topic
//         (Topic the_topic,
//          InconsistentTopicStatus status);
//
// extends interface PublisherListener
//
// void
//     on_offered_deadline_missed
//         (DataWriter writer,
//          OfferedDeadlineMissedStatus status);
//
// void
//     on_offered_incompatible_qos
//         (DataWriter writer,
//          OfferedIncompatibleQosStatus status);
//
// void
//     on_liveliness_lost
//         (DataWriter writer,
//          LivelinessLostStatus status);
//
// void
//     on_publication_matched
//         (DataWriter writer,
//          PublicationMatchedStatus status);
//
// extends interface SubscriberListener
//
// void
//     on_data_on_readers
//         (Subscriber subs);
// void
```

```

//      on_requested_deadline_missed
//      (DataReader reader,
//       RequestedDeadlineMissedStatus status);

// void
//      on_requested_incompatible_qos
//      (DataReader reader,
//       RequestedIncompatibleQosStatus status);

// void
//      on_sample_rejected
//      (DataReader reader,
//       SampleRejectedStatus status);

// void
//      on_liveliness_changed
//      (DataReader reader,
//       LivelinessChangedStatus status);

// void
//      on_data_available
//      (DataReader reader);

// void
//      on_subscription_matched
//      (DataReader reader,
//       SubscriptionMatchedStatus status);

// void
//      on_sample_lost
//      (DataReader reader,
//       SampleLostStatus status);
//
// implemented API operations
//      <no operations>
//
};

```

The following 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 interface `DataReaderListener` for further explanation.

### Synopsis

```
import DDS.*;
void
  on_data_available
    (DataReader reader);
```

#### 3.2.4.2 on\_data\_on\_readers (inherited, abstract)

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

### Synopsis

```
import DDS.*;
void
  on_data_on_readers
    (Subscriber subs);
```

#### 3.2.4.3 on\_inconsistent\_topic (inherited, abstract)

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

### Synopsis

```
import DDS.*;
void
  on_inconsistent_topic
    (Topic the_topic,
     InconsistentTopicStatus status);
```

#### 3.2.4.4 on\_liveliness\_changed (inherited, abstract)

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

### Synopsis

```
import DDS.*;
void
  on_liveliness_changed
    (DataReader reader,
     LivelinessChangedStatus status);
```

#### 3.2.4.5 on\_liveliness\_lost (inherited, abstract)

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

### Synopsis

```
import DDS.*;
void
    on_liveliness_lost
        (DataWriter writer,
         LivelinessLostStatus status);
```

#### 3.2.4.6 on\_offered\_deadline\_missed (inherited, abstract)

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

### Synopsis

```
import DDS.*;
void
    on_offered_deadline_missed
        (DataWriter writer,
         OfferedDeadlineMissedStatus status);
```

#### 3.2.4.7 on\_offered\_incompatible\_qos (inherited, abstract)

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

### Synopsis

```
import DDS.*;
void
    on_offered_incompatible_qos
        (DataWriter writer,
         OfferedIncompatibleQosStatus status);
```

#### 3.2.4.8 on\_publication\_matched (inherited, abstract)

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

### Synopsis

```
import DDS.*;
void
    on_publication_matched
        (DataWriter writer,
         PublicationMatchedException status);
```

#### 3.2.4.9 on\_requested\_deadline\_missed (inherited, abstract)

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

**Synopsis**

```
import DDS.*;
void
    on_requested_deadline_missed
        (DataReader reader,
         RequestedDeadlineMissedStatus status);
```

**3.2.4.10 on\_requested\_incompatible\_qos (inherited, abstract)**

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

**Synopsis**

```
import DDS.*;
void
    on_requested_incompatible_qos
        (DataReader reader,
         RequestedIncompatibleQosStatus status);
```

**3.2.4.11 on\_sample\_lost (inherited, abstract)**

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

**Synopsis**

```
import DDS.*;
void
    on_sample_lost
        (DataReader reader,
         SampleLostStatus status);
```

**NOTE:** This operation is not yet implemented. 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 interface `DataReaderListener` for further explanation.

**Synopsis**

```
import DDS.*;
void
    on_sample_rejected
        (DataReader reader,
         SampleRejectedStatus status);
```

**3.2.4.13 on\_subscription\_matched (inherited, abstract)**

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

## Synopsis

```
import DDS.*;
void
    on_subscription_matched
        (DataReader reader,
         SubscriptionMatchedStatus status);
```

### 3.2.5 ExtDomainParticipantListener interface

The `ExtDomainParticipantListener` interface is a subtype of both `DomainParticipantListener` and `ExtTopicListener` and thereby provides an additional OpenSplice-specific callback called `on_all_disposed_data`.




---

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 interface description of this class is as follows:

```
interface ExtDomainParticipantListener
{

    // inherited from ExtTopicListener
    //
    // void
    // on_all_data_disposed
    // (DDS.Topic the_topic);
    //
    // inherited from DomainParticipantListener
    //
    // void
    // on_inconsistent_topic
    // (DDS.Topic the_topic,
    //  DDS.InconsistentTopicStatus status);
    // void
    // on_offered_deadline_missed
    // (DDS.DataWriter writer,
    //  DDS.OfferedDeadlineMissedStatus status);
    // void
    // on_offered_incompatible_qos
    // (DDS.DataWriter writer,
    //  DDS.OfferedIncompatibleQosStatus status);
    // void
    // on_liveliness_lost
    // (DDS.DataWriter writer,
    //  DDS.LivelinessLostStatus status);
    // void
    // on_publication_matched
    // (DDS.DataWriter writer,
    //  DDS.PublicationMatchedStatus status);
```



```

// void
// on_data_on_readers
// (DDS.Subscriber subs);
// void
// on_requested_deadline_missed
// (DDS.DataReader reader,
// DDS.RequestedDeadlineMissedStatus status);
// void
// on_requested_incompatible_qos
// (DDS.DataReader reader,
// DDS.RequestedIncompatibleQosStatus status);
// void
// on_sample_rejected
// (DDS.DataReader reader,
// DDS.SampleRejectedStatus status);
// void
// on_liveliness_changed
// (DDS.DataReader reader,
// DDS.LivelinessChangedStatus status);
// void
// on_data_available
// (DDS.DataReader reader);
// void
// on_subscription_matched
// (DDS.DataReader reader,
// DDS.SubscriptionMatchedStatus status);
// void
// on_sample_lost
// (DDS.DataReader reader,
// DDS.SampleLostStatus status);
//
// implemented API operations
// <no operations>
//

};

```

The following paragraphs list all `ExtDomainParticipantListener` operations. Since these operations are all inherited, they are listed but not fully described because they are not implemented in this class. The full descriptions of these operations are given in the classes from which they are inherited.

### 3.2.5.1 `on_all_data_disposed` (inherited, abstract)

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

#### Synopsis

```
import DDS.*;
```

```
void
    on_all_data_disposed
        (DDS.Topic the_topic);
```

### 3.2.5.2 on\_data\_available (inherited, abstract)

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

#### Synopsis

```
import DDS.*;
void
    on_data_available
        (DDS.DataReader reader);
```

### 3.2.5.3 on\_data\_on\_readers (inherited, abstract)

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

#### Synopsis

```
import DDS.*;
void
    on_data_on_readers
        (DDS.Subscriber subs);
```

### 3.2.5.4 on\_inconsistent\_topic (inherited, abstract)

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

#### Synopsis

```
import DDS.*;
void
    on_inconsistent_topic
        (DDS.Topic the_topic,
         DDS.InconsistentTopicStatus status);
```

### 3.2.5.5 on\_liveliness\_changed (inherited, abstract)

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

#### Synopsis

```
import DDS.*;
void
    on_liveliness_changed
        (DDS.DataReader reader,
         DDS.LivelinessChangedStatus status);
```

### 3.2.5.6 on\_liveliness\_lost (inherited, abstract)

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

#### Synopsis

```
import DDS.*;
void
    on_liveliness_lost
        (DDS.DataWriter writer,
         DDS.LivelinessLostStatus status);
```

### 3.2.5.7 on\_offered\_deadline\_missed (inherited, abstract)

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

#### Synopsis

```
import DDS.*;
void
    on_offered_deadline_missed
        (DDS.DataWriter writer,
         DDS.OfferedDeadlineMissedStatus status);
```

### 3.2.5.8 on\_offered\_incompatible\_qos (inherited, abstract)

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

#### Synopsis

```
import DDS.*;
void
    on_offered_incompatible_qos
        (DDS.DataWriter writer,
         DDS.OfferedIncompatibleQosStatus status);
```

### 3.2.5.9 on\_publication\_matched (inherited, abstract)

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

#### Synopsis

```
import DDS.*;
void
    on_publication_matched
        (DDS.DataWriter writer,
         DDS.PublicationMatchedException status);
```

**3.2.5.10 on\_requested\_deadline\_missed (inherited, abstract)**

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

**Synopsis**

```
import DDS.*;
void
    on_requested_deadline_missed
        (DDS.DataReader reader,
         DDS.RequestedDeadlineMissedStatus status);
```

**3.2.5.11 on\_requested\_incompatible\_qos (inherited, abstract)**

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

**Synopsis**

```
import DDS.*;
void
    on_requested_incompatible_qos
        (DDS.DataReader reader,
         DDS.RequestedIncompatibleQosStatus status);
```

**3.2.5.12 on\_sample\_lost (inherited, abstract)**

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

**Synopsis**

```
import DDS.*;
void
    on_sample_lost
        (DDS.DataReader reader,
         DDS.SampleLostStatus status);
```

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

**3.2.5.13 on\_sample\_rejected (inherited, abstract)**

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

**Synopsis**

```
import DDS.*;
void
    on_sample_rejected
        (DDS.DataReader reader,
         DDS.SampleRejectedStatus status);
```

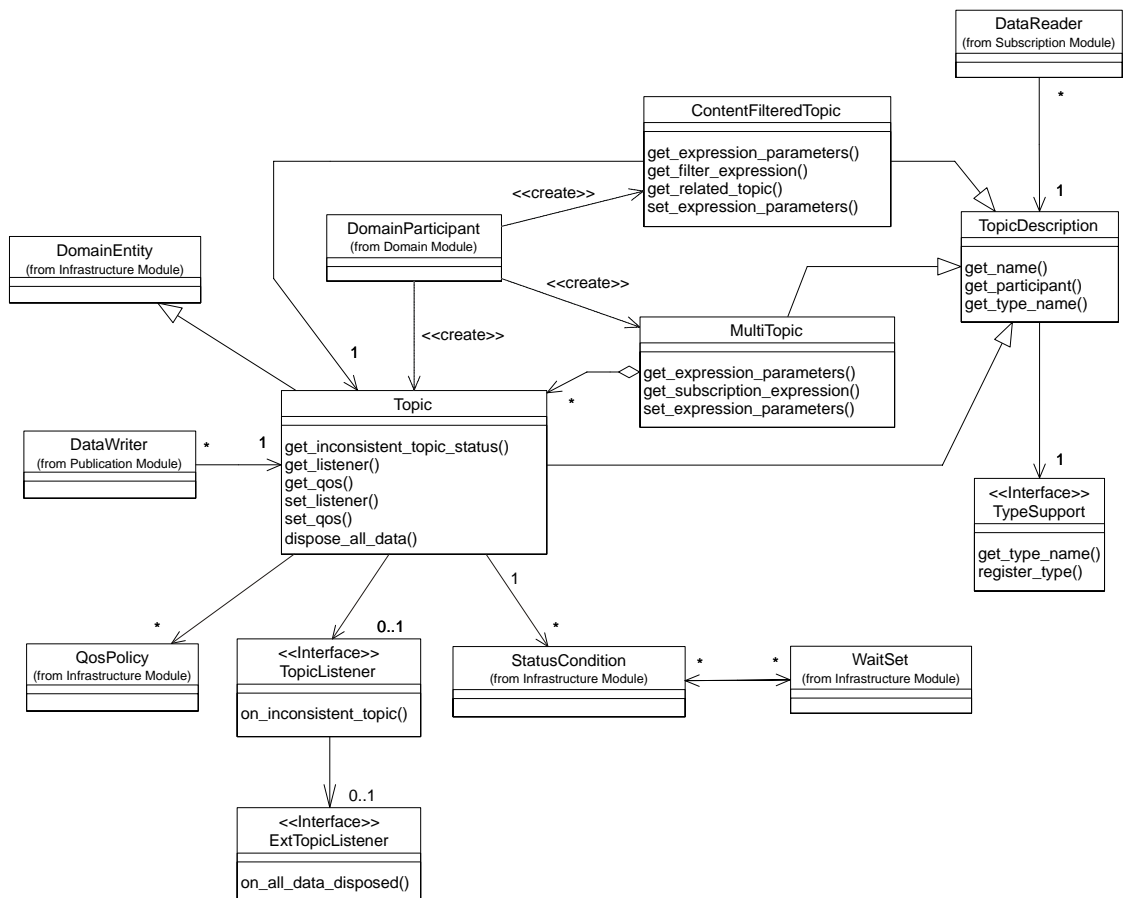
#### 3.2.5.14 on\_subscription\_matched (inherited, abstract)

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

## Synopsis

```
import DDS.*;
void
    on_subscription_matched
        (DDS.DataReader reader,
         DDS.SubscriptionMatchedStatus status);
```

### 3.3 Topic-Definition Module

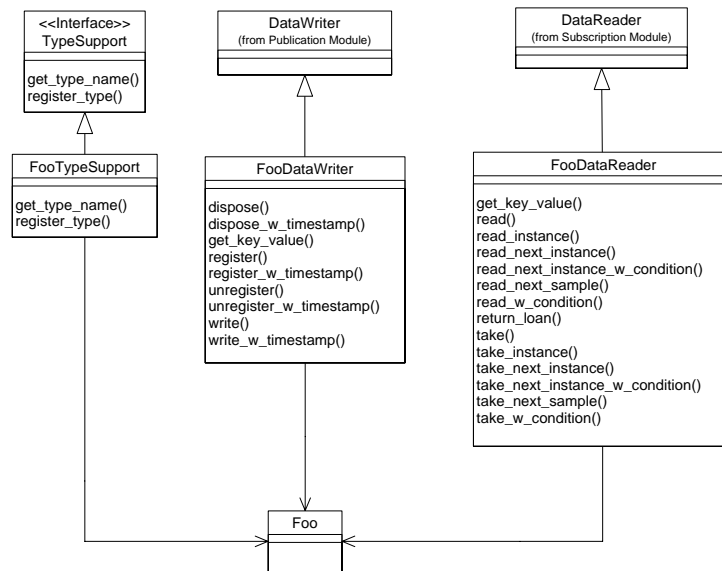


**Figure 16 DCPS Topic-Definition Module’s Class Model**

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 Pre-processor Generation of the Typed Classes for Data Type “Foo”**

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 Interface TopicDescription

This is the interface 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 is as follows:

```
public interface TopicDescription
{
    //
    // implemented API operations
    //
    public String
        get_type_name
        (void);
    public String
        get_name
        (void);
    public DomainParticipant
        get_participant
        (void);
};
```

The following paragraphs describe the usage of all `TopicDescription` operations.

### 3.3.1.1 `get_name`

#### Scope

```
DDS.TopicDescription
```

#### Synopsis

```
import DDS.*;
public String
    get_name
    (void);
```

#### Description

This operation returns the name used to create the `TopicDescription`.

#### Parameters

<none>

#### Return Value

*String* - return value 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

```
import DDS.*;
public DomainParticipant
    get_participant
        (void);
```

#### Description

This operation returns the `DomainParticipant` associated with the `TopicDescription` or the null reference.

#### Parameters

<none>

#### Return Value

*DomainParticipant* - a reference to the `DomainParticipant` associated with the `TopicDescription` or the null reference.

#### 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 reference is returned.

### 3.3.1.3 `get_type_name`

#### Scope

DDS.TopicDescription

#### Synopsis

```
import DDS.*;
public String
    get_type_name
        (void);
```

#### Description

This operation returns the registered name of the data type associated with the `TopicDescription`.



## Parameters

<none>

## Return Value

*String* - return value is 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 Interface 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 is as follows:

```
public interface Topic
{
    //
    // extends interface Entity
    //
    // public StatusCondition
    //     get_statuscondition
    //         (void);
    // public int
    //     get_status_changes
    //         (void);
    // public int
    //     enable
    //         (void);
    //
    // extends interface TopicDescription
    //
    // public String
    //     get_type_name
    //         (void);

    // public String
    //     get_name
    //         (void);

    // public DomainParticipant
```

```

//      get_participant
//      (void);
//
// implemented API operations
//
    public int
        set_qos
            (TopicQos qos);
    public int
        get_qos
            (TopicQosHolder qos);
    public int
        set_listener
            (TopicListener a_listener,
             int mask);
    public TopicListener
        get_listener
            (void);
    public int
        get_inconsistent_topic_status
            (InconsistentTopicStatusHolder status);
    public int dispose_all_data ();
};

```

The following 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 interface `Entity` for further explanation.

#### Synopsis

```

import DDS.*;
public int
    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

```
import DDS.*;
```

```
public int
    get_inconsistent_topic_status
        (InconsistentTopicStatusHolder status);
```

## Description

This operation obtains the `InconsistentTopicStatus` of the `Topic`.

## Parameters

*inout InconsistentTopicStatusHolder status* - the contents of the `InconsistentTopicStatus` object of the `Topic` will be copied into the `InconsistentTopicStatusHolder` specified by *status*.

## Return Value

*int* - 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 *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 `dispose_all_data`

## Synopsis

```
import DDS.*;
int dispose_all_data ();
```

## Description

This operation allows the application to dispose of all of the instances for a particular topic without the network overhead of using an individual dispose call for each instance individually.

## Parameters

<none>

## Return Value

*int* - Possible return codes of the operation are: `RETCODE_OK`, `DDS_RETCODE_ERROR`, `DDS_RETCODE_OUT_OF_RESOURCES`.

## DetailedDescription

This operation allows the application to dispose of all of the instances for a particular topic without the network overhead of using an individual `dispose` call for each instance individually. Its effect is equivalent to invoking a separate `dispose` operation for each individual instance on the `DataWriter` that owns it. (See the description of `FooDataWriter.dispose` in Section 3.4.2.33, *dispose*, on page 276.)



This operation *only* sets the instance state of the instances concerned to `NOT_ALIVE_DISPOSED`. It does *not* unregister the instances, and so does not automatically clean up the memory that is claimed by the instances in both the `DataReaders` and `DataWriters`.

### Blocking

The blocking (or nonblocking) behaviour of this call is undefined.

### Concurrency

If there are subsequent calls to this function before the action has been completed (completion of the disposes on all nodes, not simply return from the function), then the behaviour is undefined.

### Other notes

The effect of this call on `disposed_generation_count`, `generation_rank` and `absolute_generation_rank` is undefined.

### Return Code

- `RETCODE_OK` - a request to dispose the topic has been successfully queued.
- `DDS_RETCODE_ERROR` - an internal error has occurred.
- `DDS_RETCODE_OUT_OF_RESOURCES` - the Data Distribution Service ran out of resources to complete this operation.
- `RETCODE_ALREADY_DELETED` - the Topic has already been deleted.
- `RETCODE_NOT_ENABLED` - the Topic is not enabled.

### 3.3.2.4 `get_listener`

#### Scope

`DDS.Topic`

#### Synopsis

```
import DDS.*;
public TopicListener
    get_listener
        (void);
```

#### Description

This operation allows access to a `TopicListener`.

#### Parameters

<none>

#### Return Value

*TopicListener* - result is a reference 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 reference is returned.

### 3.3.2.5 `get_name` (inherited)

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

#### Synopsis

```
import DDS.*;
public String
    get_name
        (void);
```

### 3.3.2.6 `get_participant` (inherited)

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

## Synopsis

```
import DDS.*;
public DomainParticipant
    get_participant
        (void);
```

### 3.3.2.7 get\_qos

## Scope

DDS.Topic

## Synopsis

```
import DDS.*;
public int
    get_qos
        (TopicQosHolder qos);
```

## Description

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

## Parameters

*inout TopicQosHolder qos* - a reference to the destination TopicQosHolder object in which the QosPolicy settings will be copied.

## Return Value

*int* - 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 referenced 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 TopicQosHolder 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.

### 3.3.2.8 `get_status_changes` (inherited)

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

#### Synopsis

```
import DDS.*;
public int
    get_status_changes
        (void);
```

### 3.3.2.9 `get_statuscondition` (inherited)

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

#### Synopsis

```
import DDS.*;
public StatusCondition
    get_statuscondition
        (void);
```

### 3.3.2.10 `get_type_name` (inherited)

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

#### Synopsis

```
import DDS.*;
public String
    get_type_name
        (void);
```

### 3.3.2.11 `set_listener`

#### Scope

```
DDS.Topic
```

#### Synopsis

```
import DDS.*;
public int
    set_listener
        (TopicListener a_listener,
         int mask);
```

#### Description

This operation attaches a `TopicListener` to the `Topic`.

## Parameters

*in TopicListener a\_listener* - a reference to the TopicListener instance, which will be attached to the Topic.

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

## Return Value

*int* - 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 reference, it represents a listener that is treated as a NOOP<sup>1</sup> for all statuses activated in the bitmask.

### 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.

---

1. Short for **No-Operation**, an instruction that performs nothing at all.



### 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.

## 3.3.2.12 `set_qos`

### Scope

`DDS.Topic`

### Synopsis

```
import DDS.*;
public int
    set_qos
        (TopicQos qos);
```

### Description

This operation replaces the existing set of `QosPolicy` settings for a `Topic`.

### Parameters

*in* `TopicQos qos` - contains the new set of `QosPolicy` settings for the `Topic`.

### Return Value

*int* - 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 object 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.
- `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 Interface `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 interface 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 is as follows:

```
public interface ContentFilteredTopic
{
    //
    // extends interface TopicDescription
    //
    // public String
    //     get_type_name
    //         (void);

    // public String
    //     get_name
    //         (void);

    // public DomainParticipant
    //     get_participant
    //         (void);
    //
    // implemented API operations
    //
    public String
        get_filter_expression
            (void);

    public int
        get_expression_parameters
            (StringSeqHolder expression_parameters);

    public int
        set_expression_parameters
            (StringSeq expression_parameters);

    public Topic
        get_related_topic
            (void);
};
```

The following paragraphs describe the usage of all `ContentFilteredTopic` operations.

### 3.3.3.1 `get_expression_parameters`

#### Scope

`DDS.ContentFilteredTopic`

#### Synopsis

```
import DDS.*;
public int
    get_expression_parameters
        (StringSeqHolder expression_parameters);
```

#### Description

This operation obtains the expression parameters associated with the `ContentFilteredTopic`.

#### Parameters

*inout StringSeqHolder expression\_parameters* - a reference to the destination `StringSeqHolder` object in which the the parameters used in the SQL expression will be copied.

#### Return Value

*int* - 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

```
import DDS.*;
public String
    get_filter_expression
        (void);
```

#### Description

This operation returns the *filter\_expression* associated with the *ContentFilteredTopic*.

#### Parameters

<none>

#### Return Value

*String* - result is 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**

```
import DDS.*;
public String
    get_name
        (void);
```

**3.3.3.4 get\_participant (inherited)**

This operation is inherited and therefore not described here. See the interface TopicDescription for further explanation.

**Synopsis**

```
import DDS.*;
public DomainParticipant
    get_participant
        (void);
```

**3.3.3.5 get\_related\_topic****Scope**

```
DDS.ContentFilteredTopic
```

**Synopsis**

```
import DDS.*;
public Topic
    get_related_topic
        (void);
```

**Description**

This operation returns the Topic associated with the ContentFilteredTopic.

**Parameters**

<none>

**Return Value**

*Topic* - result is a reference 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 interface `TopicDescription` for further explanation.

#### Synopsis

```
import DDS.*;
public String
    get_type_name
        (void);
```

### 3.3.3.7 `set_expression_parameters`

#### Scope

`DDS.ContentFilteredTopic`

#### Synopsis

```
import DDS.*;
public int
    set_expression_parameters
        (String[] expression_parameters);
```

#### Description

This operation changes the expression parameters associated with the `ContentFilteredTopic`.

#### Parameters

*in* `String[] 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

*int* - 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` (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). 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 Interface 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 based expression with parameters to adapt the filter clause.

The interface description is as follows:

```
public interface MultiTopic
{
//
// extends interface TopicDescription
//
// public String
//   get_type_name
//   (void);

// public String
//   get_name
//   (void);

// public DomainParticipant
//   get_participant
//   (void);
```



```
//
// implemented API operations
//
    public String
        get_subscription_expression
            (void);

    public int
        get_expression_parameters
            (StringSeqHolder expression_parameters);

    public int
        set_expression_parameters
            (String[] expression_parameters);
};
```

The following 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

```
import DDS.*;
public int
    get_expression_parameters
        (StringSeqHolder expression_parameters);
```

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

#### Description

This operation obtains the expression parameters associated with the `MultiTopic`.

#### Parameters

*inout StringSeqHolder expression\_parameters* - a reference to the destination `StringSeqHolder` object in which the the parameters used in the SQL expression will be copied.

## Return Value

*int* - 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 interface `TopicDescription` for further explanation.

### Synopsis

```
import DDS.*;
public String
    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 interface `TopicDescription` for further explanation.

**Synopsis**

```
import DDS.*;
public DomainParticipant
    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**

```
import DDS.*;
public String
    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**

*String* - 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 interface `TopicDescription` for further explanation.

## Synopsis

```
import DDS.*;
public String
    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

```
import DDS.*;
public int
    set_expression_parameters
        (String[] 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 String[] expression\_parameters* - the handle to a sequence of strings with the parameters used in the SQL expression.

## Return Value

*int* - 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 to the `TopicListener` interface. **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 is as follows:

```
public interface TopicListener
{
    //
    // abstract external operations
    //
    public void
        on_inconsistent_topic
        (Topic the_topic,
         InconsistentTopicStatus status);
    //
    // 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

```
import DDS.*;
public void
    on_inconsistent_topic
        (Topic the_topic,
         InconsistentTopicStatus status);
```

#### Description

This operation must be implemented by the application and is called by the Data Distribution Service when the `InconsistentTopicStatus` changes.

#### Parameters

*in Topic the\_topic* - contain a reference to the `Topic` on which the conflict occurred (this is an input to the application).

*in InconsistentTopicStatus status* - contain the `InconsistentTopicStatus` object (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 object of the class `InconsistentTopicStatus`.

### 3.3.6 ExtTopicListener interface

The `ExtTopicListener` interface is a subtype of `TopicListener` and provides an OpenSplice-specific callback `on_all_disposed_data`.




---

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 interface description of this class is as follows:

```
interface ExtTopicListener
{
//
// abstract external operations
//
void
    on_all_data_disposed(DDS.Topic the_topic);
//
// implemented API operations
// <no operations>
//
};
```

### 3.3.6.1 on\_all\_data\_disposed (abstract)

#### Scope

DDS.ExtTopicListener

#### Synopsis

```
import DDS.*
void
    on_all_data_disposed(DDS.Topic the_topic);
```

#### Description

This operation must be implemented by the application and is called by the Data Distribution Service when the node has completed disposal of data as a result of a call to `Topic.dispose_all_data()`.

#### Parameters

*in DDS.Topic the\_topic* - contains a reference to the Topic object which has been disposed.

#### 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 node has completed disposal of data as a result of a call to `Topic.dispose_all_data()`.

The implementation may be left empty when this functionality is not needed. This operation will only be called when the relevant `ExtTopicListener` is installed.

### Concurrency

The threading behaviour of calls to this method are undefined, so:

- Subsequent disposal via `Topic.dispose_all_data`, and the associated callbacks may be blocked until this method returns.
- This method may be called concurrently by `OpenSplice` if other `dispose_all_data` operations complete before this method returns.

## 3.3.7 Topic-Definition type specific interfaces

This paragraph describes the generic `TypeSupport` interfaces 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` interface from an IDL type description. The `FooTypeSupport` interface that would be generated by the pre-processor for a fictional type `Foo` describes the `<type>TypeSupport` interfaces.

### 3.3.7.1 Interface `TypeSupport`

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 an extended interface of the `TypeSupport` interface. An extended interface is generated for each data type used by the application, by calling the pre-processor.

The interface description is as follows:

```
public interface TypeSupport
{
    //
    // operations
    //
    // public int
    //     register_type
    //         (Domainparticipant domain,
    //         String type_name);
    //
    // public String
    //     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.

### 3.3.7.2 `get_type_name`

This 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` inherited from the `TypeSupport` class.

#### Synopsis

```
import DDS.*;
public String
    get_type_name
        (void);
```

### 3.3.7.3 `register_type`

This 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` inherited from the `TypeSupport` class.

#### Synopsis

```
import DDS.*;
public int
    register_type
        (Domainparticipant domain,
         String type_name);
```

### 3.3.7.4 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` which implements the `TypeSupport` interface. In this paragraph, the class `FooTypeSupport` describes the operations of these `<type>TypeSupport` interfaces as an example for the fictional application type `Foo` (defined in the module `SPACE`).

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 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:

```
public class FooTypeSupport
{
    //
    // implemented API operations
    //
    public int
        register_type
        (DomainParticipant domain,
         String type_name);
    public String
        get_type_name
        (void);
};
```

The next paragraph describes the usage of the `FooTypeSupport` operation.

### 3.3.7.5 `get_type_name`

#### Scope

`SPACE.FooTypeSupport`

#### Synopsis

```
import DDS.*;
public String
    get_type_name
    (void);
```

#### Description

This operation returns the default name of the data type associated with the `FooTypeSupport`.

#### Parameters

`<none>`

#### Return Value

*String* - return value is 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.

### 3.3.7.6 register\_type

#### Scope

`SPACE.FooTypeSupport`

#### Synopsis

```
import DDS.*;
public int
    register_type
        (DomainParticipant domain,
         String type_name);
```

#### Description

This operation registers a new data type name to a `DomainParticipant`.

#### Parameters

*in DomainParticipant domain* - a reference to a `DomainParticipant` object to which the new data type is registered.

*in String type\_name* - a local alias of the new data type to be registered.

#### Return Value

*int* - 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 (that is, 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 reference 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.

## 3.3.8 `on_all_data_disposed` (abstract)

### Scope

`DDS.ExtDomainParticipantListener`

### Synopsis

```
import DDS.*;
void
    on_all_data_disposed(DDS.Topic the_topic);
```

### Description

This operation must be implemented by the application and is called by the Data Distribution Service when the node has completed disposal of data as a result of a call to `Topic.dispose_all_data()`.

### Parameters

*in* `DDS.Topic the_topic` - contains a reference to the `Topic` object which has been disposed.

## 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 node has completed disposal of data as a result of a call to `Topic.dispose_all_data()`.

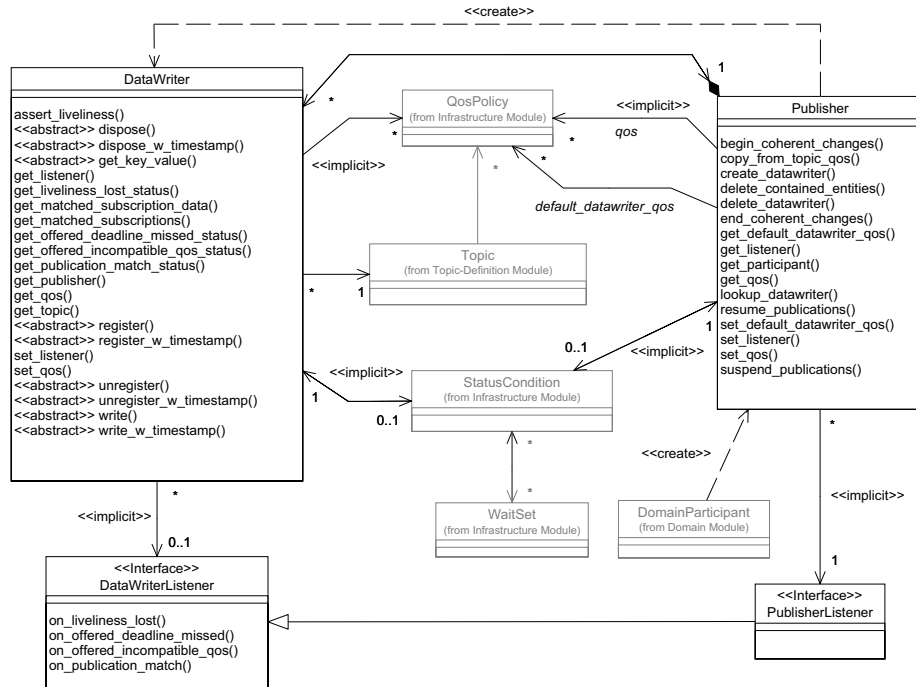
The implementation may be left empty when this functionality is not needed. This operation will only be called when the relevant `ExtDomainParticipantListener` is installed.

## Concurrency

The threading behaviour of calls to this method are undefined, so:

- Subsequent disposal via `Topic.dispose_all_data`, and the associated callbacks may be blocked until this method returns.
- This method may be called concurrently by `OpenSplice` if other `dispose_all_data` operations complete before this method returns.

## 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 interface 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 Interface 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:

```
public interface Publisher
{
    //
    // extends interface Entity
    //
    // public StatusCondition
    //     get_statuscondition
    //         (void);
    // public int
    //     get_status_changes
    //         (void);
    // public int
    //     enable
    //         (void);
    //
    // implemented API operations
    //
    public DataWriter
        create_datawriter
            (Topic a_topic,
             DataWriterQos qos,
             DataWriterListener a_listener,
             int mask);

    public int
        delete_datawriter
            (DataWriter a_datawriter);

    public DataWriter
        lookup_datawriter
```

```
        (String topic_name);
public int
    delete_contained_entities
        (void);

public int
    set_qos
        (PublisherQos qos);
public int
    get_qos
        (PublisherQosHolder qos);
public int
    set_listener
        (PublisherListener a_listener,
         int mask);
public PublisherListener
    get_listener
        (void);
public int
    suspend_publications
        (void);

public int
    resume_publications
        (void);

public int
    begin_coherent_changes
        (void);

public int
    end_coherent_changes
        (void);

public int
    wait_for_acknowledgments
        (Duration_t max_wait);

public DomainParticipant
    get_participant
        (void);

public int
    set_default_datawriter_qos
        (DataWriterQos qos);

public int
    get_default_datawriter_qos
        (DataWriterQosHolder qos);
```



```

        public int
            copy_from_topic_qos
            (DataWriterQosHolder a_datawriter_qos,
             TopicQos a_topic_qos);
    };

```

The following 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 interfaces from which they are inherited.

### 3.4.1.1 `begin_coherent_changes`

#### Scope

`DDS.Publisher`

#### Synopsis

```

import DDS.*;
public int
    begin_coherent_changes
        (void);

```

#### 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

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

#### 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 precondition for making coherent changes is that the `PresentationQos` of the `Publisher` has its `coherent_access` attribute set to `true`. If this is not the case, the `Publisher` will not accept any coherent start requests and return `RETCODE_PRECONDITION_NOT_MET`.

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_OK` - a new coherent change has successfully been started.
- `RETCODE_ERROR` - an internal error has occurred.
- `RETCODE_ALREADY_DELETED` - the `Publisher` has already been deleted.
- `RETCODE_PRECONDITION_NOT_MET` - the `Publisher` is not able to handle coherent changes because its `PresentationQos` has not set `coherent_access` to `TRUE`.

### 3.4.1.2 `copy_from_topic_qos`

#### Scope

`DDS.Publisher`

#### Synopsis

```
import DDS.*;
public int
    copy_from_topic_qos
        (DataWriterQosHolder a_datawriter_qos,
         TopicQos a_topic_qos);
```

## Description

This operation will copy policies in `a_topic_qos` to the corresponding policies in `a_datawriter_qos`.

## Parameters

*inout* `DataWriterQosHolder a_datawriter_qos` - the destination `DataWriterQos` object to which the `QosPolicy` settings should be copied.

*in* `TopicQos a_topic_qos` - the source `TopicQos` object, which should be copied.

## Return Value

*int* - 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

```
import DDS.*;
public DataWriter
    create_datawriter
        (Topic a_topic,
         DataWriterQos qos,
         DataWriterListener a_listener,
         int 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 a\_topic* - a reference to the topic for which the `DataWriter` is created.

*in DataWriterQos qos* - the `DataWriterQos` for the new `DataWriter`. In case these settings are not self consistent, no `DataWriter` is created.

*in DataWriterListener a\_listener* - a reference 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.

*in int mask* - a bit-mask in which each bit enables the invocation of the `DataWriterListener` for a certain status.

#### Return Value

*DataWriter* - Return value is a reference to the newly created `DataWriter`. In case of an error, the null reference 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 reference 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 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.

The following statuses are applicable to the `DataWriterListener`:

- `OFFERED_DEADLINE_MISSED_STATUS`
- `OFFERED_INCOMPATIBLE_QOS_STATUS`
- `LIVELINESS_LOST_STATUS`
- `PUBLICATION_MATCHED_STATUS`.



Be aware that the `PUBLICATION_MATCHED_STATUS` is not applicable when the infrastructure does not have the information available to determine connectivity. This is the case when `OpenSplice` is configured not to maintain discovery information in the Networking Service. (See the description for the `NetworkingService/Discovery/enabled` property in the Deployment Manual for more information about this subject.) In this case the operation will return `null`.

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 applicable 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

```
import DDS.*;
public int
    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

*int* - 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

```
import DDS.*;
public int
    delete_datawriter
        (DataWriter a_datawriter);
```

#### Description

This operation deletes a `DataWriter` that belongs to the `Publisher`.

#### Parameters

*in* `DataWriter a_datawriter` - a reference to the `DataWriter`, which is to be deleted.

#### Return Value

*int* - 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` reference.
- `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

```
import DDS.*;
public int
    enable
    (void);
```

#### 3.4.1.7 end\_coherent\_changes

##### Scope

DDS.Publisher

##### Synopsis

```
import DDS.*;
public int
    end_coherent_changes
    (void);
```

##### Description

This operation terminates the ‘coherent set’ initiated by the matching call to *begin\_coherent\_changes*.

##### Parameters

<none>

##### Return Value

*int* - Possible return codes of the operation are: *RETCODE\_OK*, *RETCODE\_ERROR*, *RETCODE\_ALREADY\_DELETED* or *RETCODE\_PRECONDITION\_NOT\_MET*.

##### Detailed Description

This operation terminates the ‘coherent set’ initiated by the matching call to *begin\_coherent\_changes*. If there is no matching call to *begin\_coherent\_changes*, the operation will return the error *PRECONDITION\_NOT\_MET*.

##### Return Code

When the operation returns:

- *RETCODE\_OK* - the coherent change has successfully been closed.
- *RETCODE\_ERROR* - an internal error has occurred.
- *RETCODE\_ALREADY\_DELETED* - the Publisher has already been deleted.
- *RETCODE\_PRECONDITION\_NOT\_MET* - there is no matching *begin\_coherent\_changes* call that can be closed.

### 3.4.1.8 `get_default_datawriter_qos`

#### Scope

DDS.Publisher

#### Synopsis

```
import DDS.*;
public int
    get_default_datawriter_qos
        (DataWriterQosHolder qos);
```

#### Description

This operation gets the default *DataWriterQos* of the Publisher.

#### Parameters

*inout DataWriterQosHolder qos* - a reference to the destination *DataWriterQosHolder* object in which the default *DataWriterQos* for the Publisher is written.

#### Return Value

*int* - 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 object 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* object in which the *QosPolicy* settings can be stored and pass the *qos* reference to the operation. The operation writes the default *DataWriterQos* to the object referenced to by *qos*. Any settings in the object 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.

### Return Code

When the operation returns:

- `RETCODE_OK` - the default `DataWriterQosPolicy` settings of this `Publisher` have successfully been copied into the specified `DataWriterQosHolder` 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

```
import DDS.*;
public PublisherListener
    get_listener
        (void);
```

### Description

This operation allows access to a `PublisherListener`.

### Parameters

<none>

### Return Value

*PublisherListener* - result is a reference 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 reference is returned.

### 3.4.1.10 get\_participant

#### Scope

DDS.Publisher

#### Synopsis

```
import DDS.*;
public DomainParticipant
    get_participant
        (void);
```

#### Description

This operation returns the DomainParticipant associated with the Publisher or the null reference.

#### Parameters

<none>

#### Return Value

*DomainParticipant* - a reference to the DomainParticipant associated with the Publisher or the null reference.

#### 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 reference is returned.

### 3.4.1.11 get\_qos

#### Scope

DDS.Publisher

#### Synopsis

```
import DDS.*;
public int
    get_qos
        (PublisherQosHolder qos);
```

#### Description

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

## Parameters

*inout PublisherQosHolder qos* - the destination `PublisherQosHolder` object in which the `QosPolicy` settings will be copied.

## Return Value

*int* - 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 in 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 `PublisherQosHolder` 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

```
import DDS.*;
public int
    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

```
import DDS.*;
public StatusCondition
    get_statuscondition
        (void);
```

### 3.4.1.14 lookup\_datawriter

#### Scope

DDS.Publisher

#### Synopsis

```
import DDS.*;
public DataWriter
    lookup_datawriter
        (String 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 String topic\_name* - the name of the `Topic`, which is attached to the `DataWriter` to look for.

#### Return Value

*DataWriter* - Return value is a reference to the `DataWriter` found. When no such `DataWriter` is found, the null reference 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

```
import DDS.*;
public int
    resume_publications
        (void);
```

#### Description

This operation resumes a previously suspended publication.

## Parameters

<none>

## Return Value

*int* - 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

```
import DDS.*;
public int
    set_default_datawriter_qos
        (DataWriterQos qos);
```

#### Description

This operation sets the default `DataWriterQos` of the Publisher.

## Parameters

*in DataWriterQos qos* - an instance of the DataWriterQos class, which contains the new default DataWriterQos for the newly created DataWriters.

## Return Value

*int* - 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 compound class 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.
- *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

```
import DDS.*;
```



```

public int
    set_listener
        (PublisherListener a_listener,
         int mask);

```

## Description

This operation attaches a `PublisherListener` to the `Publisher`.

## Parameters

*in PublisherListener a\_listener* - a reference to the `PublisherListener` instance, which will be attached to the `Publisher`.

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

## Return Value

*int* - Possible return codes of the operation are: `RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_UNSUPPORTED`, `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 reference, it represents a listener that is treated as a NOOP<sup>1</sup> for all statuses activated in the bitmask.

### 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)*

---

1. Short for **No-Operation**, an instruction that performs nothing at all.

- `LIVELINESS_LOST_STATUS` *(propagated)*
- `PUBLICATION_MATCHED_STATUS` *(propagated).*



Be aware that the `PUBLICATION_MATCHED_STATUS` is not applicable when the infrastructure does not have the information available to determine connectivity. This is the case when OpenSplice is configured not to maintain discovery information in the Networking Service. (See the description for the `NetworkingService/Discovery/enabled` property in the Deployment Manual for more information about this subject.) In this case the operation will return `RETCODE_UNSUPPORTED`.

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 applicable 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 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

- `RETCODE_OK` - the `PublisherListener` is attached.
- `RETCODE_ERROR` - an internal error has occurred.
- `RETCODE_UNSUPPORTED` - a status was selected that cannot be supported because the infrastructure does not maintain the required connectivity information.
- `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

```
import DDS.*;
public int
    set_qos
        (PublisherQos qos);
```

#### Description

This operation replaces the existing set of `QosPolicy` settings for a `Publisher`.

#### Parameters

*in* `PublisherQos qos` - the new set of `QosPolicy` settings for the `Publisher`.

#### Return Value

*int* - 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 `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 `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`.
- `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

```
import DDS.*;
public int
    suspend_publications
        (void);
```

##### Description

This operation will suspend the dissemination of the publications by all contained *DataWriter* objects.

##### Parameters

<none>

##### Return Value

*int* - 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 or disposed 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):

- `DataWriter.dispose`
- `DataWriter.dispose_w_timestamp`
- `DataWriter.write`
- `DataWriter.write_w_timestamp`
- `DataWriter.writedispose`
- `DataWriter.writedispose_w_timestamp`
- `DataWriter.unregister_instance`
- `DataWriter.unregister_instance_w_timestamp`

Subsequent calls to this operation have no effect. When the `Publisher` is deleted before `resume_publication` 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

```
import DDS.*;
public int
    wait_for_acknowledgments
        (Duration_t max_wait);
```

## Description

This operation blocks the calling thread until either all data written by all contained `DataWriters` is acknowledged by the local infrastructure, or until the duration specified by `max_wait` parameter elapses, whichever happens first.

## Parameters

*in* `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

*int* - 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 blocks the calling thread until either all data written by all contained `DataWriters` is acknowledged by the local infrastructure, or until the duration specified by `max_wait` parameter elapses, whichever happens first.

Data is acknowledged by the local infrastructure when it does not need to be stored in its `DataWriter`'s local history. When a locally-connected subscription (including the networking service) has no more resources to store incoming samples it will start to reject these samples, resulting in their source `DataWriters` to store them temporarily in their own local history to be retransmitted at a later moment in time. In such scenarios, the `wait_for_acknowledgments` operation will block until all contained `DataWriters` have retransmitted their entire history, which is therefore effectively empty, or until the `max_wait` timeout expires, whichever happens first. In the first case the operation will return `RETCODE_OK`, in the latter it will return `RETCODE_TIMEOUT`.



Be aware that in case the operation returns `RETCODE_OK`, the data has only been acknowledged by the local infrastructure: it does not mean all remote subscriptions have already received the data. However, delivering the data to remote nodes is then the sole responsibility of the networking service: even when the publishing application would terminate, all data that has not yet been received may be considered 'on-route' and will therefore eventually arrive (unless the networking service itself will crash). In contrast, if a `DataWriter` would still have data in its local history buffer when it terminates, this data is considered 'lost'.

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`, since best-effort `DataWriters` will never store rejected samples in their local history: they will just drop them and continue business as usual.

### Return Code

When the operation returns:

- `RETCODE_OK` - the data of all contained `DataWriters` has been acknowledged by the local infrastructure.
- `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_TIMEOUT` - not all data is acknowledged before `max_wait` elapsed.

## 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 Interface `DataWriter`

`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 interface. It must be extended for each particular application data type. For a fictional application data type `Foo` (defined in the module `SPACE`) the extended interface would be `SPACE.FooDataWriter`.

The interface description is as follows:

```
public interface DataWriter
{
//
// extends interface Entity
```

```

//
// public StatusCondition
//     get_statuscondition
//         (void);
// public int
//     get_status_changes
//         (void);
// public int
//     enable
//         (void);
//
// abstract operations (implemented in the data type specific
//   DataWriter)
//
// public long
//     register_instance
//         (<data> instance_data);
//
// public long
//     register_instance_w_timestamp
//         (<data> instance_data,
//          Time_t source_timestamp);
//
// public int
//     unregister_instance
//         (<data> instance_data,
//          long handle);
//
// public int
//     unregister_instance_w_timestamp
//         (<data> instance_data,
//          long handle,
//          Time_t source_timestamp);
//
// public int
//     write
//         (<data> instance_data,
//          long handle);
//
// public int
//     write_w_timestamp
//         (<data> instance_data,
//          long handle,
//          Time_t source_timestamp);
//
// public int
//     dispose
//         (<data> instance_data,
//          long instance_handle);
//

```



```

// public int
//   dispose_w_timestamp
//     (<data> instance_data,
//      long instance_handle,
//      Time_t source_timestamp);
//
// public int
//   writedispose
//     (<data> instance_data,
//      long instance_handle);
//
// public int
//   writedispose_w_timestamp
//     (<data> instance_data,
//      long instance_handle,
//      Time_t source_timestamp);
//
// public int
//   get_key_value
//     (<data>Holder key_holder,
//      long handle);
//
// public long
//   lookup_instance
//     (<data> instance_data);
//
// implemented API operations
//
public int
  set_qos
    (DataWriterQos qos);

public int
  get_qos
    (DataWriterQosHolder qos);

public int
  set_listener
    (DataWriterListener a_listener,
     int mask);

public DataWriterListener
  get_listener
    (void);

public Topic
  get_topic
    (void);

public Publisher

```

```

        get_publisher
            (void);

    public int
        wait_for_acknowledgments
            (Duration_t max_wait);

    public int
        get_liveliness_lost_status
            (LivelinessLostStatusHolder status);

    public int
        get_offered_deadline_missed_status
            (OfferedDeadlineMissedStatusHolder status);

    public int
        get_offered_incompatible_qos_status
            (PublicationMatchedStatusHolder status);

    public int
        get_publication_matched_status
            (PublicationMatchedStatusHolder status);

    public int
        assert_liveliness
            (void);

    public int
        get_matched_subscriptions
            (InstanceHandleSeqHolder subscription_handles);

    public int
        get_matched_subscription_data
            (SubscriptionBuiltinTopicDataHolder subscription_data,
             long subscription_handle);
};

```

The following 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

```
import DDS.*;
public int
    assert_liveliness
        (void);
```

## Description

This operation asserts the liveliness for the DataWriter.

## Parameters

<none>

## Return Value

*int* - 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 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 61 for more information on `LivelinessQosPolicy`.

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

The liveliness should be asserted by the application, depending on the `LivelinessQosPolicy`. Asserting the liveliness for this DataWriter can also be achieved by asserting the liveliness to the `DomainParticipant`.

### Return Code

When the operation returns:

- `RETCODE_OK` - the liveliness 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

```
import DDS.*;
public int
    dispose
        (<data> instance_data,
         long 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

```
import DDS.*;
public int
    dispose_w_timestamp
        (<data> instance_data,
         long instance_handle,
         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

```
import DDS.*;
public int
    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

```
import DDS.*;
public int
    get_key_value
        (<data>Holder key_holder,
         long handle);
```

### 3.4.2.7 `get_listener`

#### Scope

```
DDS.DataWriter
```

#### Synopsis

```
import DDS.*;
public DataWriterListener
    get_listener
        (void);
```

#### Description

This operation allows access to a `DataWriterListener`.

#### Parameters

<none>

#### Return Value

*DataWriterListener* - result is a reference 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 reference is returned.

### 3.4.2.8 `get_liveliness_lost_status`

#### Scope

`DDS.DataWriter`

#### Synopsis

```
import DDS.*;
public int
    get_liveliness_lost_status
        (LivelinessLostStatusHolder status);
```

#### Description

This operation obtains a `LivelinessLostStatus` object of the `DataWriter`.

#### Parameters

*inout* `LivelinessLostStatusHolder status` - the contents of the `LivelinessLostStatus` object of the `DataWriter` will be copied into the `LivelinessLostStatusHolder` specified by `status`.

#### Return Value

*int* - 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` object of the `DataWriter`. This object 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

```
import DDS.*;
public int
    get_matched_subscription_data
        (SubscriptionBuiltinTopicDataHolder subscription_data,
         long subscription_handle);
```

#### Description

This operation retrieves information on the specified subscription that is currently “associated” with the `DataWriter`.

#### Parameters

*inout SubscriptionBuiltinTopicDataHolder subscription\_data* - a Holder for the sample in which the information about the specified subscription is to be stored.

*in long subscription\_handle* - a handle to the subscription whose information needs to be retrieved.

#### Return Value

*int* - Possible return codes of the operation are: *RETCODE\_OK*, *RETCODE\_ERROR*, *RETCODE\_UNSUPPORTED*, *RETCODE\_ALREADY\_DELETED*, *RETCODE\_OUT\_OF\_RESOURCES* or *RETCODE\_NOT\_ENABLED*.

#### Detailed Description

This operation retrieves information on the specified subscription that is currently “associated” with the `DataWriter`. That is, a subscription with a matching Topic and compatible QoS that the application has not indicated should be “ignored” by means of the `ignore_subscription` operation on the `DomainParticipant` class.

The `subscription_handle` must correspond to a subscription currently associated with the `DataWriter`, otherwise the operation will fail and return *RETCODE\_BAD\_PARAMETER*. The operation `get_matched_subscriptions` can be used to find the subscriptions that are currently matched with the `DataWriter`.

The operation may also fail if the infrastructure does not hold the information necessary to fill in the `subscription_data`. This is the case when OpenSplice is configured not to maintain discovery information in the Networking Service. (See the description for the `NetworkingService/Discovery/enabled` property in the Deployment Manual for more information about this subject.) In such cases the operation will return `RETCODE_UNSUPPORTED`.

### Return Code

When the operation returns:

- `RETCODE_OK` - the information on the specified subscription has successfully been retrieved.
- `RETCODE_ERROR` - an internal error has occurred.
- `RETCODE_UNSUPPORTED` - OpenSplice is configured not to maintain the information about “associated” subscriptions.
- `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.

### 3.4.2.10 `get_matched_subscriptions`

#### Scope

`DDS.DataWriter`

#### Synopsis

```
import DDS.*;
public int
    get_matched_subscriptions
        (InstanceHandleSeqHolder subscription_handles);
```

#### Description

This operation retrieves the list of subscriptions currently "associated" with the DataWriter.

#### Parameters

*inout InstanceHandleSeqHolder subscription\_handles* - a Holder for an array which is used to pass the list of all associated subscriptions.



## Return Value

*int* - Possible return codes of the operation are: `RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_UNSUPPORTED`, `RETCODE_ALREADY_DELETED`, `DDS_RETCODE_OUT_OF_RESOURCES` or `DDS_RETCODE_NOT_ENABLED`.

## Detailed Description

This operation retrieves the list of subscriptions currently "associated" with the `DataWriter`. That is, subscriptions that have a matching Topic and compatible QoS that the application has not indicated should be "ignored" by means of the `ignore_subscription` operation on the `DomainParticipant` class.

The array inside the `subscription_handles` Holder may be pre-allocated by the application and can be re-used in a subsequent invocation of the `get_matched_subscriptions` operation. If the pre-allocated array is not big enough to hold the number of associated subscriptions, the array will automatically be (re-)allocated to fit the required size.

The handles returned in the `subscription_handles` array are the ones that are used by the DDS implementation to locally identify the corresponding matched subscription entities. You can access more detailed information about a particular subscription by passing its `subscription_handle` to either the `get_matched_subscription_data` operation or to the `read_instance` operation on the built-in reader for the "DCPSSubscription" topic.



Be aware that since an instance handle is an opaque datatype, it does not necessarily mean that the handles obtained from the `get_matched_subscriptions` operation have the same value as the ones that appear in the `instance_handle` field of the `SampleInfo` when retrieving the subscription info through corresponding "DCPSSubscriptions" built-in reader. You can't just compare two handles to determine whether they represent the same subscription. If you want to know whether two handles actually do represent the same subscription, use both handles to retrieve their corresponding `SubscriptionBuiltinTopicData` samples and then compare the `key` field of both samples.

The operation may fail if the infrastructure does not locally maintain the connectivity information. This is the case when `OpenSplice` is configured not to maintain discovery information in the `NetworkingService`. (See the description for the `NetworkingService/Discovery/enabled` property in the `Deployment Manual` for more information about this subject.) In such cases the operation will return `RETCODE_UNSUPPORTED`.

## Return Code

When the operation returns:

- `RETCODE_OK` - the list of associated subscriptions has successfully been obtained.

- *RETCODE\_ERROR* - an internal error has occurred.
- *RETCODE\_UNSUPPORTED* - OpenSplice is configured not to maintain the information about “associated” subscriptions.
- *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.

### 3.4.2.11 **get\_offered\_deadline\_missed\_status**

#### Scope

DDS.DataWriter

#### Synopsis

```
import DDS.*;
public int
    get_offered_deadline_missed_status
        (OfferedDeadlineMissedStatusHolder status);
```

#### Description

This operation obtains the *OfferedDeadlineMissedStatus* object of the *DataWriter*.

#### Parameters

*inout OfferedDeadlineMissedStatusHolder status* - the contents of the *OfferedDeadlineMissedStatus* object of the *DataWriter* will be copied into the *OfferedDeadlineMissedStatusHolder* specified by *status*.

#### Return Value

*int* - 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* object of the *DataWriter*. This object 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**

```
import DDS.*;
public int
    get_offered_incompatible_qos_status
        (OfferedIncompatibleQosStatusHolder status);
```

#### **Description**

This operation obtains the OfferedIncompatibleQosStatus object.

#### **Parameters**

*inout OfferedIncompatibleQosStatusHolder status* - the contents of the OfferedIncompatibleQosStatus object of the DataWriter will be copied into the OfferedIncompatibleQosStatusHolder specified by status.

#### **Return Value**

*int* - 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 object of the DataWriter. This object 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

```
import DDS.*;
public int
    get_publication_match_status
        (PublicationMatchedStatusHolder status);
```

#### Description

This operation obtains the `PublicationMatchedStatus` object of the `DataWriter`.

#### Parameters

*inout* `PublicationMatchedStatusHolder status` - the contents of the `PublicationMatchedStatus` object of the `DataWriter` will be copied into the `PublicationMatchedStatusHolder` specified by `status`.

#### Return Value

*int* - Possible return codes of the operation are: `RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_UNSUPPORTED`, `DDS_RETCODE_ALREADY_DELETED` or `DDS_RETCODE_OUT_OF_RESOURCES`.

## Detailed Description

This operation obtains the `PublicationMatchedStatus` object of the `DataWriter`. This object contains the information whether a new match has been discovered for the current publication, or whether an existing match has ceased to exist.

This means that the status represents that either a `DataReader` object has been discovered by the `DataWriter` with the same Topic and a compatible Qos, or that a previously discovered `DataReader` has ceased to be matched to the current `DataWriter`. A `DataReader` may cease to match when it gets deleted, when it changes its Qos to a value that is incompatible with the current `DataWriter` or when either the `DataWriter` or the `DataReader` has chosen to put its matching counterpart on its ignore-list using the `ignore_subscription` or `ignore_publication` operations on the `DomainParticipant`.

The operation may fail if the infrastructure does not hold the information necessary to fill in the `PublicationMatchedStatus`. This is the case when `OpenSplice` is configured not to maintain discovery information in the Networking Service. (See the description for the `NetworkingService/Discovery/enabled` property in the Deployment Manual for more information about this subject.) In this case the operation will return `RETCODE_UNSUPPORTED`.

The `PublicationMatchedStatus` can also be monitored using a `DataWriterListener` or by using the associated `StatusCondition`.

### Return Code

When the operation returns:

- `RETCODE_OK` - the current `PublicationMatchedStatus` of this `DataWriter` has successfully been copied into the specified status parameter.
- `RETCODE_ERROR` - an internal error has occurred.
- `RETCODE_UNSUPPORTED` - `OpenSplice` is configured not to maintain the information about “associated” subscriptions.
- `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.

### 3.4.2.14 `get_publisher`

#### Scope

`DDS.DataWriter`

#### Synopsis

```
import DDS.*;
```

```
public Publisher
get_publisher
(void);
```

### Description

This operation returns the Publisher to which the DataWriter belongs.

### Parameters

<none>

### Return Value

*Publisher* - Return value is a reference to the Publisher object 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 reference is returned.

#### 3.4.2.15 get\_qos

### Scope

```
DDS.DataWriter
```

### Synopsis

```
import DDS.*;
public int
get_qos
(DataWriterQosHolder qos);
```

### Description

This operation allows access to the existing list of QosPolicy settings for a DataWriter.

### Parameters

*inout DataWriterQosHolder qos* - the destination PublisherQosHolder object in which the QosPolicy settings will be copied.

### Return Value

*int* - 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 referenced by the `qos` parameter.

### Return Code

When the operation returns:

- `RETCODE_OK` - the existing set of `QosPolicy` values applied to this `DataWriter` has successfully been copied into the specified `DataWriterQosHolder` 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

```
import DDS.*;
public int
    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

```
import DDS.*;
public StatusCondition
    get_statuscondition
        (void);
```

### 3.4.2.18 `get_topic`

#### Scope

```
DDS.DataWriter
```

#### Synopsis

```
import DDS.*;
```

```
public Topic
    get_topic
        (void);
```

### Description

This operation returns the `Topic` object which is associated with the `DataWriter`.

### Parameters

<none>

### Return Value

*Topic* - Return value is a reference 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 reference 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

```
import DDS.*;
public long
    lookup_instance
        (<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

```
import DDS.*;
public long
    register_instance
```



```
(<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

```
import DDS.*;
public long
    register_instance_w_timestamp
        (<data> instance_data,
         Time_t source_timestamp);
```

### 3.4.2.22 set\_listener

#### Scope

```
DDS.DataWriter
```

#### Synopsis

```
import DDS.*;
public int
    set_listener
        (DataWriterListener a_listener,
         int mask);
```

#### Description

This operation attaches a `DataWriterListener` to the `DataWriter`.

#### Parameters

*in DataWriterListener a\_listener* - a reference to the `DataWriterListener` instance, which will be attached to the `DataWriter`.

*in int mask* - a bit mask in which each bit enables the invocation of the `DataWriterListener` for a certain status.

#### Return Value

*int* - Possible return codes of the operation are: `RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_UNSUPPORTED`, `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 reference, it represents a listener that is treated as a NOOP<sup>1</sup> for all statuses activated in the bitmask.

### 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`.



Be aware that the `PUBLICATION_MATCHED_STATUS` is not applicable when the infrastructure does not have the information available to determine connectivity. This is the case when OpenSplice is configured not to maintain discovery information in the Networking Service. (See the description for the `NetworkingService/Discovery/enabled` property in the Deployment Manual for more information about this subject.) In this case the operation will return `RETCODE_UNSUPPORTED`.

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 applicable statuses specified in the “Data Distribution Service for Real-time Systems Version 1.2” specification.

---

1. Short for **No-Operation**, an instruction that performs nothing at all.

### 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.

### Return Code

When the operation returns:

- `RETCODE_OK` - the `DataWriterListener` is attached
- `RETCODE_ERROR` - an internal error has occurred
- `RETCODE_UNSUPPORTED` - a status was selected that cannot be supported because the infrastructure does not maintain the required connectivity information.
- `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

```
import DDS.*;
public int
    set_qos
        (DataWriterQos qos);
```

### Description

This operation replaces the existing set of `QosPolicy` settings for a `DataWriter`.

### Parameters

*in* `DataWriterQos qos` - the new set of `QosPolicy` settings for the `DataWriter`.

## Return Value

*int* - 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 object 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` setting (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.
- `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

```
import DDS.*;
public int
    unregister_instance
        (<data> instance_data,
         long 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

```
import DDS.*;
public int
    unregister_instance_w_timestamp
        (<data> instance_data,
         long handle,
         Time_t source_timestamp);
```

### 3.4.2.26 wait\_for\_acknowledgments

#### Synopsis

```
import DDS.*;
public int
    wait_for_acknowledgments
        (Duration_t max_wait);
```

#### Description

This operation blocks the calling thread until either all data written by the `DataWriter` is acknowledged by the local infrastructure, or until the duration specified by `max_wait` parameter elapses, whichever happens first.

## Parameters

*in 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

*int* - Possible return codes of the operation are: `RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_ALREADY_DELETED`, `DDS_RETCODE_OUT_OF_RESOURCES`, `DDS_RETCODE_NOT_ENABLED` or `DDS_RETCODE_TIMEOUT`.

## Detailed Description

This operation blocks the calling thread until either all data written by the `DataWriter` is acknowledged by the local infrastructure, or until the duration specified by `max_wait` parameter elapses, whichever happens first.

Data is acknowledged by the local infrastructure when it does not need to be stored in its `Writer`'s local history. When a locally-connected subscription (including the networking service) has no more resources to store incoming samples it will start to reject these samples, resulting in its source `DataWriter` to store them temporarily in its own local history to be retransmitted at a later moment in time. In such scenarios, the `wait_for_acknowledgments` operation will block until the `DataWriter` has retransmitted its entire history, which is therefore effectively empty, or until the `max_wait` timeout expires, whichever happens first. In the first case the operation will return `RETCODE_OK`, in the latter it will return `RETCODE_TIMEOUT`.



Be aware that in case the operation returns `RETCODE_OK`, the data has only been acknowledged by the local infrastructure: it does not mean all remote subscriptions have already received the data. However, delivering the data to remote nodes is then the sole responsibility of the networking service: even when the publishing application would terminate, all data that has not yet been received may be considered 'on-route' and will therefore eventually arrive (unless the networking service itself will crash). In contrast, if the `DataWriter` would still have data in its local history buffer when it terminates, this data is considered 'lost'.

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 `DDS_RETCODE_OK`, since best-effort `DataWriters` will never store rejected samples in their local history: they will just drop them and continue business as usual.

*Return Code*

When the operation returns:

- *RETCODE\_OK* - the data of the `DataWriter` has been acknowledged by the local infrastructure.
- *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.
- *RETCODE\_TIMEOUT* - not all data is acknowledged before `max_wait` elapsed.

**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**

```
import DDS.*;
public int
    write
        (<data> instance_data,
         long 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**

```
import DDS.*;
public int
    write_w_timestamp
        (<data> instance_data,
         int handle,
         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

```
import DDS.*;
public int
    writedispose
        (<data> instance_data,
         long 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

```
import DDS.*;
public int
    writedispose
        (<data> instance_data,
         long handle,
         Time_t source_timestamp);
```

### 3.4.2.31 Interface FooDataWriter

The pre-processor generates from IDL type descriptions the application `<type>DataWriter` interfaces. For each application data type that is used as `Topic` data type, a typed interface `<type>DataWriter` is extending the `DataWriter` interface. In this paragraph, the interface `FooDataWriter` in the package `SPACE` describes the operations of these extending `<type>DataWriter` interface as an example for the fictional application type `Foo` (defined in the module `SPACE`).

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 is as follows:

```

public interface FooDataWriter
{
//
// extends interface Entity
//
// public StatusCondition
//   get_statuscondition
//   (void);
// public int
//   get_status_changes
//   (void);
// public int
//   enable
//   (void);
//
// inherited from class DataWriter
//
// public int
//   set_qos
//   (DataWriterQos qos);

// public int
//   get_qos
//   (DataWriterQosHolder qos);

// public int
//   set_listener
//   (DataWriterListener a_listener,
//   int mask);

// public DataWriterListener
//   get_listener
//   (void);

// public Topic
//   get_topic
//   (void);

// public Publisher
//   get_publisher
//   (void);

// public int
//   wait_for_acknowledgments
//   (Duration_t max_wait);

// public int
//   get_liveliness_lost_status

```

```

//      (LivelinessLostStatusHolder status);

// public int
//      get_offered_deadline_missed_status
//      (OfferedDeadlineMissedStatusHolder status);

// public int
//      get_offered_incompatible_qos_status
//      (OfferedIncompatibleQosStatusHolder status);

// public int
//      get_publication_matched_status
//      (PublicationMatchedStatusHolder status);

// public int
//      assert_liveliness
//      (void);

// public int
//      get_matched_subscriptions
//      (InstanceHandleSeqHolder subscription_handles);

// public int
//      get_matched_subscription_data
//      (SubscriptionBuiltinTopicDataHolder subscription_data,
//       long subscription_handle);
//
// implemented API operations
//
public long
    register_instance
    (Foo instance_data);
public long
    register_instance_w_timestamp
    (Foo instance_data,
     Time_t source_timestamp);
public int
    unregister_instance
    (Foo instance_data,
     long handle);
public int
    unregister_instance_w_timestamp
    (Foo instance_data,
     long handle,
     Time_t source_timestamp);
public int
    write
    (Foo instance_data,
     long handle);
public int

```

```

        write_w_timestamp
            (Foo instance_data,
             long handle,
             Time_t source_timestamp);
public int
    dispose
        (Foo instance_data,
         long instance_handle);
public int
    dispose_w_timestamp
        (Foo instance_data,
         long instance_handle,
         Time_t source_timestamp);
public int
    writediscard
        (Foo instance_data,
         long instance_handle);
public int
    writediscard_w_timestamp
        (Foo instance_data,
         long instance_handle,
         Time_t source_timestamp);
public int
    get_key_value
        (FooHolder key_holder,
         long handle);
public long
    lookup_instance
        (Foo instance_data);
};

```

The following 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 interface `DataWriter` for further explanation.

#### Synopsis

```

public int
    assert_liveliness
        (void);

```

### 3.4.2.33 dispose

#### Scope

SPACE.FooDataWriter

#### Synopsis

```
public int
dispose
(Foo instance_data,
 long instance_handle);
```

#### Description

This operation requests the Data Distribution Service to mark the instance for deletion.

#### Parameters

*in Foo instance\_data* - the actual instance to be disposed of.

*in long instance\_handle* - the handle to the instance to be disposed of.

#### Return Value

*int* - 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 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 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 73.

#### 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 52.

### Instance Handle

The `HANDLE_NIL` constant 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

Since the sample that is passed as `instance_data` is merely used 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 Java language mapping, where:

- 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, 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

### 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

```
import DDS.*;
public int
    dispose_w_timestamp
        (Foo instance_data,
         long instance_handle,
         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 Foo instance\_data* - the actual instance to be disposed of.

*in long instance\_handle* - the handle to the instance to be disposed of.

*in Time\_t source\_timestamp* - the timestamp which is provided for the *DataReader*.

## Return Value

*int* - 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 `DDS_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 interface `Entity` for further explanation.

## Synopsis

```
int
    enable
    (void);
```



### 3.4.2.36 `get_key_value`

#### Scope

`SPACE.FooDataWriter`

#### Synopsis

```
int
get_key_value
(FooHolder key_holder,
 long handle);
```

#### Description

This operation retrieves the key value of a specific instance.

#### Parameters

*inout FooHolder key\_holder* - a reference to the sample in which the key values are stored.

*in long handle* - the handle to the instance from which to get the key value.

#### Return Value

*int* - 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 an `HANDLE NIL` constant 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 that 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 interface `DataWriter` for further explanation.

#### Synopsis

```
import DDS.*;
public DataWriterListener
    get_listener
        (void);
```

### 3.4.2.38 `get_liveliness_lost_status` (inherited)

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

#### Synopsis

```
import DDS.*;
public int
    get_liveliness_lost_status
        (LivelinessLostStatusHolder status);
```

### 3.4.2.39 `get_matched_subscription_data` (inherited)

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

#### Synopsis

```
import DDS.*;
public int
    get_matched_subscription_data
        (SubscriptionBuiltinTopicData subscription_data,
         long subscription_handle);
```

### 3.4.2.40 `get_matched_subscriptions` (inherited)

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

#### Synopsis

```
import DDS.*;
```

```
public int
    get_matched_subscriptions
        (InstanceHandleSeqHolder subscription_handles);
```

#### 3.4.2.41 **get\_offered\_deadline\_missed\_status (inherited)**

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

##### **Synopsis**

```
import DDS.*;
public int
    get_offered_deadline_missed_status
        (OfferedDeadlineMissedStatusHolder status);
```

#### 3.4.2.42 **get\_offered\_incompatible\_qos\_status (inherited)**

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

##### **Synopsis**

```
import DDS.*;
public int
    get_offered_incompatible_qos_status
        (OfferedIncompatibleQosStatusHolder status);
```

#### 3.4.2.43 **get\_publication\_matched\_status (inherited)**

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

##### **Synopsis**

```
import DDS.*;
public int
    get_publication_matched_status
        (PublicationMatchedStatusHolder status);
```

#### 3.4.2.44 **get\_publisher (inherited)**

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

##### **Synopsis**

```
import DDS.*;
public Publisher
    get_publisher
        (void);
```

#### 3.4.2.45 **get\_qos (inherited)**

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

##### **Synopsis**

```
import DDS.*;
public int
    get_qos
        (DataWriterQosHolder qos);
```

#### 3.4.2.46 **get\_status\_changes (inherited)**

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

##### **Synopsis**

```
import DDS.*;
public int
    get_status_changes
        (void);
```

#### 3.4.2.47 **get\_statuscondition (inherited)**

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

##### **Synopsis**

```
import DDS.*;
public StatusCondition
    get_statuscondition
        (void);
```

#### 3.4.2.48 **get\_topic (inherited)**

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

##### **Synopsis**

```
import DDS.*;
public Topic
    get_topic
        (void);
```

#### 3.4.2.49 **lookup\_instance**

##### **Scope**

```
SPACE.FooDataWriter
```

## Synopsis

```
import DDS.*;
public long
    lookup_instance
        (Foo instance_data);
```

## Description

This operation returns the value of the instance handle which corresponds to the `instance_data`.

## Parameters

*in Foo instance\_data* - a reference to the instance for which the corresponding instance handle needs to be looked up.

## Return Value

*long* - 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 timestamped 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

Since the sample that is passed as `instance_data` is merely used to determine the identity based on the uniqueness of its key values, only the keyfields will be validated against the restrictions imposed by the IDL to Java language mapping, where:

- 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, 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.

### 3.4.2.50 `register_instance`

#### Scope

`SPACE.FooDataWriter`

#### Synopsis

```
long
register_instance
(Foo instance_data);
```

#### Description

This operation informs the Data Distribution Service that the application will be modifying a particular instance.

#### Parameters

*in Foo instance\_data* - the instance, which the application writes to or disposes of.

#### Return Value

*long* - 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` constant 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` 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`,

writedispose and writedispose\_w\_timestamp operations and specify a `HANDLE_NIL` 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

Since the sample that is passed as `instance_data` is merely used to determine the identity based on the uniqueness of its key values, only the keyfields will be validated against the restrictions imposed by the IDL to Java language mapping, where:

- 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, 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.

### 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

```
import DDS.*;
public long
    register_instance_w_timestamp
        (Foo instance_data,
         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 Time\_t source\_timestamp* - the timestamp used.

#### Return Value

*long* - Result value is the handle to the *Instance*, which must be used for writing and disposing. In case of an error, a `HANDLE_NIL` reference 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 interface `DataWriter` for further explanation.

#### Synopsis

```
import DDS.*;
```



```
public int
    set_listener
        (DataWriterListener a_listener,
         int mask);
```

### 3.4.2.53 set\_qos (inherited)

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

#### Synopsis

```
import DDS.*;
public int
    set_qos
        (DataWriterQos qos);
```

### 3.4.2.54 unregister\_instance

#### Scope

```
SPACE.FooDataWriter
```

#### Synopsis

```
import DDS.*;
public int
    unregister_instance
        (Foo instance_data,
         long handle);
```

#### Description

This operation informs the Data Distribution Service that the application will **not** be modifying a particular instance any more.

#### Parameters

*in Foo instance\_data* - the instance to which the application was writing or disposing.

*in long handle* - the handle to the Instance, which has been used for writing and disposing.

#### Return Value

*int* - 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 constant `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 492.

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 special constant `HANDLE_NIL` 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 the special constant `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 `HANDLE_NIL`.

### Sample Validation

Since the sample that is passed as `instance_data` is merely used 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 Java language mapping, where:

- 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, 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.

### 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

```
import DDS.*;
public int
    unregister_instance_w_timestamp
        (Foo instance_data,
         long handle,
         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 long handle* - the handle to the Instance, which has been used for writing and disposing.

*in Time\_t source\_timestamp* - the timestamp used.

## Return Value

*int* - 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` - the current action overflowed the available resources as specified by the combination of the `ReliabilityQosPolicy`, `HistoryQosPolicy` and `ResourceLimitsQosPolicy`. This caused blocking of the `unregister_instance_w_timestamp` operation, which could not be resolved before `max_blocking_time` of the `ReliabilityQosPolicy` elapsed.
- `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 wait\_for\_acknowledgments (inherited)

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

#### Synopsis

```
import DDS.*;
public int
    wait_for_acknowledgments
        (const Duration_t& max_wait);
```

### 3.4.2.57 write

#### Scope

```
SPACE.FooDataWriter
```

#### Synopsis

```
import DDS.*;
public int
    write
        (Foo instance_data,
         long handle);
```

#### Description

This operation modifies the value of a data instance.

#### Parameters

*in Foo instance\_data* - the data to be written.

*in long handle* - the handle to the instance as supplied by `register_instance`.

#### Return Value

*int* - 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 constant `HANDLE_NIL`, which means that the identity of the instance is automatically deduced from the `instance_data` (identified by the key).

#### Instance Handle

The special constant `HANDLE_NIL` 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 the special constant `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

Before the sample is accepted by the `DataWriter`, it will be validated against the restrictions imposed by the IDL to Java language mapping, where:

- 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
- the length of an array must exactly match the size specified in IDL

If any of these restrictions is violated, 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.

### 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.

#### 3.4.2.58 write\_w\_timestamp

##### Scope

`SPACE.FooDataWriter`

##### Synopsis

```
import DDS.*;
public int
    write_w_timestamp
        (Foo instance_data,
         long handle,
         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 Foo instance\_data* - the data to be written.

*in long handle* - the handle to the instance as supplied by `register_instance`.

*in Time\_t source\_timestamp* - the timestamp used.

## Return Value

*int* - 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 `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.

### 3.4.2.59 writedispose

#### Scope

`SPACE.FooDataWriter`

#### Synopsis

```
import DDS.*;
public int
    writedispose
        (Foo instance_data,
         long handle);
```

#### Description

This operation modifies and disposes a data instance.

#### Parameters

*in Foo instance\_data* - the data to be written and disposed.

*in long handle* - the handle to the instance as supplied by `register_instance`.

#### Return Value

*int* - 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 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 73.

#### 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 52.

#### 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

Before the sample is accepted by the `DataWriter`, it is validated against the restrictions imposed by the IDL to Java language mapping, where:

- 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

- the length of an array must exactly match the size specified in IDL

If any of these restrictions is violated, 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.

### 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` - the current action overflowed the available resources as specified by the combination of the `ReliabilityQosPolicy`, `HistoryQosPolicy` and `ResourceLimitsQosPolicy`. This caused blocking of the `writediscard` operation, which could not be resolved before `max_blocking_time` of the `ReliabilityQosPolicy` elapsed.

## 3.4.2.60 writediscard\_w\_timestamp

### Scope

`SPACE.FooDataWriter`

### Synopsis

```
import DDS.*;
public int
    writediscard_w_timestamp
        (Foo instance_data,
         long handle,
         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 Foo instance\_data* - the data to be written and disposed.

*in long handle* - the handle to the instance as supplied by `register_instance`.

*in Time\_t source\_timestamp* - the timestamp used.

## Return Value

*int* - 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 is as follows:

```
public interface PublisherListener {
//
// extends interface DataWriterListener
//
// public void
//     on_offered_deadline_missed
//         (DataWriter writer,
//          OfferedDeadlineMissedStatus status);

// public void
//     on_offered_incompatible_qos
//         (DataWriter writer,
//          OfferedIncompatibleQosStatus status);

// public void
//     on_liveliness_lost
//         (DataWriter writer,
//          LivelinessLostStatus status);

// public void
//     on_publication_matched
//         (DataWriter writer,
//          PublicationMatchedStatus status);
//
// implemented API operations
//     <no operations>
//
};
```

The following 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 interface `DataWriterListener` for further explanation.

##### Synopsis

```
import DDS.*;
public void
    on_liveliness_lost
        (DataWriter writer,
         LivelinessLostStatus status);
```

#### 3.4.3.2 `on_offered_deadline_missed` (inherited, abstract)

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

##### Synopsis

```
import DDS.*;
public void
    on_offered_deadline_missed
        (DataWriter writer,
         OfferedDeadlineMissedStatus status);
```

#### 3.4.3.3 `on_offered_incompatible_qos` (inherited, abstract)

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

##### Synopsis

```
import DDS.*;
public void
    on_offered_incompatible_qos
        (DataWriter writer,
         OfferedIncompatibleQosStatus status);
```

#### 3.4.3.4 `on_publication_matched` (inherited, abstract)

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

##### Synopsis

```
import DDS.*;
```



```

public void
    on_publication_matched
        (DataWriter writer,
         PublicationMatchedStatus status);

```

### 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 implement the `DataWriterListener` interface. **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 is as follows:

```

public interface DataWriterListener
{
    //
    // operations
    //
    public void
        on_offered_deadline_missed
            (DataWriter writer,
             OfferedDeadlineMissedStatus status);

    public void
        on_offered_incompatible_qos
            (DataWriter writer,
             OfferedIncompatibleQosStatus status);

    public void
        on_liveliness_lost
            (DataWriter writer,
             LivelinessLostStatus status);

    public void
        on_publication_matched
            (DataWriter writer,

```

```

        PublicationMatchedStatus status);
//
// implemented API operations
// <no operations>
//
};

```

The `DataWriterListener` abstract operations are fully described because they must be implemented by the application.

### 3.4.4.1 on\_liveliness\_lost

#### Scope

```
DDS.DataWriterListener
```

#### Synopsis

```

import DDS.*;
public void
    on_liveliness_lost
        (DataWriter writer,
         LivelinessLostStatus status);

```

#### Description

This operation must be implemented by the application and is called by the Data Distribution Service when the `LivelinessLostStatus` changes.

#### Parameters

*in DataWriter writer* - contains a reference to the `DataWriter` on which the `LivelinessLostStatus` has changed (this is an input to the application).

*in LivelinessLostStatus status* - contain the `LivelinessLostStatus` object (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` object.

### 3.4.4.2 `on_offered_deadline_missed`

#### Scope

`DDS.DataWriterListener`

#### Synopsis

```
import DDS.*;
public void
    on_offered_deadline_missed
        (DataWriter writer,
         OfferedDeadlineMissedStatus status);
```

#### Description

This operation must be implemented by the application and is called by the Data Distribution Service when the `OfferedDeadlineMissedStatus` changes.

#### Parameters

*in DataWriter writer* - contain a reference to the `DataWriter` on which the `OfferedDeadlineMissedStatus` has changed (this is an input to the application).

*in OfferedDeadlineMissedStatus status* - contain the `OfferedDeadlineMissedStatus` object (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` object.

### 3.4.4.3 on\_offered\_incompatible\_qos

#### Scope

`DDS.DataWriterListener`

#### Synopsis

```
import DDS.*;
public void
    on_offered_incompatible_qos
        (DataWriter writer,
         OfferedIncompatibleQosStatus status);
```

#### 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 writer* - contain a reference to the `DataWriter` on which the `OFFERED_INCOMPATIBLE_QOS_STATUS` has changed (this is an input to the application).

*in OfferedIncompatibleQosStatus status* - contain the `OfferedIncompatibleQosStatus` object (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` object.

#### 3.4.4.4 on\_publication\_matched

##### Scope

```
DDS.DataWriterListener
```

##### Synopsis

```
import DDS.*;
public void
    on_publication_matched
        (DataWriter writer,
         PublicationMatchedStatus status);
```

##### Description

This operation must be implemented by the application and is called by the Data Distribution Service when a new match has been discovered for the current publication, or when an existing match has ceased to exist.

##### Parameters

*in DataWriter writer* - contains a pointer to the `DataWriter` for which a match has been discovered (this is an input to the application provided by the Data Distribution Service).

*in PublicationMatchedStatus status* - contains the `PublicationMatchedStatus` object (this is an input to the application provided by the Data Distribution Service).

##### Return Value

<none>

##### Detailed Description

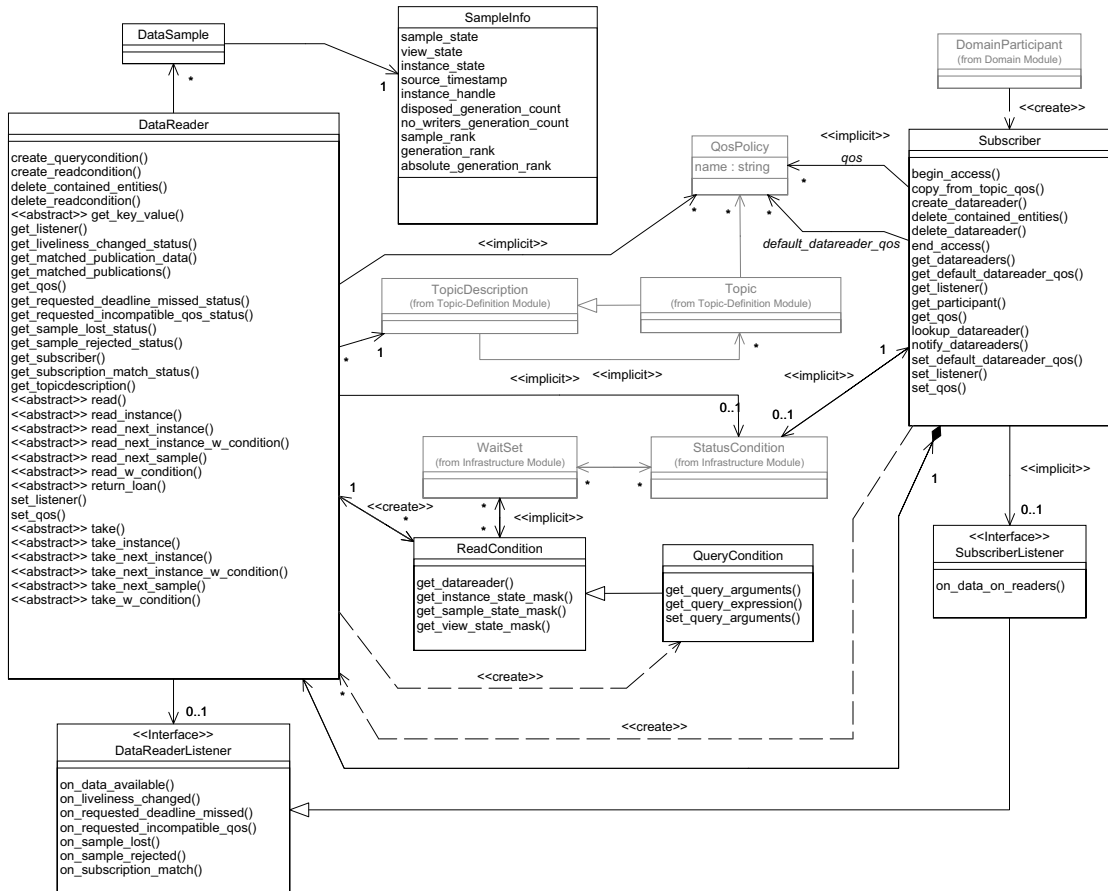
This operation must be implemented by the application and is called by the Data Distribution Service when a new match has been discovered for the current publication, or when an existing match has ceased to exist. Usually this means that a new `DataReader` that matches the `Topic` and that has compatible `Qos` as the current `DataWriter` has either been discovered, or that a previously discovered

`DataReader` has ceased to be matched to the current `DataWriter`. A `DataReader` may cease to match when it gets deleted, when it changes its Qos to a value that is incompatible with the current `DataWriter` or when either the `DataWriter` or the `DataReader` has chosen to put its matching counterpart on its ignore-list using the `ignore_subscription` or `ignore_publication` operations on the `DomainParticipant`.

The implementation of this Listener operation may be left empty when this functionality is not needed: it will only be called when the relevant `DataWriterListener` is installed and enabled for the `PUBLICATION_MATCHED_STATUS`.

The Data Distribution Service will provide a reference to the `DataWriter` in the parameter `writer` and the `PublicationMatchedStatus` object in the parameter `status` for use by the application.

### 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 (class)
- 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 Interface 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:

```
public interface Subscriber
{
//
// extends interface Entity
//
// public StatusCondition
//   get_statuscondition
//   (void);
// public int
//   get_status_changes
//   (void);
// public int
//   enable
//   (void);
//
// implemented API operations
//
public DataReader
```



```
        create_datareader
            (TopicDescription a_topic,
             DataReaderQos qos,
             DataReaderListener a_listener,
             int mask);

    public int
        delete_datareader
            (DataReader a_datareader);

    public int
        delete_contained_entities
            (void);

    public DataReader
        lookup_datareader
            (String topic_name);

    public int
        get_datareaders
            (DataReaderSeqHolder readers,
             int sample_states,
             int view_states,
             int instance_states);

    public int
        notify_datareaders
            (void);

    public int
        set_qos
            (SubscriberQos qos);

    public int
        get_qos
            (SubscriberQosHolder qos);

    public int
        set_listener
            (SubscriberListener a_listener,
             int mask);

    public SubscriberListener
        get_listener
            (void);

    public int
        begin_access
            (void);

    public int
```

```

        end_access
            (void);

    public DomainParticipant
        get_participant
            (void);

    public int
        set_default_datareader_qos
            (DataReaderQos qos);

    public int
        get_default_datareader_qos
            (DataReaderQosHolder qos);

    public int
        copy_from_topic_qos
            (DataReaderQosHolder a_datareader_qos,
             TopicQos a_topic_qos);
};

```

The following paragraphs describe how all of the `Subscriber` operations are used. 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

```

import DDS.*;
public int
    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

```

import DDS.*;
public int
    copy_from_topic_qos

```

```
(DataReaderQosHolder a_datareader_qos,
 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 DataReaderQosHolder a\_datareader\_qos* - the destination `DataReaderQos` object to which the `QosPolicy` settings will be copied.

*in TopicQos a\_topic\_qos* - the source `TopicQos`, which will be copied.

## Return Value

*int* - 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

```
import DDS.*;
public DataReader
    create_datareader
        (TopicDescription a_topic,
         DataReaderQos qos,
         DataReaderListener a_listener,
         int 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 a\_topic* - a reference to the `TopicDescription` for which the `DataReader` is created. This may be a `Topic`, `MultiTopic` or `ContentFilteredTopic`.

*in DataReaderQos qos* - the object with the `QosPolicy` settings for the new `DataReader`, when these `QosPolicy` settings are not self consistent, no `DataReader` is created.

*in DataReaderListener a\_listener* - a reference 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 int mask* - a bit-mask in which each bit enables the invocation of the `DataReaderListener` for a certain status.

#### Return Value

*DataReader* - Return value is a reference to the newly created `DataReader`. In case of an error, the `null` reference 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 reference 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 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.

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`.



Be aware that the `SUBSCRIPTION_MATCHED_STATUS` is not applicable when the infrastructure does not have the information available to determine connectivity. This is the case when `OpenSplice` is configured not to maintain discovery information in the `Networking Service`. (See the description for the `NetworkingService/Discovery/enabled` property in the `Deployment Manual` for more information about this subject.) In this case the operation will return `null`.

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 applicable 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

```
import DDS.*;
public int
    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

*int* - 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

```
import DDS.*;
public int
    delete_datareader
        (DataReader a_datareader);
```

### Description

This operation deletes a `DataReader` that belongs to the `Subscriber`.

### Parameters

*in* `DataReader a_datareader` - a reference to the `DataReader`, which is to be deleted.



## Return Value

*int* - 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`, or when the `DataReader` still contains unreturned loans. In those cases the operation also 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`
- `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, the `DataReader` contains one or more `ReadCondition` or `QueryCondition` objects or the `DataReader` still contains unreturned loans.

### 3.5.1.6 enable (inherited)

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

## Synopsis

```
import DDS.*;
public int
    enable
        (void);
```

### 3.5.1.7 end\_access

#### Scope

DDS.Subscriber

#### Synopsis

```
import DDS.*;
public int
    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

```
import DDS.*;
public int
    get_datareaders
        (DataReaderSeqHolder readers,
         int sample_states,
         int view_states,
         int 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

```
import DDS.*;
public void
    get_default_datareader_qos
        (DataReaderQosHolder qos);
```

#### Description

This operation gets the default QosPolicy settings of the DataReader.

## Parameters

*inout DataReaderQosHolder qos* - a reference to the destination `DataReaderQosHolder` object in which the default `DataReaderQos` for the `Subscriber` is written..

## Return Value

*int* - 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` object 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 object referenced to by `qos`. Any settings in the object 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 3, *QosPolicy Default Attributes*, on page 40.

### Return Code

When the operation returns:

- `RETCODE_OK` - the default `DataReader QosPolicy` settings of this `Subscriber` have successfully been copied into the specified `DataReaderQosHolder` 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

```
import DDS.*;
public SubscriberListener
    get_listener
        (void);
```

## Description

This operation allows access to a `SubscriberListener`.

## Parameters

<none>

## Return Value

*SubscriberListener* - result is a reference 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 reference is returned.

### 3.5.1.11 `get_participant`

## Scope

`DDS.Subscriber`

## Synopsis

```
import DDS.*;
public DomainParticipant
    get_participant
        (void);
```

## Description

This operation returns the `DomainParticipant` associated with the `Subscriber` or the null pointer.

## Parameters

<none>

## Return Value

*DomainParticipant* - a reference 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

```
import DDS.*;
public int
    get_qos
        (SubscriberQosHolder qos);
```

#### Description

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

#### Parameters

*inout* `SubscriberQosHolder qos` - the destination `SubscriberQosHolder` object in which the `QosPolicy` settings will be copied.

#### Return Value

*int* - 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 referenced 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 `SubscriberQosHolder` 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 interface `Entity` for further explanation.

#### Synopsis

```
import DDS.*;
public int
    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

```
import DDS.*;
public StatusCondition
    get_statuscondition
        (void);
```

### 3.5.1.15 `lookup_datareader`

#### Scope

```
DDS.Subscriber
```

#### Synopsis

```
import DDS.*;
public DataReader
    lookup_datareader
        (String 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 String topic\_name* - the name of the `Topic`, which is attached to the `DataReader` to look for.

## Return Value

*DataReader* - Return value is a reference to the *DataReader* found. When no such *DataReader* is found, the null reference 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

```
import DDS.*;
public int
    notify_datareaders
        (void);
```

## Description

This operation invokes the *on\_data\_available* operation on *DataReaderListener* objects which are attached to contained *DataReader* entities and which have new, available data.

## Parameters

<none>

## Return Value

*int* - 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 for *DataReaderListener* objects that are attached to contained *DataReader* entities which have received information that has not yet been processed by those *DataReaders*.

The `notify_datareaders` operation ignores the bit mask value of the individual `DataReaderListener` objects, even when the `DATA_AVAILABLE_STATUS` bit has not been set on a `DataReader` that has new data available. The `on_data_available` operation will still be invoked, when the `DATA_AVAILABLE_STATUS` bit has not been set, 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` - the Data Distribution Service ran out of resources to complete this operation.

### 3.5.1.17 `set_default_datareader_qos`

#### Scope

`DDS.Subscriber`

#### Synopsis

```
import DDS.*;
public int
    set_default_datareader_qos
        (DataReaderQos qos);
```

#### Description

This operation sets the default `DataReaderQos` of the `DataReader`.

#### Parameters

*in* `DataReaderQos qos` - the `DataReaderQos` object, which contains the new default `QosPolicy` settings for the newly created `DataReaders`.

#### Return Value

*int* - 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

```
import DDS.*;
public int
    set_listener
        (SubscriberListener a_listener,
         int mask);
```

#### Description

This operation attaches a `SubscriberListener` to the `Subscriber`.

## Parameters

*in SubscriberListener a\_listener* - a reference to the SubscriberListener instance, which will be attached to the Subscriber.

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

## Return Value

*int* - Possible return codes of the operation are: RETCODE\_OK, RETCODE\_ERROR, RETCODE\_UNSUPPORTED, 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 reference, it represents a listener that is treated as a NOOP<sup>1</sup> for all statuses activated in the bitmask.

### 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.

---

1. Short for **No-Operation**, an instruction that performs nothing at all.



Be aware that the `SUBSCRIPTION_MATCHED_STATUS` is not applicable when the infrastructure does not have the information available to determine connectivity. This is the case when OpenSplice is configured not to maintain discovery information in the Networking Service. (See the description for the `NetworkingService/Discovery/enabled` property in the Deployment Manual for more information about this subject.) In this case the operation will return `RETCODE_UNSUPPORTED`.

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 applicable 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).

### Return Code

When the operation returns:

- `RETCODE_OK` - the `SubscriberListener` is attached
- `RETCODE_ERROR` - an internal error has occurred
- `RETCODE_UNSUPPORTED` - a status was selected that cannot be supported because the infrastructure does not maintain the required connectivity information.
- `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

```
import DDS.*;
public int
    set_qos
        (SubscriberQos qos);
```

#### Description

This operation replaces the existing set of `QosPolicy` settings for a `Subscriber`.

#### Parameters

*in* `SubscriberQos qos` - the new set of `QosPolicy` settings for the `Subscriber`.

#### Return Value

*int* - 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 object with 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`.
- `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` class.

### 3.5.2.1 Interface `DataReader`

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 interface. 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 is as follows:

```
public interface DataReader
{
//
// extends interface class Entity
//
// public StatusCondition
//     get_statuscondition
//         (void);
// public int
//     get_status_changes
//         (void);
// public int
//     enable
//         (void);
//
// operations (implemented in the data type specific DataReader)
//
// public int
//     read
//         (<type>SeqHolder data_values,
//          SampleInfoSeqHolder info_seq,
//          int max_samples,
//          int sample_states,
//          int view_states,
//          int instance_states);
```

```

// public int
//     take
//         (<type>SeqHolder data_values,
//          SampleInfoSeqHolder info_seq,
//          int max_samples,
//          int sample_states,
//          int view_states,
//          int instance_states);
// public int
//     read_w_condition
//         (<type>SeqHolder data_values,
//          SampleInfoSeqHolder info_seq,
//          int max_samples,
//          ReadCondition a_condition);
// public int
//     take_w_condition
//         (<type>SeqHolder data_values,
//          SampleInfoSeqHolder info_seq,
//          int max_samples,
//          ReadCondition a_condition);
// public int
//     read_next_sample
//         (<type>Holder data_value,
//          SampleInfoHolder sample_info);
// public int
//     take_next_sample
//         (<type>Holder data_value,
//          SampleInfoHolder sample_info);
// public int
//     read_instance
//         (<type>SeqHolder data_values,
//          SampleInfoSeqHolder info_seq
//          int max_samples,
//          long a_handle,
//          int sample_states,
//          int view_states,
//          int instance_states);
// public int
//     take_instance
//         (<type>SeqHolder data_values,
//          SampleInfoSeqHolder info_seq
//          int max_samples,
//          long a_handle,
//          int sample_states,
//          int view_states,
//          int instance_states);
// public int
//     read_next_instance
//         (<type>SeqHolder data_values,
//          SampleInfoSeqHolder info_seq

```

```

//      int max_samples,
//      long a_handle,
//      int sample_states,
//      int view_states,
//      int instance_states);
// public int
//   take_next_instance
//     (<type>SeqHolder data_values,
//      SampleInfoSeqHolder info_seq
//      int max_samples,
//      long a_handle,
//      int sample_states,
//      int view_states,
//      int instance_states);
// public int
//   read_next_instance_w_condition
//     (<type>SeqHolder data_values,
//      SampleInfoSeqHolder info_seq
//      int max_samples,
//      long a_handle,
//      ReadCondition a_condition);
// public int
//   take_next_instance_w_condition
//     (<type>SeqHolder data_values,
//      SampleInfoSeqHolder info_seq
//      int max_samples,
//      long a_handle,
//      ReadCondition a_condition);
// public int
//   return_loan
//     (<type>SeqHolder data_values,
//      SampleInfoSeqHolder info_seq);
// public int
//   get_key_value
//     (<type>SeqHolder key_holder,
//      long handle);
// public long
//   lookup_instance
//     (<type> instance_data);
//
// implemented API operations
//
//   public ReadCondition
//     create_readcondition
//       (int sample_states,
//        int view_states,
//        int instance_states);
//
//   public QueryCondition
//     create_querycondition

```



```
(int sample_states,
int view_states,
int instance_states,
String query_expression,
String[] query_parameters);

public int
delete_readcondition
    (ReadCondition a_condition);

public int
delete_contained_entities
    (void);

public int
set_qos
    (DataReaderQos qos);

public int
get_qos
    (DataReaderQosHolder qos);

public int
set_listener
    (DataReaderListener a_listener,
    int mask);

public DataReaderListener
get_listener
    (void);

public TopicDescription
get_topicdescription
    (void);

public Subscriber
get_subscriber
    (void);

public int
get_sample_rejected_status
    (SampleRejectedStatusHolder status);

public int
get_liveliness_changed_status
    (LivelinessChangedStatusHolder status);

public int
get_requested_deadline_missed_status
    (RequestedDeadlineMissedStatusHolder status);
```

```

    public int
        get_requested_incompatible_qos_status
            (RequestedIncompatibleQosStatusHolder status);

    public int
        get_subscription_matched_status
            (SubscriptionMatchedStatusHolder status);

    public int
        get_sample_lost_status
            (SampleLostStatusHolder status);

    public int
        wait_for_historical_data
            (Duration_t max_wait);

    public int
        get_matched_publications
            (InstanceHandleSeqHolder publication_handles);

    public int
        get_matched_publication_data
            (PublicationBuiltinTopicDataHolder publication_data,
             long publication_handle);
};

```

The following 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

```

import DDS.*;
public QueryCondition
    create_querycondition
        (int sample_states,
         int view_states,
         int instance_states,
         String query_expression,

```

```
String[] query_parameters);
```

## Description

This operation creates a new `QueryCondition` for the `DataReader`.

## Parameters

*in int sample\_states* - a mask, which selects only those samples with the desired sample states.

*in int view\_states* - a mask, which selects only those samples with the desired view states.

*in int instance\_states* - a mask, which selects only those samples with the desired instance states.

*in String query\_expression* - the query string, which must be a subset of the SQL query language.

*in String[] 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 parameter in the `query_expression`, the `query_parameters` should at least contain `n+1 = 9` values).

## Return Value

*QueryCondition* - Result value is a reference to the `QueryCondition`. When the operation fails, the null reference 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 reference 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 parameter 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

```
import DDS.*;
public ReadCondition
    create_readcondition
        (int sample_states,
         int view_states,
         int instance_states);
```

#### Description

This operation creates a new `ReadCondition` for the `DataReader`.

#### Parameters

*in int sample\_states* - a mask, which selects only those samples with the desired sample states.

*in int view\_states* - a mask, which selects only those samples with the desired view states.

*in int instance\_states* - a mask, which selects only those samples with the desired instance states.

#### Return Value

*ReadCondition* - Result value is a reference to the `ReadCondition`. When the operation fails, the null reference 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 reference 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

```
import DDS.*;
public int
    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

`int` - 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

```
import DDS.*;
public int
    delete_readcondition
        (ReadCondition a_condition);
```

#### Description

This operation deletes a ReadCondition or QueryCondition which is attached to the DataReader.

#### Parameters

*in* `ReadCondition a_condition` - a reference to the ReadCondition or QueryCondition which is to be deleted.

## Return Value

*int* - 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`
- `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

```
import DDS.*;
public int
    enable
        (void);
```

### 3.5.2.7 `get_key_value` (abstract)

This 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

```
import DDS.*;
public int
    get_key_value
        (<type>Holder key_holder,
         long handle);
```

### 3.5.2.8 `get_listener`

#### Scope

```
DDS.DataReader
```

#### Synopsis

```
import DDS.*;
public DataReaderListener
    get_listener
        (void);
```

#### Description

This operation allows access to a `DataReaderListener`.

#### Parameters

<none>

#### Return Value

*DataReaderListener* - result is a reference 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 reference is returned.



### 3.5.2.9 `get_liveliness_changed_status`

#### Scope

`DDS.DataReader`

#### Synopsis

```
import DDS.*;
public int
    get_liveliness_changed_status
        (LivelinessChangedStatusHolder status);
```

#### Description

This operation obtains the `LivelinessChangedStatus` object of the `DataReader`.

#### Parameters

*inout* `LivelinessChangedStatusHolder status` - the contents of the `LivelinessChangedStatus` object of the `DataReader` will be copied into the `LivelinessChangedStatusHolder` specified by `status`.

#### Return Value

*int* - 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 `LivelinessChangedStatus` object of the `DataReader`. This object 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

```
import DDS.*;
public int
    get_matched_publication_data
        (PublicationBuiltinTopicDataHolder publication_data,
         long publication_handle);
```

#### Description

This operation retrieves information on the specified publication that is currently “associated” with the `DataReader`.

#### Parameters

*inout* `PublicationBuiltinTopicDataHolder publication_data` - a Holder for the sample in which the information about the specified publication is to be stored.

*in* `long publication_handle` - a handle to the publication whose information needs to be retrieved.

#### Return Value

*int* - Possible return codes of the operation are: `RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_UNSUPPORTED`, `RETCODE_ALREADY_DELETED`, `RETCODE_OUT_OF_RESOURCES` or `RETCODE_NOT_ENABLED`.

#### Detailed Description

This operation retrieves information on the specified publication that is currently “associated” with the `DataReader`. That is, a publication with a matching Topic and compatible QoS that the application has not indicated should be “ignored” by means of the `ignore_publication` operation on the `DomainParticipant`.

The `publication_handle` must correspond to a publication currently associated with the `DataReader`, otherwise the operation will fail and return `RETCODE_BAD_PARAMETER`. The operation `get_matched_publications` can be used to find the publications that are currently matched with the `DataReader`.

The operation may also fail if the infrastructure does not hold the information necessary to fill in the `publication_data`. This is the case when OpenSplice is configured not to maintain discovery information in the Networking Service. (See

the description for the `NetworkingService/Discovery/enabled` property in the Deployment Manual for more information about this subject.) In this case the operation will return `RETCODE_UNSUPPORTED`.

### Return Code

When the operation returns:

- `RETCODE_OK` - the information on the specified publication has successfully been retrieved.
- `RETCODE_ERROR` - an internal error has occurred.
- `RETCODE_UNSUPPORTED` - OpenSplice is configured not to maintain the information about “associated” publications.
- `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.

## 3.5.2.11 `get_matched_publications`

### Scope

`DDS.DataReader`

### Synopsis

```
import DDS.*;
public int
    get_matched_publications
        (InstanceHandleSeqHolder publication_handles);
```

### Description

This operation retrieves the list of publications currently "associated" with the `DataReader`.

### Parameters

*inout* `InstanceHandleSeqHolder publication_handles` - a Holder for an array which is used to pass the list of all associated publications.

### Return Value

*int* - Possible return codes of the operation are: `RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_UNSUPPORTED`, `RETCODE_ALREADY_DELETED`, `RETCODE_OUT_OF_RESOURCES` or `RETCODE_NOT_ENABLED`.

## Detailed Description

This operation retrieves the list of publications currently "associated" with the `DataReader`. That is, subscriptions that have a matching Topic and compatible QoS that the application has not indicated should be "ignored" by means of the `ignore_publication` operation on the `DomainParticipant`.

The array inside the `publication_handles` Holder may be pre-allocated by the application and can be re-used in a subsequent invocation of the `get_matched_publications` operation. If the pre-allocated array is not big enough to hold the number of associated publications, the array will automatically be (re-)allocated to fit the required size.

The handles returned in the `publication_handles` array are the ones that are used by the DDS implementation to locally identify the corresponding matched publication entities. You can access more detailed information about a particular publication by passing its `publication_handle` to either the `get_matched_publication_data` operation or to the `read_instance` operation on the built-in reader for the "DCPSPublication" topic.



Be aware that since an instance handle is an opaque datatype, it does not necessarily mean that the handles obtained from the `get_matched_publications` operation have the same value as the ones that appear in the `instance_handle` field of the `SampleInfo` when retrieving the publication info through corresponding "DCSPublications" built-in reader. You can't just compare two handles to determine whether they represent the same publication. If you want to know whether two handles actually do represent the same publication, use both handles to retrieve their corresponding `PublicationBuiltinTopicData` samples and then compare the key field of both samples.

The operation may fail if the infrastructure does not locally maintain the connectivity information. This is the case when OpenSplice is configured not to maintain discovery information in the Networking Service. (See the description for the `NetworkingService/Discovery/enabled` property in the Deployment Manual for more information about this subject.) In this case the operation will return `DDS_RETCODE_UNSUPPORTED`.

### Return Code

When the operation returns:

- `RETCODE_OK` - the list of associated publications has successfully been obtained.
- `RETCODE_ERROR` - an internal error has occurred.
- `RETCODE_UNSUPPORTED` - OpenSplice is configured not to maintain the information about "associated" publications.
- `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.

### 3.5.2.12 **get\_qos**

#### Scope

DDS.DataReader

#### Synopsis

```
import DDS.*;
public int
    get_qos
        (DataReaderQosHolder qos);
```

#### Description

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

#### Parameters

*inout DataReaderQosHolder qos* - the destination DataReaderQosHolder object in which the QosPolicy settings will be copied.

#### Return Value

*int* - 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 referenced 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 DataReaderQosHolder 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

```
import DDS.*;
public int
    get_requested_deadline_missed_status
        (RequestedDeadlineMissedStatusHolder status);
```

#### Description

This operation obtains the `RequestedDeadlineMissedStatus` object of the `DataReader`.

#### Parameters

*inout* `RequestedDeadlineMissedStatusHolder status` - the contents of the `RequestedDeadlineMissedStatus` object of the `DataReader` will be copied into the `RequestedDeadlineMissedStatusHolder` specified by `status`.

#### Return Value

*int* - 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` object of the `DataReader`. This object 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

```
import DDS.*;
public int
    get_requested_incompatible_qos_status
        (RequestedIncompatibleQosStatusHolder status);
```

#### Description

This operation obtains the `RequestedIncompatibleQosStatus` object of the `DataReader`.

#### Parameters

*inout* `RequestedIncompatibleQosStatusHolder status` - the contents of the `RequestedIncompatibleQosStatus` object of the `DataReader` will be copied into the `RequestedIncompatibleQosStatusHolder` specified by `status`.

#### Return Value

*int* - 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` object of the `DataReader`. This object 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

```
import DDS.*;
public int
    get_sample_lost_status
        (SampleLostStatusHolder status);
```

#### Description

This operation obtains the SampleLostStatus object of the DataReader.

#### Parameters

*inout SampleLostStatusHolder status* - the contents of the SampleLostStatus object of the DataReader will be copied into the SampleLostStatusHolder 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

*int* - 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 object of the DataReader. This object 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**

```
import DDS.*;
public int
    get_sample_rejected_status
        (SampleRejectedStatusHolder status);
```

**Detailed Description**

This operation obtains the `SampleRejectedStatus` object of the `DataReader`.

**Parameters**

*inout* `SampleRejectedStatusHolder status` - the contents of the `SampleRejectedStatus` object of the `DataReader` will be copied into the `SampleRejectedStatusHolder` specified by `status`.

**Return Value**

*int* - 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` object of the `DataReader`. This object contains the information whether a received sample has been rejected. Samples may be rejected by the `DataReader` when it runs out of `resource_limits` to store incoming samples. Usually this means that old samples need to be ‘consumed’ (for example by ‘taking’ them instead of ‘reading’ them) to make room for newly incoming samples.

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 interface `Entity` for further explanation.

**Synopsis**

```
import DDS.*;
public int
    get_status_changes
        (void);
```

**3.5.2.18 `get_statuscondition` (inherited)**

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

**Synopsis**

```
import DDS.*;
public StatusCondition
    get_statuscondition
        (void);
```

**3.5.2.19 `get_subscriber`****Scope**

```
DDS.DataReader
```

**Synopsis**

```
import DDS.*;
public Subscriber
    get_subscriber
        (void);
```

**Description**

This operation returns the `Subscriber` to which the `DataReader` belongs.

**Parameters**

<none>

**Return Value**

*Subscriber* - Return value is a reference to the Subscriber object 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 reference is returned.

**3.5.2.20 get\_subscription\_matched\_status****Scope**

DDS.DataReader

**Synopsis**

```
import DDS.*;
public int
    get_subscription_match_status
        (SubscriptionMatchedStatusHolder status);
```

**Description**

This operation obtains the SubscriptionMatchedStatus object of the DataReader.

**Parameters**

*inout SubscriptionMatchedStatusHolder status* - the contents of the SubscriptionMatchedStatus object of the DataReader will be copied into the SubscriptionMatchedStatusHolder specified by status.

**Return Value**

*int* - Possible return codes of the operation are: RETCODE\_OK, RETCODE\_ERROR, RETCODE\_UNSUPPORTED, DDS\_RETCODE\_ALREADY\_DELETED or DDS\_RETCODE\_OUT\_OF\_RESOURCES.

**Detailed Description**

This operation obtains the SubscriptionMatchedStatus object of the DataReader. This object contains the information whether a new match has been discovered for the current subscription, or whether an existing match has ceased to exist.

This means that the status represents that either a `DataWriter` object has been discovered by the `DataReader` with the same Topic and a compatible Qos, or that a previously discovered `DataWriter` has ceased to be matched to the current `DataReader`. A `DataWriter` may cease to match when it gets deleted, when it changes its Qos to a value that is incompatible with the current `DataReader` or when either the `DataReader` or the `DataWriter` has chosen to put its matching counterpart on its ignore-list using the `ignore_publication` or `ignore_subscription` operations on the `DomainParticipant`.

The operation may fail if the infrastructure does not hold the information necessary to fill in the `SubscriptionMatchedStatus`. This is the case when OpenSplice is configured not to maintain discovery information in the Networking Service. (See the description for the `NetworkingService/Discovery/enabled` property in the Deployment Manual for more information about this subject.) In this case the operation will return `RETCODE_UNSUPPORTED`.

The `SubscriptionMatchedStatus` can also be monitored using a `DataReaderListener` or by using the associated `StatusCondition`.

### Return Code

When the operation returns:

- `RETCODE_OK` - the current `SubscriptionMatchedStatus` of this `DataReader` has successfully been copied into the specified status parameter.
- `RETCODE_ERROR` - an internal error has occurred.
- `RETCODE_UNSUPPORTED` - OpenSplice is configured not to maintain the information about “associated” subscriptions.
- `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.

### 3.5.2.21 `get_topicdescription`

#### Scope

`DDS.DataReader`

#### Synopsis

```
import DDS.*;
public TopicDescription
    get_topicdescription
        (void);
```

## Description

This operation returns the `TopicDescription` which is associated with the `DataReader`.

## Parameters

<none>

## Return Value

*TopicDescription* - Return value is a reference to the `TopicDescription` object 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 reference is returned.

### 3.5.2.22 lookup\_instance (abstract)

To use this operation, a datatype specific interface `<type>DataReader` must be used. For further explanation see the description for the fictional data type `Foo` interface `FooDataReader`.

## Synopsis

```
import DDS.*;
public long
    lookup_instance
        (<type> instance_data);
```

### 3.5.2.23 read (abstract)

To use this operation, a datatype specific interface `<type>DataReader` must be used. For further explanation see the description for the fictional data type `Foo` interface `FooDataReader`.

## Synopsis

```
import DDS.*;
public int
    read
        (<type>SeqHolder data_values,
         SampleInfoSeqHolder info_seq,
         int max_samples,
         int sample_states,
         int view_states,
         int instance_states);
```

### 3.5.2.24 read\_instance (abstract)

To use this operation, a datatype specific interface `<type>DataReader` must be used. For further explanation see the description for the fictional data type `Foo` interface `FooDataReader`.

#### Synopsis

```
import DDS.*;
public int
    read_instance
        (<type>SeqHolder data_values,
         SampleInfoSeqHolder info_seq,
         int max_samples,
         long a_handle,
         int sample_states,
         int view_states,
         int instance_states);
```

### 3.5.2.25 read\_next\_instance (abstract)

To use this operation, a datatype specific interface `<type>DataReader` must be used. For further explanation see the description for the fictional data type `Foo` interface `FooDataReader`.

#### Synopsis

```
import DDS.*;
public int
    read_next_instance
        (<type>SeqHolder data_values,
         SampleInfoSeqHolder info_seq,
         int max_samples,
         long a_handle,
         int sample_states,
         int view_states,
         int instance_states);
```

### 3.5.2.26 read\_next\_instance\_w\_condition (abstract)

To use this operation, a datatype specific interface `<type>DataReader` must be used. For further explanation see the description for the fictional data type `Foo` interface `FooDataReader`.

#### Synopsis

```
import DDS.*;
public int
    read_next_instance_w_condition
        (<type>SeqHolder data_values,
         SampleInfoSeqHolder info_seq,
```

```
int max_samples,
long a_handle,
ReadCondition a_condition);
```

### 3.5.2.27 read\_next\_sample (abstract)

To use this operation, a datatype specific interface `<type>DataReader` must be used. For further explanation see the description for the fictional data type `Foo` interface `FooDataReader`.

#### Synopsis

```
import DDS.*;
public int
    read_next_sample
        (<type>Holder data_value,
         SampleInfoHolder sample_info);
```

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

### 3.5.2.28 read\_w\_condition (abstract)

To use this operation, a datatype specific interface `<type>DataReader` must be used. For further explanation see the description for the fictional data type `Foo` interface `FooDataReader`.

#### Synopsis

```
import DDS.*;
public int
    read_w_condition
        (<type>SeqHolder data_values,
         SampleInfoSeqHolder info_seq,
         int max_samples,
         ReadCondition a_condition);
```

### 3.5.2.29 return\_loan (abstract)

To use this operation, a datatype specific interface `<type>DataReader` must be used. For further explanation see the description for the fictional data type `Foo` interface `FooDataReader`.

#### Synopsis

```
import DDS.*;
public int
    return_loan
        (<type>SeqHolder data_values,
         SampleInfoSeqHolder info_seq);
```

### 3.5.2.30 `set_listener`

#### Scope

`DDS.DataReader`

#### Synopsis

```
import DDS.*;
public int
    set_listener
        (DataReaderListener a_listener,
         int mask);
```

#### Description

This operation attaches a `DataReaderListener` to the `DataReader`.

#### Parameters

*in DataReaderListener a\_listener* - a reference to the `DataReaderListener` instance, which will be attached to the `DataReader`.

*in int mask* - a bit mask in which each bit enables the invocation of the `DataReaderListener` for a certain status.

#### Return Value

*int* - Possible return codes of the operation are: `RETCODE_OK`, `RETCODE_ERROR`, `RETCODE_UNSUPPORTED`, `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 reference, it represents a listener that is treated as a NOOP<sup>1</sup> for all statuses activated in the bitmask.

#### 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 performs nothing at all.



`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`.



Be aware that the `SUBSCRIPTION_MATCHED_STATUS` is not applicable when the infrastructure does not have the information available to determine connectivity. This is the case when OpenSplice is configured not to maintain discovery information in the Networking Service. (See the description for the `NetworkingService/Discovery/enabled` property in the Deployment Manual for more information about this subject.) In this case the operation will return `RETCODE_UNSUPPORTED`.

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 applicable 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_UNSUPPORTED` - a status was selected that cannot be supported because the infrastructure does not maintain the required connectivity information.
- `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

```
import DDS.*;
public int
    set_qos
        (DataReaderQos qos);
```

#### Description

This operation replaces the existing set of `QosPolicy` settings for a `DataReader`.

#### Parameters

*in* `DataReaderQos qos` - the new set of `QosPolicy` settings for the `DataReader`.

## Return Value

*int* - 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` setting (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.
- `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)

To use this operation, a datatype specific interface `<type>DataReader` must be used. For further explanation see the description for the fictional data type `Foo` interface `FooDataReader`.

#### Synopsis

```
import DDS.*;
public int
    take
        (<type>SeqHolder data_values,
         SampleInfoSeqHolder info_seq,
         int max_samples,
         int sample_states,
         int view_states,
         int instance_states);
```

### 3.5.2.33 take\_instance (abstract)

To use this operation, a datatype specific interface `<type>DataReader` must be used. For further explanation see the description for the fictional data type `Foo` interface `FooDataReader`.

#### Synopsis

```
import DDS.*;
public int
    take_instance
        (<type>SeqHolder data_values,
         SampleInfoSeqHolder info_seq,
         int max_samples,
         long a_handle,
         int sample_states,
         int view_states,
         int instance_states);
```

### 3.5.2.34 take\_next\_instance (abstract)

To use this operation, a datatype specific interface `<type>DataReader` must be used. For further explanation see the description for the fictional data type `Foo` interface `FooDataReader`.

#### Synopsis

```
import DDS.*;
public int
    take_next_instance
        (<type>SeqHolder data_values,
         SampleInfoSeqHolder info_seq,
         int max_samples,
```

```

    long a_handle,
    int sample_states,
    int view_states,
    int instance_states);

```

### 3.5.2.35 take\_next\_instance\_w\_condition (abstract)

To use this operation, a datatype specific interface `<type>DataReader` must be used. For further explanation see the description for the fictional data type `Foo` interface `FooDataReader`.

#### Synopsis

```

import DDS.*;
public int
    take_next_instance_w_condition
        (<type>SeqHolder data_values,
         SampleInfoSeqHolder info_seq,
         int max_samples,
         long a_handle,
         ReadCondition a_condition);

```

### 3.5.2.36 take\_next\_sample (abstract)

To use this operation, a datatype specific interface `<type>DataReader` must be used. For further explanation see the description for the fictional data type `Foo` interface `FooDataReader`.

#### Synopsis

```

import DDS.*;
public int
    take_next_sample
        (<type>Holder data_value,
         SampleInfoHolder sample_info);

```

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

### 3.5.2.37 take\_w\_condition (abstract)

To use this operation, a datatype specific interface `<type>DataReader` must be used. For further explanation see the description for the fictional data type `Foo` interface `FooDataReader`.

#### Synopsis

```

import DDS.*;
public int
    take_w_condition
        (<type>SeqHolder data_values,
         SampleInfoSeqHolder info_seq,
         int max_samples,

```

```
ReadCondition a_condition);
```

### 3.5.2.38 wait\_for\_historical\_data

#### Scope

```
DDS.DataReader
```

#### Synopsis

```
import DDS.*;
public int
    wait_for_historical_data
        (Duration_t max_wait);
```

#### Description

This operation will block the application thread until all “historical” data is received.

#### Parameters

*in 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

*int* - 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 Interface **FooDataReader**

The pre-processor generates from IDL type descriptions the application `<type>DataReader` interfaces. For each application data type that is used as `Topic` data type, a typed interface `<type>DataReader` extends the `DataReader` interface. In this paragraph, the interface `FooDataReader` in the package `SPACE` describes the operations of these extending `<type>DataReader` interfaces as an example for the fictional application type `Foo` (defined in the module `SPACE`).

#### 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 is as follows:

```

public interface FooDataReader
{
    //
    // extends interface Entity
    //
    // public StatusCondition
    //     get_statuscondition
    //         (void);
    // public int
    //     get_status_changes
    //         (void);
    // public int
    //     enable
    //         (void);
    //
    // extended to interface DataReader
    //
    // public ReadCondition
    //     create_readcondition
    //         (int sample_states,
    //          int view_states,
    //          int instance_states);

    // public QueryCondition
    //     create_querycondition
    //         (int sample_states,
    //          int view_states,
    //          int instance_states,
    //          String query_expression,
    //          String[] query_parameters);

    // public int
    //     delete_readcondition
    //         (ReadCondition a_condition);

    // public int
    //     delete_contained_entities
    //         (void);

    // public int
    //     set_qos
    //         (DataReaderQos qos);

    // public int
    //     get_qos
    //         (DataReaderQosHolder qos);

    // public int
    //     set_listener
    //         (DataReaderListener a_listener,

```



```

//          int mask);

// public DataReaderListener
//   get_listener
//   (void);

// public TopicDescription
//   get_topicdescription
//   (void);

// public Subscriber
//   get_subscriber
//   (void);

// public int
//   get_sample_rejected_status
//   (SampleRejectedStatusHolder status);

// public int
//   get_liveliness_changed_status
//   (LivelinessChangedStatusHolder status);

// public int
//   get_requested_deadline_missed_status
//   (RequestedDeadlineMissedStatusHolder status);

// public int
//   get_requested_incompatible_qos_status
//   (RequestedIncompatibleQosStatusHolder status);

// public int
//   get_subscription_matched_status
//   (SubscriptionMatchedStatusHolder status);

// public int
//   get_sample_lost_status
//   (SampleLostStatusHolder status);

// public int
//   wait_for_historical_data
//   (Duration_t max_wait);

// public int
//   get_matched_publications
//   (InstanceHandleSeqHolder publication_handles);

// public int
//   get_matched_publication_data
//   (PublicationBuiltinTopicDataHolder publication_data,
//    long publication_handle);

```

```

//
// implemented API operations
//
public int
    read
        (FooSeqHolder data_values,
         SampleInfoSeqHolder info_seq,
         int max_samples,
         int sample_states,
         int view_states,
         int instance_states);
public int
    take
        (FooSeqHolder data_values,
         SampleInfoSeqHolder info_seq,
         int max_samples,
         int sample_states,
         int view_states,
         int instance_states);
public int
    read_w_condition
        (FooSeqHolder data_values,
         SampleInfoSeqHolder info_seq,
         int max_samples,
         ReadCondition a_condition);
public int
    take_w_condition
        (FooSeqHolder data_values,
         SampleInfoSeqHolder info_seq,
         int max_samples,
         ReadCondition a_condition);
public int
    read_next_sample
        (FooHolder data_value,
         SampleInfoHolder sample_info);
public int
    take_next_sample
        (FooHolder data_value,
         SampleInfoHolder sample_info);
public int
    read_instance
        (FooSeqHolder data_values,
         SampleInfoSeqHolder info_seq,
         int max_samples,
         long a_handle,
         int sample_states,
         int view_states,
         int instance_states);
public int
    take_instance

```

```

        (FooSeqHolder data_values,
         SampleInfoSeqHolder info_seq,
         int max_samples,
         long a_handle,
         int sample_states,
         int view_states,
         int instance_states);
public int
    read_next_instance
        (FooSeqHolder data_values,
         SampleInfoSeqHolder sample_info,
         int max_samples,
         long a_handle,
         int sample_states,
         int view_states,
         int instance_states);
public int
    take_next_instance
        (FooSeqHolder data_values,
         SampleInfoSeqHolder sample_info,
         int max_samples,
         long a_handle,
         int sample_states,
         int view_states,
         int instance_states);
public int
    read_next_instance_w_condition
        (FooSeqHolder data_values,
         SampleInfoSeqHolder info_seq,
         int max_samples,
         long a_handle,
         ReadCondition a_condition);
public int
    take_next_instance_w_condition
        (FooSeqHolder data_values,
         SampleInfoSeqHolder info_seq,
         int max_samples,
         long a_handle,
         ReadCondition a_condition);
public int
    return_loan
        (FooSeqHolder data_values,
         SampleInfoSeqHolder info_seq);
public int
    get_key_value
        (FooHolder key_holder
         long handle);
long
    lookup_instance
        (Foo instance_data);

```

```
};
```

The following 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 interface `DataReader` for further explanation.

##### **Synopsis**

```
QueryCondition
create_querycondition
(int sample_states,
 int view_states,
 int instance_states,
 String query_expression,
 String[] query_parameters);
```

#### 3.5.2.41 **create\_readcondition (inherited)**

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

##### **Synopsis**

```
ReadCondition
create_readcondition
(int sample_states,
 int view_states,
 int instance_states);
```

#### 3.5.2.42 **delete\_contained\_entities (inherited)**

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

##### **Synopsis**

```
int
delete_contained_entities
(void);
```

#### 3.5.2.43 **delete\_readcondition (inherited)**

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

**Synopsis**

```
import DDS.*;
public int
    delete_readcondition
        (ReadCondition a_condition);
```

**3.5.2.44 enable (inherited)**

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

**Synopsis**

```
public int
    enable
        (void);
```

**3.5.2.45 get\_key\_value****Scope**

```
SPACE.FooDataReader
```

**Synopsis**

```
public int
    get_key_value
        (FooHolder key_holder,
         long handle);
```

**Description**

This operation retrieves the key value of a specific instance.

**Parameters**

*inout FooHolder key\_holder* - a reference to the sample in which the key values are stored.

*in long handle* - the handle to the instance from which to get the key value.

**Return Value**

*int* - 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 an `HANDLE_NIL` constant 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 that 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 `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` - this instance is not registered.

### 3.5.2.46 `get_listener` (inherited)

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

### Synopsis

```
import DDS.*;
public DataReaderListener
    get_listener
        (void);
```

### 3.5.2.47 `get_liveliness_changed_status` (inherited)

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

### Synopsis

```
import DDS.*;
public int
    get_liveliness_changed_status
        (LivelinessChangedStatusHolder status);
```

**3.5.2.48 get\_matched\_publication\_data (inherited)**

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

**Synopsis**

```
import DDS.*;
public int
    get_matched_publication_data
        (PublicationBuiltinTopicDataHolder publication_data,
         long publication_handle);
```

**3.5.2.49 get\_matched\_publications (inherited)**

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

**Synopsis**

```
import DDS.*;
public int
    get_matched_publications
        (InstanceHandleSeqHolder publication_handles);
```

**3.5.2.50 get\_qos (inherited)**

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

**Synopsis**

```
import DDS.*;
public void
    get_qos
        (DataReaderQosHolder qos);
```

**3.5.2.51 get\_requested\_deadline\_missed\_status (inherited)**

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

**Synopsis**

```
import DDS.*;
public int
    get_requested_deadline_missed_status
        (RequestedDeadlineMissedStatusHolder status);
```

**3.5.2.52 get\_requested\_incompatible\_qos\_status (inherited)**

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

**Synopsis**

```
import DDS.*;
public int
    get_requested_incompatible_qos_status
        (RequestedIncompatibleQosStatusHolder status);
```

**3.5.2.53 get\_sample\_lost\_status (inherited)**

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

**Synopsis**

```
import DDS.*;
public int
    get_sample_lost_status
        (SampleLostStatusHolder status);
```

**3.5.2.54 get\_sample\_rejected\_status (inherited)**

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

**Synopsis**

```
import DDS.*;
public int
    get_sample_rejected_status
        (SampleRejectedStatusHolder status);
```

**3.5.2.55 get\_status\_changes (inherited)**

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

**Synopsis**

```
public int
    get_status_changes
        (void);
```

**3.5.2.56 get\_statuscondition (inherited)**

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

**Synopsis**

```
import DDS.*;
public StatusCondition
    get_statuscondition
        (void);
```



### 3.5.2.57 **get\_subscriber (inherited)**

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

#### **Synopsis**

```
import DDS.*;
public Subscriber
    get_subscriber
        (void);
```

### 3.5.2.58 **get\_subscription\_match\_status (inherited)**

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

#### **Synopsis**

```
import DDS.*;
public int
    get_subscription_matched_status
        (SubscriptionMatchedStatusHolder status);
```

### 3.5.2.59 **get\_topicdescription (inherited)**

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

#### **Synopsis**

```
import DDS.*;
public TopicDescription
    get_topicdescription
        (void);
```

### 3.5.2.60 **lookup\_instance**

#### **Scope**

```
SPACE.FooDataReader
```

#### **Synopsis**

```
import DDS.*;
public long
    lookup_instance
        (Foo instance_data);
```

#### **Description**

This operation returns the value of the instance handle which corresponds to the `instance_data`.

**Parameters**

*in Foo instance\_data* - the instance for which the corresponding instance handle needs to be looked up.

**Return Value**

*long* - 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**

```
import DDS.*;
public int
    read
        (FooSeqHolder data_values,
         SampleInfoSeqHolder info_seq,
         int max_samples,
         int sample_states,
         int view_states,
         int instance_states);
```

**Description**

This operation reads a sequence of *Foo* samples from the *FooDataReader*.

**Parameters**

*inout FooSeqHolder data\_values* - the returned sample data sequence. *data\_values* is also used as an input to control the behaviour of this operation.

*inout SampleInfoSeqHolder info\_seq* - the returned *SampleInfo* object sequence. *info\_seq* is also used as an input to control the behaviour of this operation.

*in int max\_samples* - the maximum number of samples that is returned.

*in int sample\_states* - a mask, which selects only those samples with the desired sample states.

*in int view\_states* - a mask, which selects only those samples with the desired view states.

*in int instance\_states* - a mask, which selects only those samples with the desired instance states.

## Return Value

*int* - 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`. 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 interface 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` parameter), the operation also returns a sequence of `SampleInfo` objects with the parameter `info_seq`. The `info_seq` objects and `data_values` also determine the behaviour of this operation.

### Resource Control

The initial length of the `data_values` and `info_seq` sequences (`received_data.value.length` and `info_seq.value.length`) determine the precise behaviour of the read operation. The behaviour of the read operation is as specified by the following rules:

- On successful output, the sequence holders contain arrays whose length is equal to the number of returned samples. These arrays may be different from the ones originally passed in the sequence holders.
- If the Holder objects have `value == null`, or if their `value` fields point to arrays that have `length == 0`, the `received_data` and `info_seq` sequence are filled with elements that are “loaned” by the `FooDataReader`. On output, the sequence holders contain arrays whose length is equal to the number of returned samples. In this case the application will need to “return the loan” to the Data Distribution Service using the `return_loan` operation.
- If the Holder objects have `value` fields that point to arrays that have `length > 0`, the read operation will copy the `Foo` samples and `info_seq` values into the elements already allocated inside the sequences (effectively overwriting the current state of these elements). On output the sequence holders contain arrays whose length is equal to the number of returned samples. The application can pre-allocate the elements of the array and does not need to “return the loan”. The number of samples copied depends on the relative values of `length` and `max_samples`:
  - If `max_samples == LENGTH_UNLIMITED`, at most `length` 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 <= length`, 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 > length` 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 `FooDataReader`, because the output sequence can not 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 Holder objects have their value set to null.

The application must remember if it is necessary to “return the loan” or not. However, in many cases it may be simpler to always call `return_loan`, as this operation is harmless if the sequence does not hold a loan.

### Data Sequence

On output, the sequence of data values and the sequence of `SampleInfo` objects are of the same length and are in an one-to-one correspondence. Each `SampleInfo` object 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\_BAD\_PARAMETER* - one or more of the *received\_data* and *info\_seq* parameters is an invalid reference.
- *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* - the *max\_samples > length* and *max\_samples* is not *LENGTH\_UNLIMITED*
- *RETCODE\_NO\_DATA* - no samples that meet the constraints are available.

### 3.5.2.62 read\_instance

#### Scope

SPACE.FooDataReader

#### Synopsis

```
import DDS.*;
public int
    read_instance
        (FooSeqHolder data_values,
         SampleInfoSeqHolder info_seq,
         int max_samples,
         long a_handle,
         int sample_states,
         int view_states,
         int instance_states);
```

#### Description

This operation reads a sequence of *Foo* samples of a single instance from the *FooDataReader*.

#### Parameters

*inout FooSeqHolder data\_values* - the returned sample data sequence. *data\_values* is also used as an input to control the behaviour of this operation.

*inout SampleInfoSeqHolder info\_seq* - the returned *SampleInfo* object sequence. *info\_seq* is also used as an input to control the behaviour of this operation.

*in int max\_samples* - the maximum number of samples that is returned.

*in long a\_handle* - the single instance, the samples belong to.

*in int sample\_states* - a mask, which selects only those samples with the desired sample states.

*in int view\_states* - a mask, which selects only those samples with the desired view states.

*in int instance\_states* - a mask, which selects only those samples with the desired instance states.

## Return Value

*int* - 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` - one or more of the `received_data` and `info_seq` parameters is an invalid reference 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` - the `max_samples > length` and `max_samples` is not `LENGTH_UNLIMITED`
- `RETCODE_NO_DATA` - no samples that meet the constraints are available.

### 3.5.2.63 read\_next\_instance

#### Scope

SPACE.FooDataReader

#### Synopsis

```
import DDS.*;
public int
    read_next_instance
        (FooSeqHolder data_values,
         SampleInfoSeqHolder info_seq,
         int max_samples,
         long a_handle,
         int sample_states,
         int view_states,
         int instance_states);
```

#### Description

This operation reads a sequence of `Foo` samples of the next single instance from the `FooDataReader`.

#### Parameters

*inout FooSeqHolder data\_values* - the returned sample data sequence. `data_values` is also used as an input to control the behaviour of this operation.

*inout SampleInfoSeqHolder info\_seq* - the returned `SampleInfo` object sequence. `info_seq` is also used as an input to control the behaviour of this operation.

*in int max\_samples* - the maximum number of samples that is returned.

*in long a\_handle* - the current single instance, the returned samples belong to the next single instance.

*in int sample\_states* - a mask, which selects only those samples with the desired sample states.

*in int view\_states* - a mask, which selects only those samples with the desired view states.

*in int instance\_states* - a mask, which selects only those samples with the desired instance states.



## Return Value

*int* - 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.value` 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.value`, 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` - one or more of the `received_data` and `info_seq` parameters is an invalid reference 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` - the `max_samples > length` and `max_samples` is not `LENGTH_UNLIMITED`
- `RETCODE_NO_DATA` - no samples that meet the constraints are available.

### 3.5.2.64 `read_next_instance_w_condition`

#### Scope

`SPACE.FooDataReader`

#### Synopsis

```
import DDS.*;
public int
    read_next_instance_w_condition
        (FooSeqHolder data_values,
         SampleInfoSeqHolder info_seq,
         int max_samples,
         long a_handle,
         ReadCondition a_condition);
```

#### Description

This operation reads a sequence of `Foo` samples of the next single instance from the `FooDataReader`.

#### Parameters

*inout* `FooSeqHolder data_values` - the returned sample data sequence.  
`data_values` is also used as an input to control the behaviour of this operation.

*inout SampleInfoSeqHolder info\_seq* - the returned SampleInfo object sequence. *info\_seq* is also used as an input to control the behaviour of this operation.

*in int max\_samples* - the maximum number of samples that is returned.

*in long a\_handle* - the current single instance, the returned samples belong to the next single instance.

*in ReadCondition a\_condition* - a reference to a ReadCondition object or QueryCondition object which filters the data before it is returned by the read operation.

## Return Value

*int* - 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 FooDataReader\_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 FooDataReader\_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* - one or more of the received\_data, info\_seq and a\_condition parameters is an invalid reference 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* - the *max\_samples > length* and *max\_samples* is not *LENGTH\_UNLIMITED*
- *RETCODE\_NO\_DATA* - no samples that meet the constraints are available.

### 3.5.2.65 read\_next\_sample

#### Scope

SPACE.FooDataReader

#### Synopsis

```
import DDS.*;
public int
    read_next_sample
        (FooHolder data_value,
         SampleInfoHolder 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

```
import DDS.*;
public int
    read_w_condition
        (FooSeqHolder data_values,
         SampleInfoSeqHolder info_seq,
         int max_samples,
         ReadCondition a_condition);
```

#### Description

This operation reads a sequence of *Foo* samples from the *FooDataReader*, filtered by a *ReadCondition* or *QueryCondition*.

#### Parameters

*inout FooSeqHolder data\_values* - the returned sample data sequence. *data\_values* is also used as an input to control the behaviour of this operation.

*inout SampleInfoSeqHolder info\_seq* - the returned SampleInfo object sequence. *info\_seq* is also used as an input to control the behaviour of this operation.

*in int max\_samples* - the maximum number of samples that is returned.

*in ReadCondition a\_condition* - a reference to a ReadCondition or QueryCondition which filters the data before it is returned by the read operation.

## Return Value

*int* - 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* - one or more of the received\_data, info\_seq and a\_condition parameters is an invalid reference.
- *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` - the `max_samples > length` and `max_samples` is not `LENGTH_UNLIMITED`
- `RETCODE_NO_DATA` - no samples that meet the constraints are available.

### 3.5.2.67 `return_loan`

#### Scope

`SPACE.FooDataReader`

#### Synopsis

```
import DDS.*;
public int
    return_loan
        (FooSeqHolder data_values,
         SampleInfoSeqHolder 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* `FooSeqHolder data_values` - the sample data sequence which was loaned from the `DataReader`.

*inout* `SampleInfoSeqHolder info_seq` - the `SampleInfo` object sequence which was loaned from the `DataReader`.

#### Return Value

*int* - 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 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 `length=0` at the time the operation `read` or `take` was called. The application must remember if it is necessary to “return the loan” or not. 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` their `Holder` objects will have their value set to `null`.

### 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_BAD_PARAMETER` - either or both of `received_data` or `info_seq` is an invalid reference
- `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 `received_data` and `info_seq` does not belong to a single related pair
  - the `received_data` and `info_seq` was not obtained from this `FooDataReader`

### 3.5.2.68 set\_listener (inherited)

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

#### Synopsis

```
import DDS.*;
public int
    set_listener
        (DataReaderListener a_listener,
         int mask);
```

### 3.5.2.69 set\_qos (inherited)

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

#### Synopsis

```
import DDS.*;
public int
    set_qos
        (DataReaderQos qos);
```

### 3.5.2.70 take

#### Scope

```
SPACE.FooDataReader
```

#### Synopsis

```
import DDS.*;
public int
    take
        (FooSeqHolder data_values,
         SampleInfoSeqHolder info_seq,
         int max_samples,
         int sample_states,
         int view_states,
         int 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 FooSeqHolder data\_values* - the returned sample data sequence. *data\_values* is also used as an input to control the behaviour of this operation.

*inout SampleInfoSeqHolder info\_seq* - the returned SampleInfo object sequence. *info\_seq* is also used as an input to control the behaviour of this operation.

*in int max\_samples* - the maximum number of samples that is returned.

*in int sample\_states* - a mask, which selects only those samples with the desired sample states.

*in int view\_states* - a mask, which selects only those samples with the desired view states.

*in int instance\_states* - a mask, which selects only those samples with the desired instance states.

## Return Value

*int* - 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` 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_BAD_PARAMETER` - either or both of `received_data` or `info_seq` is an invalid reference
- `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* - the `max_samples > length` and `max_samples` is not `LENGTH_UNLIMITED`
- *RETCODE\_NO\_DATA* - no samples that meet the constraints are available.

### 3.5.2.71 take\_instance

#### Scope

`SPACE.FooDataReader`

#### Synopsis

```
import DDS.*;
public int
    take_instance
        (FooSeqHolder data_values,
         SampleInfoSeqHolder info_seq,
         int max_samples,
         long a_handle,
         int sample_states,
         int view_states,
         int 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 FooSeqHolder data\_values* - the returned sample data sequence. `data_values` is also used as an input to control the behaviour of this operation.

*inout SampleInfoSeqHolder info\_seq* - the returned `SampleInfo` object sequence. `info_seq` is also used as an input to control the behaviour of this operation.

*in int max\_samples* - the maximum number of samples that is returned.

*in long a\_handle* - the single instance, the samples belong to.

*in int sample\_states* - a mask, which selects only those samples with the desired sample states.

*in int view\_states* - a mask, which selects only those samples with the desired view states.

*in int instance\_states* - a mask, which selects only those samples with the desired instance states.

## Return Value

*int* - 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` - either or both of `received_data` or `info_seq` is an invalid reference 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` - the `max_samples > length` and `max_samples` is not `LENGTH_UNLIMITED`
- `RETCODE_NO_DATA` - no samples that meet the constraints are available.

### 3.5.2.72 `take_next_instance`

#### Scope

`SPACE.FooDataReader`

#### Synopsis

```
import DDS.*;
public int
    take_next_instance
        (FooSeqHolder data_values,
         SampleInfoSeqHolder sample_info,
         int max_samples,
         long a_handle,
         int sample_states,
```

```
int view_states,
int 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 FooSeqHolder data\_values* - the returned sample data sequence. *data\_values* is also used as an input to control the behaviour of this operation.

*inout SampleInfoSeqHolder info\_seq* - the returned `SampleInfo` object sequence. *info\_seq* is also used as an input to control the behaviour of this operation.

*in int max\_samples* - the maximum number of samples that is returned.

*in long a\_handle* - the current single instance, the returned samples belong to the next single instance.

*in int sample\_states* - a mask, which selects only those samples with the desired sample states.

*in int view\_states* - a mask, which selects only those samples with the desired view states.

*in int instance\_states* - a mask, which selects only those samples with the desired instance states.

## Return Value

*int* - 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* - either or both of *received\_data* or *info\_seq* is an invalid reference 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* - the *max\_samples > length* and *max\_samples* is not *LENGTH\_UNLIMITED*
- *RETCODE\_NO\_DATA* - no samples that meet the constraints are available.

### 3.5.2.73 **take\_next\_instance\_w\_condition**

#### Scope

SPACE.*FooDataReader*

#### Synopsis

```
import DDS.*;
public int
    take_next_instance_w_condition
        (FooSeqHolder data_values,
         SampleInfoSeqHolder info_seq,
         int max_samples,
         long a_handle,
         ReadCondition a_condition);
```

#### 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 FooSeqHolder data\_values* - the returned sample data sequence. *data\_values* is also used as an input to control the behaviour of this operation.

*inout SampleInfoSeqHolder info\_seq* - the returned *SampleInfo* object sequence. *info\_seq* is also used as an input to control the behaviour of this operation.

*in int max\_samples* - the maximum number of samples that is returned.

*in long a\_handle* - the current single instance, the returned samples belong to the next single instance.

*in ReadCondition a\_condition* - a reference to a ReadCondition or QueryCondition which filters the data before it is returned by the read operation.

## Return Value

*int* - 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* - one or more of the received\_data, info\_seq and a\_condition parameters is an invalid reference 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* - the max\_samples > length and max\_samples is not LENGTH\_UNLIMITED
- *RETCODE\_NO\_DATA* - no samples that meet the constraints are available.

### 3.5.2.74 take\_next\_sample

#### Scope

SPACE.FooDataReader

#### Synopsis

```
import DDS.*;
public int
    take_next_sample
        (FooHolder data_value,
         SampleInfoHolder 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

```
import DDS.*;
public int
    take_w_condition
        (FooSeqHolder data_values,
         SampleInfoSeqHolder info_seq,
         int max_samples,
         ReadCondition 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 FooSeqHolder data\_values* - the returned sample data sequence. *data\_values* is also used as an input to control the behaviour of this operation.

*inout SampleInfoSeqHolder info\_seq* - the returned SampleInfo object sequence. *info\_seq* is also used as an input to control the behaviour of this operation.

*in int max\_samples* - the maximum number of samples that is returned.

*in ReadCondition a\_condition* - a reference to a ReadCondition or QueryCondition which filters the data before it is returned by the read operation.

## Return Value

*int* - 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` - one or more of the `received_data`, `info_seq` and `a_condition` parameters is an invalid reference.
- `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` - the `max_samples > length` and `max_samples` is not `LENGTH_UNLIMITED`.
- `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 interface `DataReader` for further explanation.

### Synopsis

```
import DDS.*;
public int
    wait_for_historical_data
        (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 Class SampleInfo

The class `SampleInfo` represents the additional information that accompanies the data in each sample that is read or taken.

The interface description of this class is as follows:

```
public class SampleInfo
{
    public int sample_state;
    public int view_state;
    public int instance_state;
    public Time_t source_timestamp;
    public long instance_handle;
    public long publication_handle;
    public int disposed_generation_count;
    public int no_writers_generation_count;
    public int sample_rank;
    public int generation_rank;
    public int absolute_generation_rank;
    public boolean valid_data;
};
```

The next paragraph describes the usage of the `SampleInfo` struct.

#### 3.5.4.1 SampleInfo

##### Scope

DDS

##### Synopsis

```
import DDS.*;
public class SampleInfo
{
    public int sample_state;
    public int view_state;
    public int instance_state;
    public Time_t source_timestamp;
    public long instance_handle;
    public long publication_handle;
    public int disposed_generation_count;
    public int no_writers_generation_count;
```

```

    public int sample_rank;
    public int generation_rank;
    public int absolute_generation_rank;
    public boolean valid_data;
};

```

## Description

The class `SampleInfo` represents the additional information that accompanies the data in each sample that is read or taken.

## Attributes

*int sample\_state* - whether or not the corresponding data sample has already been read.

*int view\_state* - whether the `DataReader` has already seen samples of the most-current generation of the related instance.

*int 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.

*long instance\_handle* - the handle that identifies locally the corresponding instance.

*long publication\_handle* - the handle that identifies locally the `DataWriter` that modified the instance. In fact it is an `instance_handle` of the built-in `DCPSPublication` sample that describes this `DataWriter`. It can be used as a parameter to the `DataReader` operation `get_matched_publication_data` to obtain this built-in `DCPSPublication` sample.

*int disposed\_generation\_count* - the number of times the instance has become alive after it was disposed of explicitly by a `DataWriter`.

*int no\_writers\_generation\_count* - the number of times the instance has become alive after it was disposed of because there were no `DataWriter` objects.

*int sample\_rank* - the number of samples related to the same instance that are found in the collection returned by a `read` or `take` operation.

*int generation\_rank* - the generation difference between the time the sample was received and the time the most recent sample in the collection was received.

*int 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 class *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` is used by the DDS implementation to locally identify the corresponding source `DataWriter`. You can access more detailed information about this particular publication by passing its `publication_handle` to either the `get_matched_publication_data` operation on the `DataReader` or to the `read_instance` operation on the built-in reader for the “DCPSPublication” topic.



Be aware that since an instance handle is an opaque datatype, it does not necessarily mean that the handle obtained from the `publication_handle` has the same value as the one that appears in the `instance_handle` field of the `SampleInfo` when retrieving the publication info through corresponding “DCPSPublication” built-in reader. You can’t just compare two handles to determine whether they represent the same publication. If you want to know whether two handles actually do represent the same publication, use both handles to retrieve their corresponding `PublicationBuiltinTopicData` samples and then compare the key field of both samples.

- `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 implement the `SubscriberListener` interface.




---

*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.*

---

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 is as follows:

```
public interface SubscriberListener
{
    //
    // extends interface DataReaderListener
    //
    // public void
    //     on_requested_deadline_missed
    //     (DataReader reader,
    //      RequestedDeadlineMissedStatus status);

    // public void
    //     on_requested_incompatible_qos
    //     (DataReader reader,
    //      RequestedIncompatibleQosStatus status);

    // public void
    //     on_sample_rejected
    //     (DataReader reader,
    //      SampleRejectedStatus status);

    // public void
    //     on_liveliness_changed
```

```

//      (DataReader reader,
//      LivelinessChangedStatus status);

// public void
//      on_data_available
//      (DataReader reader);

// public void
//      on_subscription_matched
//      (DataReader reader,
//      SubscriptionMatchedStatus status);

// public void
//      on_sample_lost
//      (DataReader reader,
//      SampleLostStatus status);
//
// external operations
//
//      public void
//          on_data_on_readers
//          (Subscriber subs);
//
// implemented API operations
//      <no operations>
//
};

```

The following 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)

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

#### Synopsis

```

import DDS.*;
public void
    on_data_available
        (DataReader reader);

```

### 3.5.5.2 on\_data\_on\_readers

#### Scope

```

DDS.SubscriberListener

```

## Synopsis

```
import DDS.*;
public void
    on_data_on_readers
        (Subscriber subs);
```

## 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 subs* - contain a reference 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)

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

#### Synopsis

```
import DDS.*;
public void
    on_liveliness_changed
        (DataReader reader,
         LivelinessChangedStatus status);
```

### 3.5.5.4 on\_requested\_deadline\_missed (inherited)

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

#### Synopsis

```
import DDS.*;
public void
    on_requested_deadline_missed
        (DataReader reader,
         RequestedDeadlineMissedStatus status);
```

### 3.5.5.5 on\_requested\_incompatible\_qos (inherited)

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

#### Synopsis

```
import DDS.*;
public void]
    on_requested_incompatible_qos
        (DataReader reader,
         RequestedIncompatibleQosStatus status);
```

### 3.5.5.6 on\_sample\_lost (inherited)

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

#### Synopsis

```
import DDS.*;
public void
    on_sample_lost
        (DataReader reader,
         SampleLostStatus status);
```

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



### 3.5.5.7 on\_sample\_rejected (inherited)

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

#### Synopsis

```
import DDS.*;
public void
    on_sample_rejected
        (DataReader reader,
         SampleRejectedStatus status);
```

### 3.5.5.8 on\_subscription\_matched (inherited,)

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

#### Synopsis

```
import DDS.*;
public void
    on_subscription_matched
        (DataReader reader,
         SubscriptionMatchedStatus status);
```

## 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 implement the `DataReaderListener` interface. **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:

```
public interface DataReaderListener
{
    //
```

```

// external operations
//
    public void
        on_requested_deadline_missed
            (DataReader reader,
             RequestedDeadlineMissedStatus status);
    public void
        on_requested_incompatible_qos
            (DataReader reader,
             RequestedIncompatibleQosStatus status);

    public void
        on_sample_rejected
            (DataReader reader,
             SampleRejectedStatus status);

    public void
        on_liveliness_changed
            (DataReader reader,
             LivelinessChangedStatus status);

    public void
        on_data_available
            (DataReader reader);

    public public void
        on_subscription_matched
            (DataReader reader,
             SubscriptionMatchedStatus status);

    public void
        on_sample_lost
            (DataReader reader,
             SampleLostStatus status);
//
// implemented API operations
// <no operations>
//
};

```

The following 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

#### Scope

DDS.DataReaderListener

## Synopsis

```
import DDS.*;
public void
    on_data_available
        (DataReader reader);
```

## 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 reader* - contain a reference 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

#### Scope

DDS.DataReaderListener

#### Synopsis

```
import DDS.*;
public void
    on_liveliness_changed
        (DataReader reader,
         LivelinessChangedStatus status);
```

#### 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 reader* - contain a reference 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 LivelinessChangedStatus status* - contain the LivelinessChangedStatus object (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 object for use by the application.

### 3.5.6.3 on\_requested\_deadline\_missed

#### Scope

DDS.DataReaderListener

#### Synopsis

```
import DDS.*;
public void
    on_requested_deadline_missed
        (DataReader reader,
         RequestedDeadlineMissedStatus status);
```

#### 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 reader* - contain a reference to the `DataReader` for which the deadline was missed (this is an input to the application provided by the Data Distribution Service).

*in RequestedDeadlineMissedStatus status* - contain the `RequestedDeadlineMissedStatus` object (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` object in the parameter `status` for use by the application.

### 3.5.6.4 on\_requested\_incompatible\_qos

#### Scope

DDS.DataReaderListener

#### Synopsis

```
import DDS.*;
public void
    on_requested_incompatible_qos
        (DataReader reader,
         RequestedIncompatibleQosStatus status);
```

#### 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 reader* - a reference to the DataReader provided by the Data Distribution Service.

*in RequestedIncompatibleQosStatus status* - the RequestedIncompatibleQosStatus object 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 object 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`

#### Scope

`DDS.DataReaderListener`

#### Synopsis

```
import DDS.*;
public void
    on_sample_lost
        (DataReader reader,
         SampleLostStatus status);
```

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

### 3.5.6.6 `on_sample_rejected`

#### Scope

`DDS.DataReaderListener`

#### Synopsis

```
import DDS.*;
public void
    on_sample_rejected
        (DataReader reader,
         SampleRejectedStatus status);
```

#### 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 reader` - contain a reference to the `DataReader` for which a sample has been rejected (this is an input to the application provided by the Data Distribution Service).
- in* `SampleRejectedStatus status` - contain the `SampleRejectedStatus` object (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. Samples may be rejected by the `DataReader` when it runs out of `resource_limits` to store incoming samples. Usually this means that old samples need to be ‘consumed’ (for example by ‘taking’ them instead of ‘reading’ them) to make room for newly incoming samples.

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` object in the parameter `status` for use by the application.

### 3.5.6.7 on\_subscription\_matched (abstract)

#### Scope

`DDS.DataReaderListener`

#### Synopsis

```
import DDS.*;
public void
    on_subscription_matched
        (DataReader reader,
         SubscriptionMatchedStatus status);
```

#### Description

This operation must be implemented by the application and is called by the Data Distribution Service when a new match has been discovered for the current subscription, or when an existing match has ceased to exist.

#### Parameters

*in DataReader reader* - contains a reference to the `DataReader` for which a match has been discovered (this is an input to the application provided by the Data Distribution Service).

*in SubscriptionMatchedStatus status* - contains the `SubscriptionMatchedStatus` object (this is an input to the application provided by the Data Distribution Service).



## Return Value

<none>

## Detailed Description

This operation must be implemented by the application and is called by the Data Distribution Service when a new match has been discovered for the current subscription, or when an existing match has ceased to exist. Usually this means that a new `DataWriter` that matches the `Topic` and that has compatible `Qos` as the current `DataReader` has either been discovered, or that a previously discovered `DataWriter` has ceased to be matched to the current `DataReader`. A `DataWriter` may cease to match when it gets deleted, when it changes its `Qos` to a value that is incompatible with the current `DataReader` or when either the `DataReader` or the `DataWriter` has chosen to put its matching counterpart on its ignore-list using the `ignore_publication` or `ignore_subscription` operations on the `DomainParticipant`.

The implementation of this Listener operation may be left empty when this functionality is not needed: it will only be called when the relevant `DataReaderListener` is installed and enabled for the `SUBSCRIPTION_MATCHED_STATUS`.

The Data Distribution Service will provide a reference to the `DataReader` in the parameter `reader` and the `SubscriptionMatchedStatus` object in the parameter `status` for use by the application.

### 3.5.7 Interface 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:

```
public interface ReadCondition
{
//
// extends to interface Condition
//
```

```

// boolean
//     get_trigger_value
//         (void);
//
// implemented API operations
//
    int
        get_sample_state_mask
            (void);

    int
        get_view_state_mask
            (void);

    int
        get_instance_state_mask
            (void);

    DataReader
        get_datareader
            (void);
};

```

The following 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

```

import DDS.*;
public DataReader
    get_datareader
        (void);

```

#### Description

This operation returns the `DataReader` associated with the `ReadCondition`.

#### Parameters

<none>

#### Return Value

*DataReader* - Result value is a reference 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

```
import DDS.*;
public int
    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

*int* - 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

```
import DDS.*;
```

```
public int
    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 interface `Condition` for further explanation.

### Synopsis

```
import DDS.*;
public boolean
    get_trigger_value
        (void);
```

#### 3.5.7.5 `get_view_state_mask`

### Scope

```
DDS.ReadCondition
```

### Synopsis

```
import DDS.*;
public int
    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 Interface 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 parameterised by arguments that are dynamically changeable with the `set_query_parameters` operation.

The interface description is as follows:

```
public interface QueryCondition
{
    //
```

```

// extends interface ReadCondition
//
// public int
//     get_sample_state_mask
//         (void);

// public int
//     get_view_state_mask
//         (void);

// public int
//     get_instance_state_mask
//         (void);

// public DataReader
//     get_datareader
//         (void);
// public boolean
//     get_trigger_value
//         (void);
//
// implemented API operations
//
    public String
        get_query_expression
            (void);

    public int
        get_query_parameters
            (StringSeqHolder query_parameters);

    public int
        set_query_parameters
            (String[] query_parameters);
};

```

The following 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 interface `ReadCondition` for further explanation.

#### Synopsis

```

import DDS.*;
public DataReader

```

```
get_datareader
(void);
```

### 3.5.8.2 `get_instance_state_mask` (inherited)

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

#### Synopsis

```
import DDS.*;
public int
    get_instance_state_mask
        (void);
```

### 3.5.8.3 `get_query_parameters`

#### Scope

```
DDS.QueryCondition
```

#### Synopsis

```
import DDS.*;
public int
    get_query_parameters
        (StringSeqHolder query_parameters);
```

#### Description

This operation obtains the `query_parameters` associated with the `QueryCondition`.

#### Parameters

*inout StringSeqHolder query\_parameters* - a reference to the destination `StringSeqHolder` object in which the the parameters used in the SQL expression will be copied.

#### Return Value

*int* - 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 Holder 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

```
import DDS.*;
public String
    get_query_expression
        (void);
```

#### Description

This operation returns the query expression associated with the `QueryCondition`.

#### Parameters

<none>

#### Return Value

*String* - 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 interface `ReadCondition` for further explanation.

#### Synopsis

```
import DDS.*;
public int
    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

```
import DDS.*;
public 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

```
import DDS.*;
public int
    get_view_state_mask
        (void);
```

#### 3.5.8.8 `set_query_parameters`

#### Scope

```
DDS.QueryCondition
```

#### Synopsis

```
import DDS.*;
```

```
public int
    set_query_arguments
        (String[] parameters);
```

## Description

This operation changes the query parameters associated with the QueryCondition.

## Parameters

*in String[] 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

*int* - 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 parameters 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 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.

# APPENDICES





# Quality Of Service

Each Entity is accompanied by an `<Entity>Qos` class that implements the basic mechanism for an application to specify Quality of Service attributes. This class consists of Entity specific `QosPolicy` attributes. `QosPolicy` attributes are classes where each class specifies the information that controls an Entity related (configurable) attribute of the Data Distribution Service. A `QosPolicy` attribute class 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 class of `QosPolicy` classes, identified as `<Entity>Qos`. Each `<Entity>Qos` class only contains those `QosPolicy` classes relevant to the specific Entity. The `<Entity>Qos` class serves as the parameter to operations which require a `Qos`. `<Entity>Qos` class 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`<sup>1</sup>.

## 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 class is as shown below.

```
//public class <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`.

```
//
//public class <Entity>Qos
//
public class DomainParticipantFactoryQos
{
    public EntityFactoryQosPolicy entity_factory;
};

public class DomainParticipantQos
{
    public UserDataQosPolicy user_data;
    public EntityFactoryQosPolicy entity_factory;
    public SchedulingQosPolicy watchdog_scheduling;
    public SchedulingQosPolicy listener_scheduling;
};

public class TopicQos
{
    public TopicDataQosPolicy topic_data;
    public DurabilityQosPolicy durability;
    public DurabilityServiceQosPolicy durability_service;
    public DeadlineQosPolicy deadline;
    public LatencyBudgetQosPolicy latency_budget;
    public LivelinessQosPolicy liveliness;
    public ReliabilityQosPolicy reliability;
    public DestinationOrderQosPolicy destination_order;
    public HistoryQosPolicy history;
    public ResourceLimitsQosPolicy resource_limits;
    public TransportPriorityQosPolicy transport_priority;
    public LifespanQosPolicy lifespan;
    public OwnershipQosPolicy ownership;
};

public class DataWriterQos
{
    public DurabilityQosPolicy durability;
    public DeadlineQosPolicy deadline;
    public LatencyBudgetQosPolicy latency_budget;
    public LivelinessQosPolicy liveliness;
    public ReliabilityQosPolicy reliability;
    public DestinationOrderQosPolicy destination_order;
    public HistoryQosPolicy history;
    public ResourceLimitsQosPolicy resource_limits;
    public TransportPriorityQosPolicy transport_priority;
    public LifespanQosPolicy lifespan;
    public UserDataQosPolicy user_data;
    public OwnershipQosPolicy ownership;
    public OwnershipStrengthQosPolicy ownership_strength;
    public WriterDataLifecycleQosPolicy
        writer_data_lifecycle;
};
```

```
};

public class PublisherQos
{
    public PresentationQosPolicy presentation;
    public PartitionQosPolicy partition;
    public GroupDataQosPolicy group_data;
    public EntityFactoryQosPolicy entity_factory; };

public class DataReaderQos
{
    public DurabilityQosPolicy durability;
    public DeadlineQosPolicy deadline;
    public LatencyBudgetQosPolicy latency_budget;
    public LivelinessQosPolicy liveliness;
    public ReliabilityQosPolicy reliability;
    public DestinationOrderQosPolicy destination_order;
    public HistoryQosPolicy history;
    public ResourceLimitsQosPolicy resource_limits;
    public UserDataQosPolicy user_data;
    public OwnershipQosPolicy ownership;
    public TimeBasedFilterQosPolicy time_based_filter;
    public ReaderDataLifecycleQosPolicy
        reader_data_lifecycle;
};

public class SubscriberQos
{
    public PresentationQosPolicy presentation;
    public PartitionQosPolicy partition;
    public GroupDataQosPolicy group_data;
    public EntityFactoryQosPolicy entity_factory; };

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

The following paragraphs describe the usage of each <Entity>Qos class.

## DataReaderQos

### Scope

DDS

### Synopsis

```
import DDS.*;
public class DataReaderQos
{
    public DurabilityQosPolicy durability;
    public DeadlineQosPolicy deadline;
```

```

    public LatencyBudgetQosPolicy latency_budget;
    public LivelinessQosPolicy liveliness;
    public ReliabilityQosPolicy reliability;
    public DestinationOrderQosPolicy destination_order;
    public HistoryQosPolicy history;
    public ResourceLimitsQosPolicy resource_limits;
    public UserDataQosPolicy user_data;
    public OwnershipQosPolicy ownership;
    public TimeBasedFilterQosPolicy time_based_filter;
    public ReaderDataLifecycleQosPolicy
        reader_data_lifecycle;
};

```

## Description

This class provides the basic mechanism for an application to specify Quality of Service attributes for a DataReader.

## Attributes

*public DurabilityQosPolicy durability* - whether the data should be stored for late joining readers.

*public DeadlineQosPolicy deadline* - the period within which a new sample is expected.

*public LatencyBudgetQosPolicy latency\_budget* - used by the Data Distribution Service for optimization.

*public LivelinessQosPolicy liveliness* - the way the liveliness of the DataReader is asserted to the Data Distribution Service.

*public ReliabilityQosPolicy reliability* - the reliability of the data distribution.

*public DestinationOrderQosPolicy destination\_order* - the order in which the DataReader timely orders the data.

*public HistoryQosPolicy history* - how samples should be stored.

*public ResourceLimitsQosPolicy resource\_limits* - the maximum amount of resources to be used.

*public UserDataQosPolicy user\_data* - used to attach additional information to the DataReader.

*public OwnershipQosPolicy ownership* - whether a DataWriter exclusively owns an instance.

*public TimeBasedFilterQosPolicy time\_based\_filter* - the maximum data rate at which the DataReader will receive changes.



`public ReaderDataLifecycleQosPolicy reader_data_lifecycle` - the minimum time after which a data instance is disposed of when the `instance_state` has become `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` object 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`.

See Section 3.1.3, *Class QosPolicy*, on page 35 for a list of all `<name>QosPolicy` settings, their meaning, characteristics, possible values and applicability to `DataReaders`.

The initial value of the default `DataReaderQos` in the `Subscriber` are given in the following table:

**Table 16 DATAREADER\_QOS\_DEFAULT**

QosPolicy	Attribute	Value
durability	kind	VOLATILE_DURABILITY_QOS
	service_cleanup_delay	0
deadline	period	DURATION_INFINITY
latency_budget	duration	0
liveliness	kind	AUTOMATIC_LIVELINESS_QOS
	lease_duration	DURATION_INFINITY
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 DATAREADER\_QOS\_DEFAULT (Continued)**

QosPolicy	Attribute	Value
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_	DURATION_INFINITE
	nowriter_samples_delay	
	autopurge_disposed_samples_delay	DURATION_INFINITE

## DataWriterQos

### Scope

DDS

### Synopsis

```
import DDS.*;

public class DataReaderQos
{
    public DurabilityQosPolicy durability;
    public DeadlineQosPolicy deadline;
    public LatencyBudgetQosPolicy latency_budget;
    public LivelinessQosPolicy liveliness;
    public ReliabilityQosPolicy reliability;
    public DestinationOrderQosPolicy destination_order;
    public HistoryQosPolicy history;
    public ResourceLimitsQosPolicy resource_limits;
    public TransportPriorityQosPolicy transport_priority;
    public LifespanQosPolicy lifespan;
    public UserDataQosPolicy user_data;
    public OwnershipQosPolicy ownership;
    public OwnershipStrengthQosPolicy ownership_strength;
    public WriterDataLifecycleQosPolicy
writer_data_lifecycle;
```

```
};
```

## Description

This class provides the basic mechanism for an application to specify Quality of Service attributes for a `DataWriter`.

## Attributes

*public DurabilityQosPolicy durability* - whether the data should be stored for late joining readers.

*public DeadlineQosPolicy deadline* - the period within which a new sample is written.

*public LatencyBudgetQosPolicy latency\_budget* - used by the Data Distribution Service for optimization.

*public LivelinessQosPolicy liveliness* - the way the liveliness of the `DataSource` is asserted to the Data Distribution Service.

*public ReliabilityQosPolicy reliability* - the reliability of the data distribution.

*public DestinationOrderQosPolicy destination\_order* - the order in which the `DataReader` timely orders the data.

*public HistoryQosPolicy history* - how samples should be stored.

*public ResourceLimitsQosPolicy resource\_limits* - the maximum amount of resources to be used.

*public TransportPriorityQosPolicy transport\_priority* - a priority hint for the underlying transport layer.

*public LifespanQosPolicy lifespan* - the maximum duration of validity of the data written by the `DataSource`.

*public UserDataQosPolicy user\_data* - used to attach additional information to the `DataSource`.

*public OwnershipQosPolicy ownership* - whether a `DataSource` exclusively owns an instance.

*public OwnershipStrengthQosPolicy ownership\_strength* - the strength to determine the ownership.

*public 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` object 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 17 DATAWRITER\_QOS\_DEFAULT**

QosPolicy	Attribute	Value
durability	kind	VOLATILE_DURABILITY_QOS
	service_cleanup_delay	0
deadline	period	DURATION_INFINITE
latency_budget	duration	0
liveliness	kind	AUTOMATIC_LIVELINESS_QOS
	lease_duration	DURATION_INFINITY
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
user_data	value.length	0

**Table 17 DATAWRITER\_QOS\_DEFAULT (Continued)**

QosPolicy	Attribute	Value
ownership	kind	SHARED_OWNERSHIP_QOS
ownership_strength	value	0
writer_data_lifecycle	autodispose_ unregistered_instances	true

## DomainParticipantFactoryQos

### Scope

DDS

### Synopsis

```
import DDS.*;
public class DomainParticipantFactoryQos
{ public EntityFactoryQosPolicy entity_factory; };
```

### Description

This class 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 18 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` class as a parameter.

## DomainParticipantQos

### Scope

DDS

### Synopsis

```
import DDS.*;
public class DomainParticipantQos
{
    public UserDataQosPolicy user_data;
    public EntityFactoryQosPolicy entity_factory;
    public SchedulingQosPolicy watchdog_scheduling;
    public SchedulingQosPolicy listener_scheduling;
};
```

### Description

This class provides the basic mechanism for an application to specify Quality of Service attributes for a DomainParticipant.

### Attributes

*public UserDataQosPolicy user\_data* - used to attach additional information to the DomainParticipant.

*public 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 object 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 19 PARTICIPANT\_QOS\_DEFAULT**

QosPolicy	Attribute	Value
<code>user_data</code>	<code>value.length</code>	0
<code>entity_factory</code>	<code>autoenable_created_entities</code>	true
<code>watchdog_scheduling</code>	<code>scheduling_class.kind</code>	SCHEDULE_DEFAULT
	<code>scheduling_priority_kind.kind</code>	PRIORITY_RELATIVE
	<code>scheduling_priority</code>	0
<code>listener_scheduling</code>	<code>scheduling_class.kind</code>	SCHEDULE_DEFAULT
	<code>scheduling_priority_kind.kind</code>	PRIORITY_RELATIVE
	<code>scheduling_priority</code>	0

## PublisherQos

### Scope

DDS

### Synopsis

```
import DDS.*;
public class PublisherQos
{
    public PresentationQosPolicy presentation;
    public PartitionQosPolicy partition;
    public GroupDataQosPolicy group_data;
    public EntityFactoryQosPolicy entity_factory; };
```

### Description

This class provides the basic mechanism for an application to specify Quality of Service attributes for a Publisher.

### Attributes

*public PresentationQosPolicy presentation* - the dependency of changes to data-instances.

*public PartitionQosPolicy partition* - the partitions in which the Publisher is active.

*public GroupDataQosPolicy group\_data* - used to attach additional information to the Publisher.

*public 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` object 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 20 PUBLISHER\_QOS\_DEFAULT**

QosPolicy	Attribute	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

```
import DDS.*;
public class SubscriberQos
{
    public PresentationQosPolicy presentation;
    public PartitionQosPolicy partition;
```



```
public GroupDataQosPolicy group_data;
public EntityFactoryQosPolicy entity_factory; };
```

## Description

This class provides the basic mechanism for an application to specify Quality of Service attributes for a Subscriber.

## Attributes

*public PresentationQosPolicy presentation* - the dependency of changes to data-instances.

*public PartitionQosPolicy partition* - the partitions in which the Subscriber is active.

*public GroupDataQosPolicy group\_data* - used to attach additional information to the Subscriber.

*public 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 object 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 21 SUBSCRIBER\_QOS\_DEFAULT**

QosPolicy	Attribute	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

```
import DDS.*;
public class TopicQos
{
    public TopicDataQosPolicy topic_data;
    public DurabilityQosPolicy durability;
    public DurabilityServiceQosPolicy durability_service;
    public DeadlineQosPolicy deadline;
    public LatencyBudgetQosPolicy latency_budget;
    public LivelinessQosPolicy liveliness;
    public ReliabilityQosPolicy reliability;
    public DestinationOrderQosPolicy destination_order;
    public HistoryQosPolicy history;
    public ResourceLimitsQosPolicy resource_limits;
    public TransportPriorityQosPolicy transport_priority;
    public LifespanQosPolicy lifespan;
    public OwnershipQosPolicy ownership; };
```

## Description

This class provides the basic mechanism for an application to specify Quality of Service attributes for a Topic.

## Attributes

*public TopicDataQosPolicy topic\_data* - used to attach additional information to the Topic.

*public DurabilityQosPolicy durability* - whether the data should be stored for late joining readers.

*public DurabilityServiceQosPolicy durability\_service* - the behaviour of the “transient/persistent service” of the Data Distribution System regarding Transient and Persistent Topic instances.

*public DeadlineQosPolicy deadline* - the period within which a new sample is expected or written.

*public LatencyBudgetQosPolicy latency\_budget* - used by the Data Distribution Service for optimization.

*public LivelinessQosPolicy liveliness* - the way the liveliness of the Topic is asserted to the Data Distribution Service.

*public ReliabilityQosPolicy reliability* - the reliability of the data distribution.

*public DestinationOrderQosPolicy destination\_order* - the order in which the DataReader timely orders the data.

*public HistoryQosPolicy history* - how samples should be stored.

*public ResourceLimitsQosPolicy resource\_limits* - the maximum amount of resources to be used.

*public TransportPriorityQosPolicy transport\_priority* - a priority hint for the underlying transport layer.

*public LifespanQosPolicy lifespan* - the maximum duration of validity of the data written by a DataWriter.

*public 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` object 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 22 TOPIC\_QOS\_DEFAULT**

QosPolicy	Attribute	Value
topic_data	value.length	0
durability	kind	VOLATILE_DURABILITY_QOS
	service_cleanup_delay	0

**Table 22 TOPIC\_QOS\_DEFAULT (Continued)**

<b>QosPolicy</b>	<b>Attribute</b>	<b>Value</b>
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*

## Duration and Time

```
package DDS;
public final class Duration_t
{
    public int sec = (int)0;
    public int nanosec = (int)0;
    public Duration_t ()
    {
    }
    public Duration_t (int _sec, int _nanosec)
    {
        sec = _sec;
        nanosec = _nanosec;
    }
} // class Duration_t
public interface DURATION_INFINITE_SEC
{
    public static final int value = (int)(0x7fffffff);
}
public interface DURATION_INFINITE_NSEC
{
    public static final int value = (int)(0x7ffffffffL);
}
public interface DURATION_ZERO_SEC
{
    public static final int value = (int)(0);
}
public interface DURATION_ZERO_NSEC
{
    public static final int value = (int)(0L);
}
public final class Time_t
{
    public int sec = (int)0;
    public int nanosec = (int)0;
    public Time_t ()
    {
    }
    public Time_t (int _sec, int _nanosec)
    {
        sec = _sec;
```

```
        nanosec = _nanosec;
    }
} // class Time_t
```

## Pre-defined Values

```
public interface HANDLE_NIL
{
    public static final long value = (long)(0L);
}
public interface LENGTH_UNLIMITED
{
    public static final int value = (int)((int)-1);
}
public interface TIMESTAMP_INVALID_SEC
{
    public static final int value = (int)((int)-1);
}
public interface TIMESTAMP_INVALID_NSEC
{
    public static final int value = (int)(0xffffffffL);
}
```

## Return Codes

```
public interface RETCODE_OK
{
    public static final int value = (int)(0);
}
public interface RETCODE_ERROR
{
    public static final int value = (int)(1);
}
public interface RETCODE_UNSUPPORTED
{
    public static final int value = (int)(2);
}
public interface RETCODE_BAD_PARAMETER
{
    public static final int value = (int)(3);
}
public interface RETCODE_PRECONDITION_NOT_MET
{
    public static final int value = (int)(4);
}
public interface RETCODE_OUT_OF_RESOURCES
{
    public static final int value = (int)(5);
}
public interface RETCODE_NOT_ENABLED
{

```

```

        public static final int value = (int)(6);
    }
    public interface RETCODE_IMMUTABLE_POLICY
    {
        public static final int value = (int)(7);
    }
    public interface RETCODE_INCONSISTENT_POLICY
    {
        public static final int value = (int)(8);
    }
    public interface RETCODE_ALREADY_DELETED
    {
        public static final int value = (int)(9);
    }
    public interface RETCODE_TIMEOUT
    {
        public static final int value = (int)(10);
    }
    public interface RETCODE_NO_DATA
    {
        public static final int value = (int)(11);
    }
    public interface RETCODE_ILLEGAL_OPERATION
    {
        public static final int value = (int)(12);
    }

```

## Status to Support Listeners and Conditions

```

    public interface INCONSISTENT_TOPIC_STATUS
    {
        public static final int value = (int)((int)(0x0001L << 0L));
    }
    public interface OFFERED_DEADLINE_MISSED_STATUS
    {
        public static final int value = (int)((int)(0x0001L << 1L));
    }
    public interface REQUESTED_DEADLINE_MISSED_STATUS
    {
        public static final int value = (int)((int)(0x0001L << 2L));
    }
    public interface OFFERED_INCOMPATIBLE_QOS_STATUS
    {
        public static final int value = (int)((int)(0x0001L << 5L));
    }
    public interface REQUESTED_INCOMPATIBLE_QOS_STATUS
    {
        public static final int value = (int)((int)(0x0001L << 6L));
    }
    public interface SAMPLE_LOST_STATUS

```

```

{
    public static final int value = (int)((int)(0x0001L << 7L));
}
public interface SAMPLE_REJECTED_STATUS
{
    public static final int value = (int)((int)(0x0001L << 8L));
}
public interface DATA_ON_READERS_STATUS
{
    public static final int value = (int)((int)(0x0001L << 9L));
}
public interface DATA_AVAILABLE_STATUS
{
    public static final int value = (int)((int)(0x0001L << 10L));
}
public interface LIVELINESS_LOST_STATUS
{
    public static final int value = (int)((int)(0x0001L << 11L));
}
public interface LIVELINESS_CHANGED_STATUS
{
    public static final int value = (int)((int)(0x0001L << 12L));
}
public interface PUBLICATION_MATCH_STATUS
{
    public static final int value = (int)((int)(0x0001L << 13L));
}
public interface SUBSCRIPTION_MATCH_STATUS
{
    public static final int value = (int)((int)(0x0001L << 14L));
}
/* Note: ANY_STATUS is deprecated, please use spec version
 * specific constants.  e.g. STATUS_MASK_ANY_V1_2
 */
public interface ANY_STATUS
{
    public static final int value = (int)(0x7FE7L);
}

public interface STATUS_MASK_ANY_V1_2
{
    public static final int value = (int)(0x7FE7L);
}

public interface STATUS_MASK_NONE
{
    public static final int value = (int)(0x0L);
}

```



## States

### Sample States to Support Reads

```
public interface READ_SAMPLE_STATE
{
    public static final int value = (int)((int)(0x0001L << 0L));
}
public interface NOT_READ_SAMPLE_STATE
{
    public static final int value = (int)((int)(0x0001L << 1L));
}
// This is a bit mask SampleStateKind
public interface ANY_SAMPLE_STATE
{
    public static final int value = (int)(0xffffL);
}
```

### View States to Support Reads

```
public interface NEW_VIEW_STATE
{
    public static final int value = (int)((int)(0x0001L << 0L));
}
public interface NOT_NEW_VIEW_STATE
{
    public static final int value = (int)((int)(0x0001L << 1L));
}
// This is a bit mask ViewStateKind
public interface ANY_VIEW_STATE
{
    public static final int value = (int)(0xffffL);
}
```

### Instance States to Support Reads

```
public interface ALIVE_INSTANCE_STATE
{
    public static final int value = (int)((int)(0x0001L << 0L));
}
public interface NOT_ALIVE_DISPOSED_INSTANCE_STATE
{
    public static final int value = (int)((int)(0x0001L << 1L));
}
public interface NOT_ALIVE_NO_WRITERS_INSTANCE_STATE
{
    public static final int value = (int)((int)(0x0001L << 2L));
}
// This is a bit mask InstanceStateKind
public interface ANY_INSTANCE_STATE
{
}
```

```

        public static final int value = (int)(0xffffL);
    }
    public interface NOT_ALIVE_INSTANCE_STATE
    {
        public static final int value = (int)(0x006L);
    }

```

## QosPolicy

### Names

```

public interface USERDATA_QOS_POLICY_NAME
{
    public static final String value = "UserData";
}
public interface DURABILITY_QOS_POLICY_NAME
{
    public static final String value = "Durability";
}
public interface PRESENTATION_QOS_POLICY_NAME
{
    public static final String value = "Presentation";
}
public interface DEADLINE_QOS_POLICY_NAME
{
    public static final String value = "Deadline";
}
public interface LATENCYBUDGET_QOS_POLICY_NAME
{
    public static final String value = "LatencyBudget";
}
public interface OWNERSHIP_QOS_POLICY_NAME
{
    public static final String value = "Ownership";
}
public interface OWNERSHIPSTRENGTH_QOS_POLICY_NAME
{
    public static final String value = "OwnershipStrength";
}
public interface LIVELINESS_QOS_POLICY_NAME
{
    public static final String value = "Liveliness";
}
public interface TIMEBASEDFILTER_QOS_POLICY_NAME
{
    public static final String value = "TimeBasedFilter";
}
public interface PARTITION_QOS_POLICY_NAME
{
    public static final String value = "Partition";
}

```

```

    }
    public interface RELIABILITY_QOS_POLICY_NAME
    {
        public static final String value = "Reliability";
    }
    public interface DESTINATIONORDER_QOS_POLICY_NAME
    {
        public static final String value = "DestinationOrder";
    }
    public interface HISTORY_QOS_POLICY_NAME
    {
        public static final String value = "History";
    }
    public interface RESOURCELIMITS_QOS_POLICY_NAME
    {
        public static final String value = "ResourceLimits";
    }
    public interface ENTITYFACTORY_QOS_POLICY_NAME
    {
        public static final String value = "EntityFactory";
    }
    public interface WRITERDATALIFECYCLE_QOS_POLICY_NAME
    {
        public static final String value =
            "WriterDataLifecycle";
    }
    public interface READERDATALIFECYCLE_QOS_POLICY_NAME
    {
        public static final String value = "ReaderDataLifecycle";
    }
    public interface TOPICDATA_QOS_POLICY_NAME
    {
        public static final String value = "TopicData";
    }
    public interface GROUPDATA_QOS_POLICY_NAME
    {
        public static final String value = "GroupData";
    }
    public interface TRANSPORTPRIORITY_QOS_POLICY_NAME
    {
        public static final String value = "TransportPriority";
    }
    public interface LIFESPAN_QOS_POLICY_NAME
    {
        public static final String value = "Lifespan";
    }
    public interface DURABILITYSERVICE_QOS_POLICY_NAME
    {
        public static final String value = "DurabilityService";
    }
}

```

## Identifications

```

public interface INVALID_QOS_POLICY_ID
{
    public static final int value = (int)(0);
}
public interface USERDATA_QOS_POLICY_ID
{
    public static final int value = (int)(1);
}
public interface DURABILITY_QOS_POLICY_ID
{
    public static final int value = (int)(2);
}
public interface PRESENTATION_QOS_POLICY_ID
{
    public static final int value = (int)(3);
}
public interface DEADLINE_QOS_POLICY_ID
{
    public static final int value = (int)(4);
}
public interface LATENCYBUDGET_QOS_POLICY_ID
{
    public static final int value = (int)(5);
}
public interface OWNERSHIP_QOS_POLICY_ID
{
    public static final int value = (int)(6);
}
public interface OWNERSHIPSTRENGTH_QOS_POLICY_ID
{
    public static final int value = (int)(7);
}
public interface LIVELINESS_QOS_POLICY_ID
{
    public static final int value = (int)(8);
}
public interface TIMEBASEDFILTER_QOS_POLICY_ID
{
    public static final int value = (int)(9);
}
public interface PARTITION_QOS_POLICY_ID
{
    public static final int value = (int)(10);
}
public interface RELIABILITY_QOS_POLICY_ID
{
    public static final int value = (int)(11);
}

```

```
public interface DESTINATIONORDER_QOS_POLICY_ID
{
    public static final int value = (int)(12);
}
public interface HISTORY_QOS_POLICY_ID
{
    public static final int value = (int)(13);
}
public interface RESOURCELIMITS_QOS_POLICY_ID
{
    public static final int value = (int)(14);
}
public interface ENTITYFACTORY_QOS_POLICY_ID
{
    public static final int value = (int)(15);
}
public interface WRITERDATA_LIFECYCLE_QOS_POLICY_ID
{
    public static final int value = (int)(16);
}
public interface READERDATA_LIFECYCLE_QOS_POLICY_ID
{
    public static final int value = (int)(17);
}
public interface TOPICDATA_QOS_POLICY_ID
{
    public static final int value = (int)(18);
}
public interface GROUPDATA_QOS_POLICY_ID
{
    public static final int value = (int)(19);
}
public interface TRANSPORTPRIORITY_QOS_POLICY_ID
{
    public static final int value = (int)(20);
}
public interface LIFESPAN_QOS_POLICY_ID
{
    public static final int value = (int)(21);
}
public interface DURABILITYSERVICE_QOS_POLICY_ID
{
    public static final int value = (int)(22);
}
```





# *Platform Specific Model IDL Interface*

## **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_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 LivelinessChangedStatus {
    long alive_count;
    long not_alive_count;
    long alive_count_change;
    long not_alive_count_change;
    InstanceHandle_t last_publication_handle;
};

struct LivelinessChangedStatus {
    long active_count;
    long inactive_count;
    long active_count_change;
    long inactive_count_change;
};

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 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;
    };

```

## 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 ExtTopicListener : TopicListener {
    void
    on_all_data_disposed(in Topic the_topic);
};

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 {

};

interface ExtDomainParticipantListener : DomainParticipantListener,
    ExtTopicListener {

};

```

## 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= 0x0006;

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 WRITERDATALIFECYCLE_QOS_POLICY_NAME=
    "WriterDataLifecycle";
const string READERDATALIFECYCLE_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 WRITERDATALIFECYCLE_QOS_POLICY_ID = 16;
const QosPolicyId_t READERDATALIFECYCLE_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);

ReturnCode_t
delete_contained_entities();

```

```
};

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
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 {

stringget_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 StatusKindMask 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);

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_value,
//     inout SampleInfo sample_info);
//
// ReturnCode_t
// take_next_sample(
//     inout Data data_value,
//     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);
```

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_value,
    inout DDS::SampleInfo sample_info);

DDS::ReturnCode_t
take_next_sample(
    inout Foo data_value,

```

```

        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:

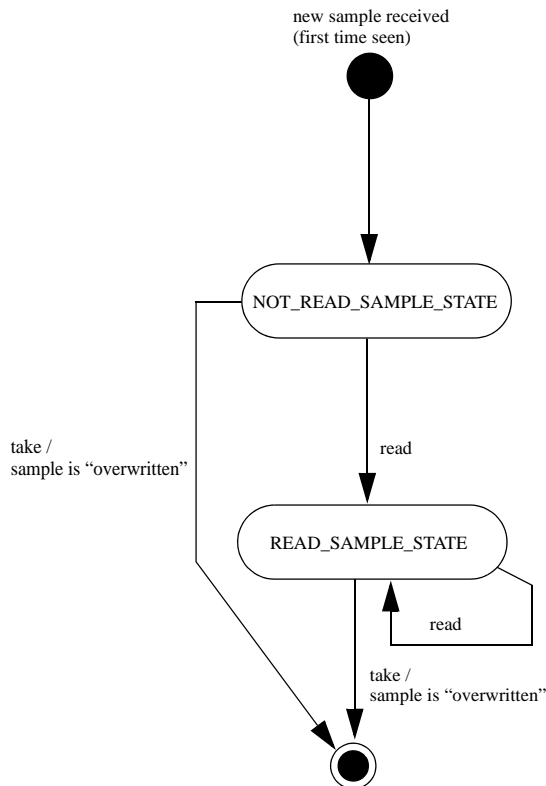
- The `sample_state` (`READ_SAMPLE_STATE` or `NOT_READ_SAMPLE_STATE`)
- The `view_state`, (`NEW_VIEW_STATE` or `NOT_NEW_VIEW_STATE`)
- The `instance_state` (`ALIVE_INSTANCE_STATE`, `NOT_ALIVE_DISPOSED_INSTANCE_STATE` or `NOT_ALIVE_NO_WRITERS_INSTANCE_STATE`)

## 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: sample\_state for a Single Sample 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:

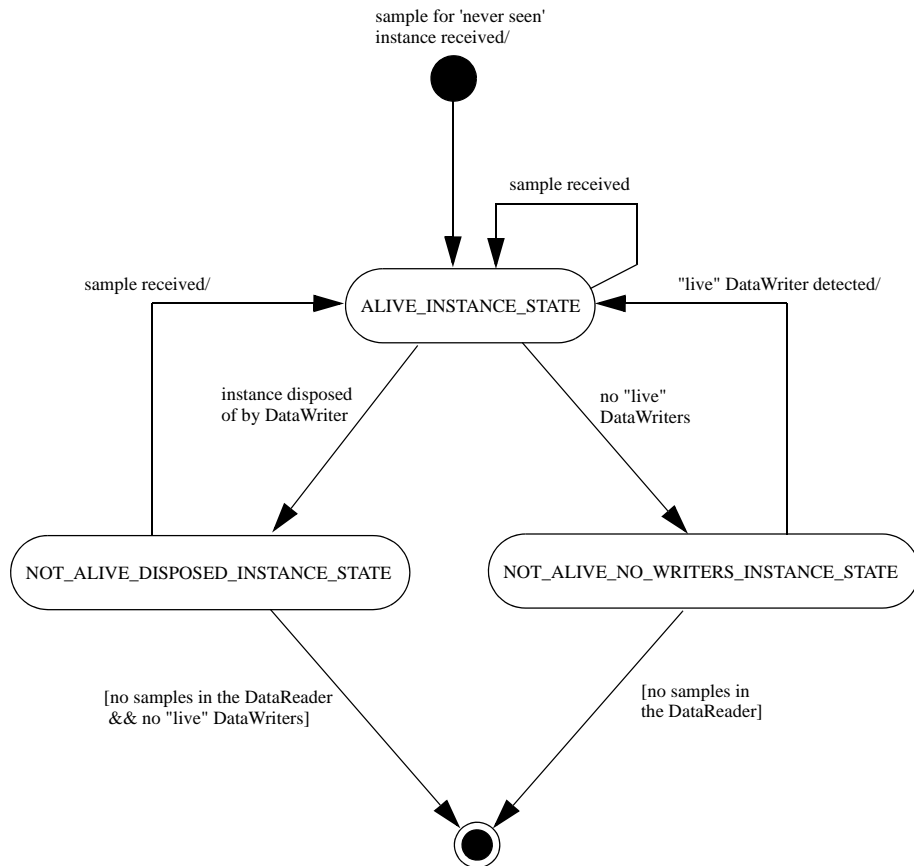
- `ALIVE_INSTANCE_STATE` indicates that
  - samples have been received for the instance
  - and there are live `DataWriter` objects writing the instance

- and 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**

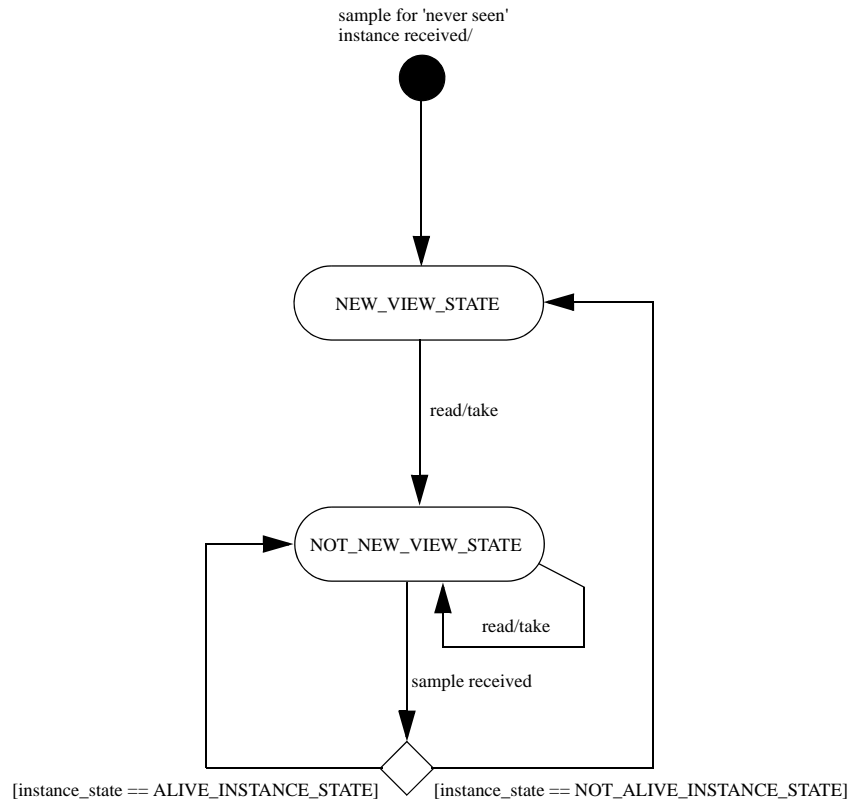
### Snapshot

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: view\_state for a Single Instance 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 (i.e., `READ_SAMPLE_STATE`, `NOT_READ_SAMPLE_STATE`, or `ANY_SAMPLE_STATE`)
- the desired view states (i.e., `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` objects 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**

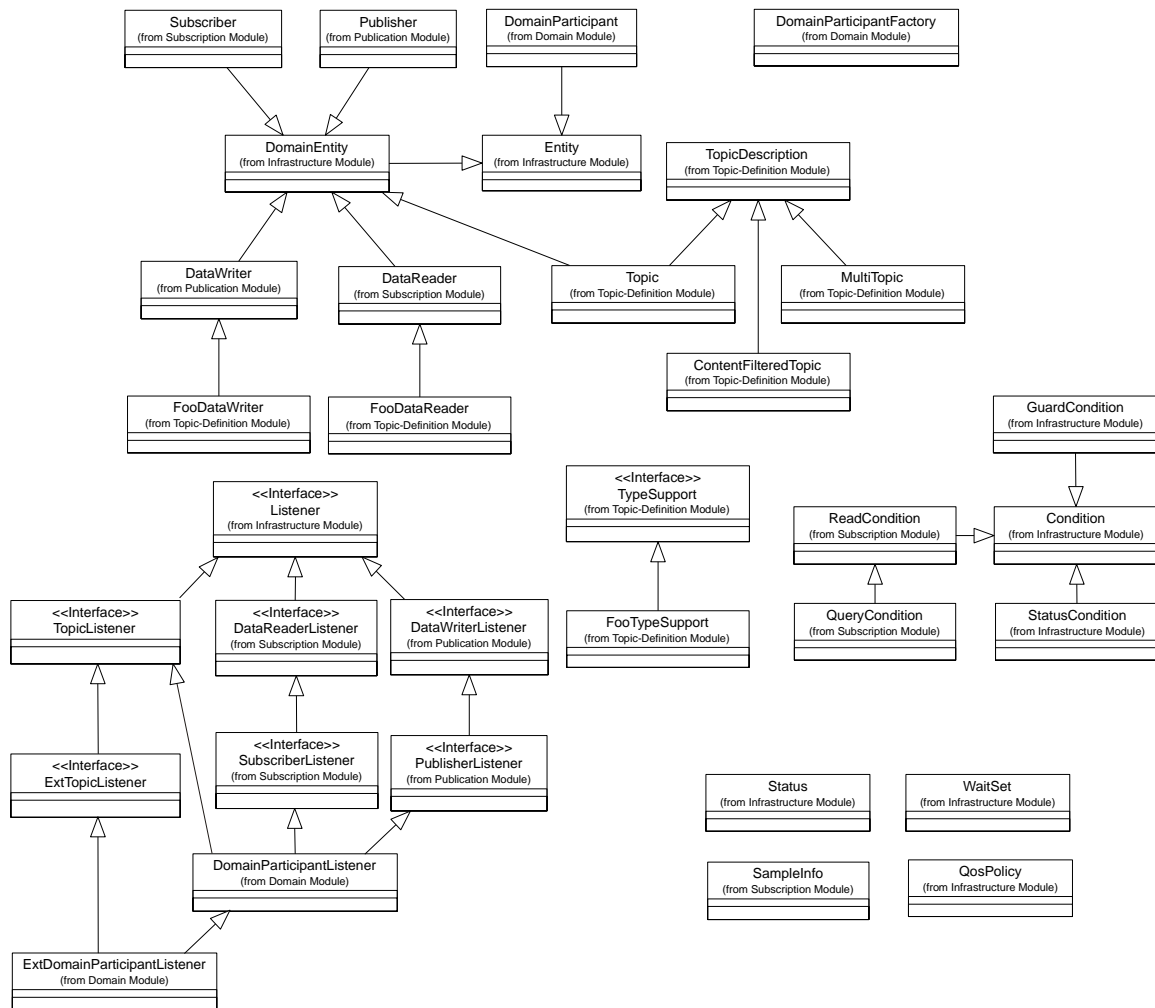
The act of taking a sample removes it from the `DataReader` so it cannot be ‘read’ or ‘taken’ again. If the sample belongs to the most recent generation of the instance, it will also set the `view_state` of the instance to `NOT_NEW_VIEW_STATE`. It will not affect the `instance_state` of the instance.



## Appendix

## E *Interface Inheritance*

This appendix gives an overview of the inheritance relations of the DCPS interfaces.



### 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 23 Communication Statuses**

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 “active” or “inactive”.
	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 “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 class. The information from an instance of this class 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` object), 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 object 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.

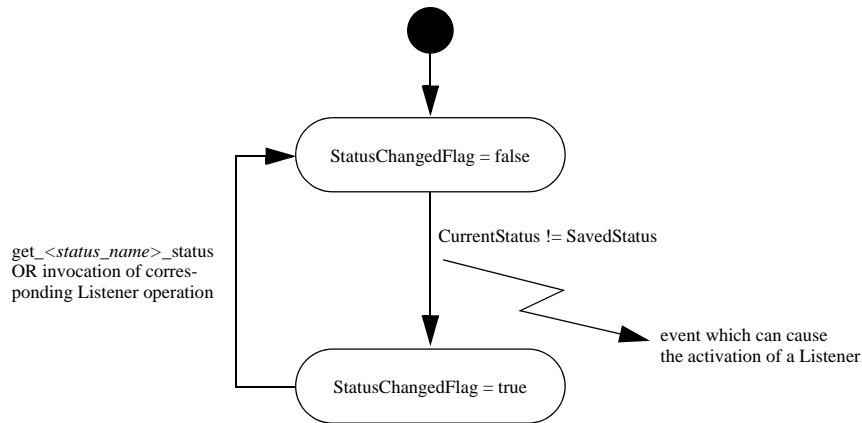
For the plain communication status, the `StatusChangedFlag` flag is initially set to `false`. It becomes `true` whenever the plain communication status changes and it is reset to `false` each time the application accesses the plain communication status via the proper `get_<status_name>_status` operation 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<sup>1</sup> for all statuses activated in its bitmask and the act of calling this ‘nil’ listener does not reset the corresponding communication statuses.

---

1. Short for **No-Operation**, an instruction that performs nothing at all.



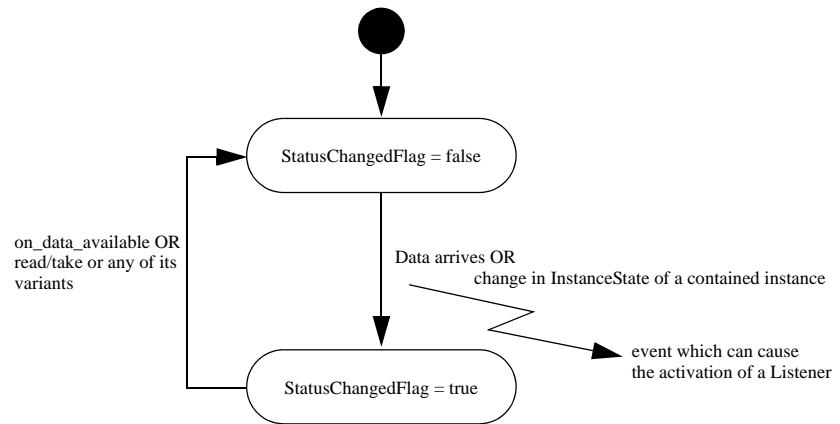
**Figure 24: Plain Communication Status State Chart**

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.

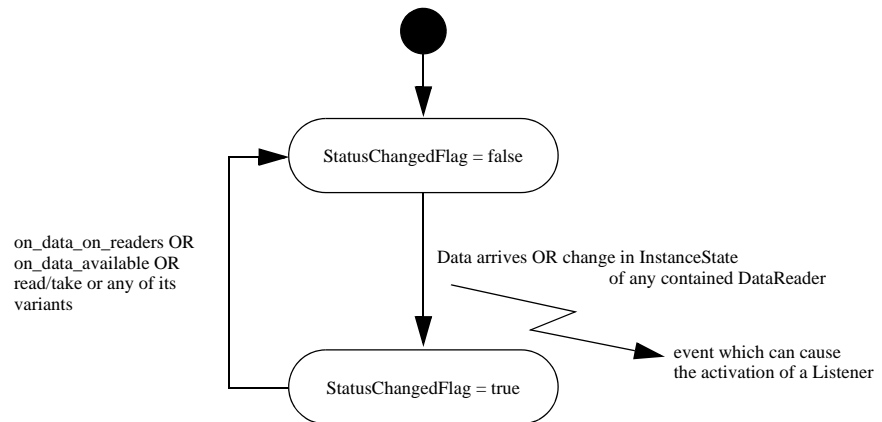
For the read communication status, the `StatusChangedFlag` flag is initially set to `false`. It becomes `true` when data arrives, or when the `InstanceState` of a contained instance changes. This can be caused by either:

- 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**

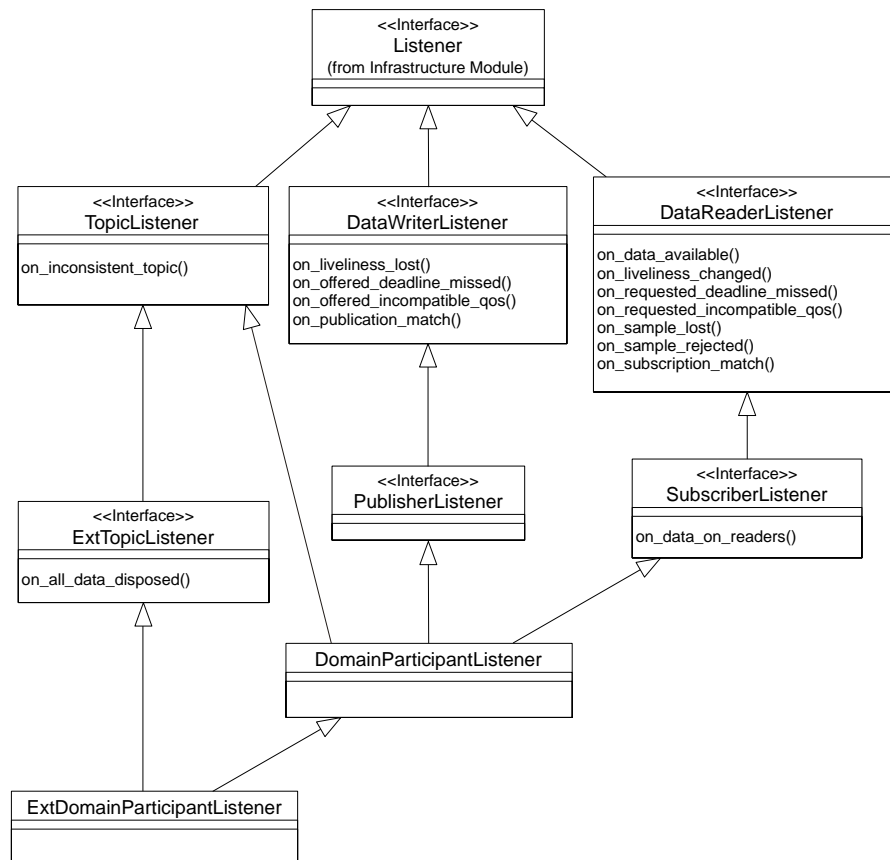
- The status flag of the `DATA_ON_READERS_STATUS` becomes `FALSE` when any of the following events occurs:
  - The corresponding listener operation (`on_data_on_readers`) is called on the corresponding Subscriber.
  - The `on_data_available` listener operation is called on any `DataReader` belonging 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 `StatusMask` 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.



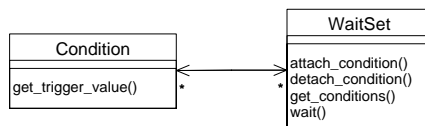
**Figure 27: DCPS Listeners**

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 Java 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.

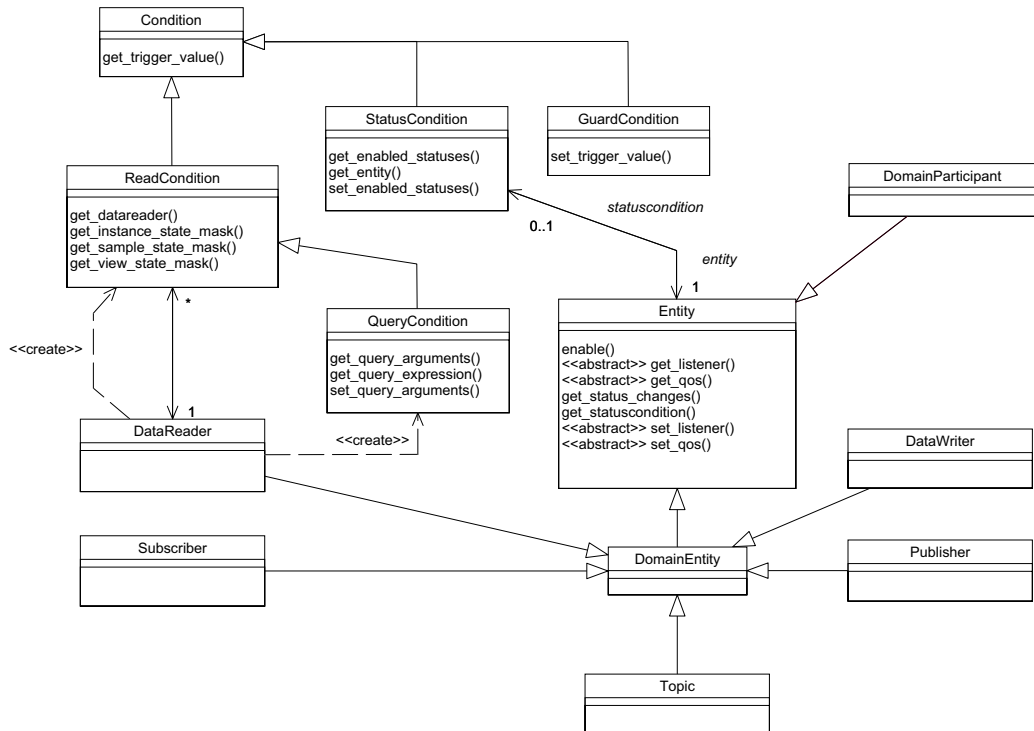
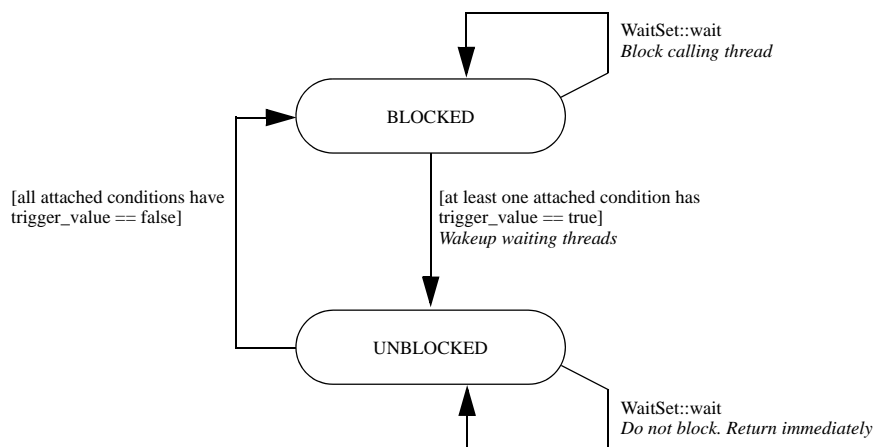


Figure 29 DCPS Conditions

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 class with several data 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 Java object 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 preprocessor, `idlpp -l java -S <idl_filename>.idl`, a tool which translates the IDL topic definition into an equivalent Java-definition. The `-l java` option indicates that Java code has to be generated. The `-S` option indicates that this Java-code should be StandAlone Java-code, i.e. 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, by 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 Preprocessor

The subset of OMG IDL that can be used to define the topics are described below.

### 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.

abstract	exception	inout	provides	truncatable
any	emits	interface	public	typedef
attribute	enum	local	publishes	typeid
boolean	eventtype	long	raises	typeprefix
case	factory	module	readonly	unsigned
char	false	multiple	setraises	union
component	finder	native	sequence	uses
const	fixed	Object	short	ValueBase
consumes	float	octet	string	valuetype
context	getraises	oneway	struct	void
custom	home	out	supports	wchar
default	import	primarykey	switch	wstring
double	in	private	true	

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 of IDL is used by the pre-processor. A description of the Data Distribution Service IDL subset, written in BNF-notation, is as follows:

```
<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

```

**Note:** 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 Java 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  $n + 1$ th 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 1 topic\_expression*

```
"SELECT flight_name, x, y, z AS height FROM 'Location'
NATURAL JOIN 'FlightPlan' WHERE height < 1000 AND x <23"
```

### *Example 2 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 Java Language Mapping Specification formal/02-08-05*, 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 geometric, wireframe effect. The lighting is soft, and the overall color palette is muted, with a slight purple or blue tint. The word "GLOSSARY" is printed in a dark blue, serif font in the upper right quadrant.

# 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 geometric, wireframe effect. The lighting is soft, and the overall color palette is muted, with a slight purple/blue tint. The word "INDEX" is printed in a dark blue, serif font in the upper right quadrant.

# INDEX



# Index

---

## A

Affected Entities .....	429	assert_liveliness (inherited) .....	275
assert_liveliness .....	121, 250	Attributes .....	432, 435, 438, 439, 441, 442

---

## B

Basic Usage .....	429	Bibliography .....	525
begin_access .....	314	Blocking Behavior of a Waitset State Chart ..	510
begin_coherent_changes .....	225	Blocking Behaviour .....	510

---

## C

Casting of Objects .....	13	Conditions .....	460
Class Condition .....	107	Conditions and Waitsets .....	508
Class DataSample .....	401	contains_entity .....	122
Class DomainEntity .....	35	copy_from_topic_qos .....	226, 314
Class DomainParticipant .....	117	create_contentfilteredtopic .....	123
Class DomainParticipantFactory .....	162	create_datareader .....	316
Class FooTypeSupport .....	217	create_datawriter .....	228
Class GuardCondition .....	109	create_multitopic .....	124
Class QosPolicy .....	35	create_participant .....	163
Class SampleInfo .....	401	create_publisher .....	126
Class Status .....	85	create_querycondition .....	338, 372
Class StatusCondition .....	111	create_readcondition .....	340
Class WaitSet .....	102	create_readcondition (inherited) .....	372
Communication Status Event .....	503	create_subscriber .....	129
Communication Statuses .....	502	create_topic .....	131
Complex Topics .....	514		

---

## D

Data Distribution Service IDL Keywords .....	514	DataWriterListener interface .....	305
Data Distribution Service IDL Pragma Keylist ..	515	DataWriterQos .....	434
Data Distribution Service IDL Subset in BNF-notation .....	515	DCPS Conditions .....	108, 509
DataReader Statecraft for a Read Communication Status .....	505	DCPS Domain Module's Class Model ..	20, 117
DATAREADER_QOS_DEFAULT .....	433	DCPS Holder Classes .....	12
DataReaderListener interface .....	409	DCPS Infrastructure Module's Class Model ..	18, 28
DataReaderQos .....	431	DCPS Inheritance .....	499
DATAWRITER_QOS_DEFAULT .....	436	DCPS Listeners .....	84, 507
		DCPS Module Composition .....	17
		DCPS Publication Module Class Model .....	23

DCPS Publication Module's Class Model ... 222  
 DCPS Status Values ..... 87  
 DCPS Subscription Module's Class Model 24, 311  
 DCPS Topic-Definition Module's Class Model 21,  
 ..... 189  
 DCPS WaitSets ..... 102, 508  
 dds\_dcps.idl ..... 455  
 DDS\_DeadlineQosPolicy ..... 59  
 DDS\_PARTICIPANT\_QOS\_DEFAULT... 437  
 DDS\_PublisherListener interface ..... 301  
 DeadlineQosPolicy ..... 44, 46  
 delete\_contained\_entities ... 134, 230, 319, 341  
 delete\_contained\_entities (inherited) ..... 372  
 delete\_contentfilteredtopic ..... 136  
 delete\_datareader ..... 320  
 delete\_datawriter ..... 232  
 delete\_multitopic ..... 137  
 delete\_participant ..... 167  
 delete\_publisher ..... 138

delete\_readcondition ..... 342  
 delete\_readcondition (inherited) ..... 372  
 delete\_subscriber ..... 139  
 delete\_topic ..... 140  
 Description ..... 432, 435, 438, 439, 441, 442  
 DestinationOrderQosPolicy ..... 46  
 detach\_condition ..... 104  
 Detailed Description 433, 436, 438, 440, 441, 443  
 dispose ..... 276  
 dispose (abstract) ..... 252  
 dispose\_w\_timestamp ..... 279  
 dispose\_w\_timestamp (abstract) ..... 252  
 Document Structure ..... 3  
 Domain Module ..... 19, 117  
 DomainParticipantListener Interface ..... 178  
 DomainParticipantQos ..... 438  
 DurabilityQosPolicy ..... 48  
 DurabilityServiceQosPolicy ..... 52  
 Duration and Time ..... 445

## E

enable ..... 29  
 enable (inherited) ... 142, 194, 233, 252, 280, 321,  
 343, ..... 373

end\_access ..... 322  
 end\_coherent\_changes ..... 233  
 EntityFactoryQosPolicy ..... 54

## F

find\_topic ..... 142  
 Foo.idl ..... 484

Functionality ..... 17

## G

get\_builtin\_subscriber ..... 143  
 get\_conditions ..... 105  
 get\_current\_time ..... 144  
 get\_datareader ..... 418  
 get\_datareader (inherited) ..... 422  
 get\_datareaders ..... 322  
 get\_default\_datareader\_qos ..... 322  
 get\_default\_datawriter\_qos ..... 234  
 get\_default\_participant\_qos ..... 168  
 get\_default\_publisher\_qos ..... 145  
 get\_default\_subscriber\_qos ..... 146  
 get\_default\_topic\_qos ..... 147

get\_discovered\_participant\_data ..... 149  
 get\_discovered\_participants ..... 149  
 get\_discovered\_topic\_data ..... 149  
 get\_discovered\_topics ..... 149  
 get\_domain\_id ..... 150  
 get\_enabled\_statuses ..... 112  
 get\_entity ..... 114  
 get\_expression\_parameters ..... 204, 209  
 get\_filter\_expression ..... 205  
 get\_inconsistent\_topic\_status ..... 194  
 get\_instance ..... 169  
 get\_instance\_handle ..... 31

get_instance_state_mask . . . . .	419	get_query_expression . . . . .	424
get_instance_state_mask (inherited) . . . . .	423	get_related_topic . . . . .	206
get_key_value . . . . .	281, 344, 373	get_requested_deadline_missed_status . . . . .	350
get_key_value (abstract) . . . . .	253	get_requested_deadline_missed_status (inherited)	375
get_listener . . . . .	151, 197, 235, 253, 323, 344	get_requested_incompatible_qos_status . . . . .	351
get_listener (abstract) . . . . .	32	get_requested_incompatible_qos_status	(inherited) . . . . . 375
get_listener (inherited) . . . . .	282, 374	get_sample_lost_status . . . . .	352
get_liveliness_changed_status . . . . .	345	get_sample_lost_status (inherited) . . . . .	376
get_liveliness_changed_status (inherited) . . . . .	374	get_sample_rejected_status . . . . .	353
get_liveliness_lost_status . . . . .	254	get_sample_rejected_status (inherited) . . . . .	376
get_liveliness_lost_status (inherited) . . . . .	282	get_sample_state_mask . . . . .	419
get_matched_publication_data . . . . .	346	get_sample_state_mask (inherited) . . . . .	425
get_matched_publication_data (inherited) . . . . .	375	get_status_changes . . . . .	32
get_matched_publications . . . . .	347	get_status_changes (inherited) 152, 199, 237, 263,	284, . . . . . 326, 354, 376
get_matched_publications (inherited) . . . . .	375	get_statuscondition . . . . .	34
get_matched_publications (inherited) . . . . .	375	get_statuscondition (inherited) 152, 199, 237, 263,	284, . . . . . 326, 354, 376
get_matched_subscription_data . . . . .	255	get_subscriber . . . . .	354
get_matched_subscription_data (inherited) . . . . .	282	get_subscriber (inherited) . . . . .	377
get_matched_subscriptions . . . . .	256	get_subscription_expression . . . . .	211
get_matched_subscriptions (inherited) . . . . .	282	get_subscription_match_status . . . . .	355
get_name . . . . .	191	get_subscription_match_status (inherited) . . . . .	377
get_name (inherited) . . . . .	197, 205, 210	get_topic . . . . .	263
get_offered_deadline_missed_status . . . . .	258	get_topic (inherited) . . . . .	284
get_offered_deadline_missed_status (inherited) . . . . .	283	get_topicdescription . . . . .	356
get_offered_incompatible_qos_status . . . . .	259	get_topicdescription (inherited) . . . . .	377
get_offered_incompatible_qos_status (inherited) . . . . .	283	get_trigger_value . . . . .	108
get_participant . . . . .	192, 236, 324	get_trigger_value (inherited) . . . . .	110, 114, 420, 425
get_participant (inherited) . . . . .	197, 206, 210	get_type_name . . . . .	192, 217, 218
get_publication_match_status . . . . .	260	get_type_name (inherited) . . . . .	199, 207, 211
get_publication_match_status (inherited) . . . . .	283	get_view_state_mask . . . . .	420
get_publisher . . . . .	261	get_view_state_mask (inherited) . . . . .	425
get_publisher (inherited) . . . . .	283	GroupDataQosPolicy . . . . .	55
get_qos . . . . .	151, 198, 236, 262, 325, 349	GuardCondition Trigger State . . . . .	511
get_qos (abstract) . . . . .	32		
get_qos (inherited) . . . . .	284, 375		
get_query_arguments . . . . .	423		

---

## H

HistoryQosPolicy . . . . . 56

---

## I

Identifications . . . . . 452

IDL Mapping to Java . . . . . 8

IDL Preprocessor . . . . .	514	Interface ContentFilteredTopic . . . . .	202
IDL to Host Language Mapping . . . . .	514	Interface DataReader . . . . .	334
IDL-Constant Mapping . . . . .	8	Interface DataWriter . . . . .	247
IDL-Sequence Mapping . . . . .	9	Interface Entity . . . . .	28
ignore_participant . . . . .	153	Interface FooDataReader . . . . .	367
ignore_publication . . . . .	153	Interface FooDataWriter . . . . .	272
ignore_subscription . . . . .	153	Interface MultiTopic . . . . .	208
ignore_topic . . . . .	154	Interface Publisher . . . . .	223
Infrastructure Module . . . . .	18, 28	Interface QueryCondition . . . . .	421
Inheritance of Abstract Operations . . . . .	15	Interface ReadCondition . . . . .	417
Instance States to Support Reads . . . . .	449	Interface Subscriber . . . . .	312
instance_state . . . . .	490	Interface Topic . . . . .	193
instance_state for a Single Instance State Chart . . . . .	492	Interface TopicDescription . . . . .	190
		Interface TypeSupport . . . . .	216

---

## J

Java Reference Guide Document Structure . . . .	3
---	---

---

## L

LatencyBudgetQosPolicy . . . . .	58	LivelinessQosPolicy . . . . .	61, 63
LifespanQosPolicy . . . . .	60	lookup_datareader . . . . .	326
Listener Interface . . . . .	83	lookup_datawriter . . . . .	238
Listeners . . . . .	458, 506	lookup_instance . . . . .	377
Listeners Interfaces . . . . .	14	lookup_instance (abstract) . . . . .	357
LivelinessChangedStatus . . . . .	90	lookup_participant . . . . .	170
LivelinessLostStatus . . . . .	91	lookup_topicdescription . . . . .	154

---

## M

Memory Management . . . . .	11
-----------------------------	----

---

## N

Names . . . . .	450	notify_datareaders . . . . .	327
-----------------	-----	------------------------------	-----

---

## O

OfferedDeadlineMissedStatus . . . . .	92	on_data_on_readers (inherited, abstract) . . . . .	181, 186
OfferedIncompatibleQosStatus . . . . .	93	on_inconsistent_topic (abstract) . . . . .	214
on_data_available . . . . .	410	on_inconsistent_topic (inherited, abstract) . . . . .	181, 186
on_data_available (inherited) . . . . .	406	on_liveliness_changed . . . . .	412
on_data_available (inherited, abstract) . . . . .	180, 186	on_liveliness_changed (inherited) . . . . .	408
on_data_on_readers . . . . .	406		



on\_liveliness\_changed (inherited, abstract) . . 181, 186  
 on\_liveliness\_lost . . . . . 306  
 on\_liveliness\_lost (inherited, abstract) . . 181, 187, 304  
 on\_offered\_deadline\_missed . . . . . 307  
 on\_offered\_deadline\_missed (inherited, abstract) . 182, . . . . . 187, 304  
 on\_offered\_incompatible\_qos . . . . . 308  
 on\_offered\_incompatible\_qos (inherited, abstract) 182, . . . . . 187, 304  
 on\_publication\_match . . . . . 309  
 on\_publication\_match (inherited, abstract) . . 182, 187, . . . . . 304  
 on\_requested\_deadline\_missed . . . . . 413  
 on\_requested\_deadline\_missed (inherited) . . 408  
 on\_requested\_deadline\_missed (inherited, abstract) . . . . . 182, 188  
 on\_requested\_incompatible\_qos . . . . . 414  
 on\_requested\_incompatible\_qos (inherited) . . 408  
 on\_requested\_incompatible\_qos (inherited, abstract) . . . . . 183, 188  
 on\_sample\_lost . . . . . 415  
 on\_sample\_lost (inherited) . . . . . 408  
 on\_sample\_lost (inherited, abstract) . . . 183, 188  
 on\_sample\_rejected . . . . . 415  
 on\_sample\_rejected (inherited) . . . . . 409  
 on\_sample\_rejected (inherited, abstract) . 183, 188  
 on\_subscription\_match (abstract) . . . . . 416  
 on\_subscription\_match (inherited, abstract) . 183, 189  
 on\_subscription\_match (inherited,) . . . . . 409  
 Operations . . . . . 4  
 Operations Concerning States . . . . . 495  
 OwnershipQosPolicy . . . . . 63, 491  
 OwnershipStrengthQosPolicy . . . . . 66

---

## P

Parameter Passing . . . . . 11  
 PARTICIPANT\_QOS\_DEFAULT . . . . . 439  
 PartitionQosPolicy . . . . . 66  
 Plain Communication Status State Chart . . . . 504  
 Pre-defined Bit Mask Definitions . . . . . 494  
 Pre-defined Values . . . . . 446, 455  
 Pre-processor Generation of the Typed Classes for Data Type “Foo” . . . . . 190  
 PresentationQosPolicy . . . . . 67  
 Publication Module . . . . . 22, 222  
 Publication Type Specific Classes . . . . . 247  
 PublicationMatchStatus . . . . . 96  
 PUBLISHER\_QOS\_DEFAULT . . . . . 440  
 PublisherListener interface . . . . . 303  
 PublisherQos . . . . . 439

---

## Q

QoS . . . . . 462  
 QosPolicy . . . . . 450  
 QosPolicy Basics . . . . . 43

QosPolicy Default Attributes . . . . . 40  
 QosPolicy Settings . . . . . 36

---

## R

read . . . . . 357, 378, 495  
 read\_instance . . . . . 358, 382, 497  
 read\_next\_instance . . . . . 358, 384  
 read\_next\_instance\_w\_condition . . . . . 386  
 read\_next\_instance\_w\_condition (abstract) . . 358  
 read\_next\_sample . . . . . 359, 388, 496  
 read\_w\_condition . . . . . 359, 388  
 ReadCondition and QueryCondition Trigger State 511  
 ReaderDataLifecycleQosPolicy . . . . . 73  
 register\_instance . . . . . 286  
 register\_instance (abstract) . . . . . 264  
 register\_instance\_w\_timestamp . . . . . 288  
 register\_instance\_w\_timestamp (abstract) . . . 265

register\_type ..... 217, 219  
 ReliabilityQosPolicy ..... 74  
 Requested Offered DestinationOrderQosPolicy 48  
 Requested Offered DurabilityQosPolicy ..... 50  
 Requested Offered PresentationQosPolicy .... 72  
 Requested Offered ReliabilityQosPolicy .. 64, 76  
 RequestedDeadlineMissedStatus ..... 96  
 RequestedIncompatibleQosStatus ..... 97  
 ResourceLimitsQosPolicy ..... 77  
 resume\_publications ..... 238  
 Return Codes ..... 7, 446, 456  
 return\_loan ..... 359, 390

## S

Sample States to Support Reads ..... 449  
 sample\_state ..... 489  
 sample\_state for a Single Sample State Chart. 490  
 SampleInfo ..... 401  
 SampleInfo Class ..... 489  
 SampleLostStatus ..... 99  
 SampleRejectedStatus ..... 100  
 Scope ..... 431, 434, 437, 438, 439, 440, 442  
 set\_default\_datareader\_qos ..... 328  
 set\_default\_datawriter\_qos ..... 239  
 set\_default\_participant\_qos ..... 171  
 set\_default\_publisher\_qos ..... 155  
 set\_default\_subscriber\_qos ..... 156  
 set\_default\_topic\_qos ..... 157  
 set\_enabled\_statuses ..... 115  
 set\_expression\_parameters ..... 207, 212  
 set\_listener ..... 158, 199, 240, 265, 329, 360  
 set\_listener (abstract) ..... 34  
 set\_listener (inherited) ..... 288, 392  
 set\_qos ..... 161, 201, 243, 267, 332, 362  
 set\_qos (abstract) ..... 35  
 set\_qos (inherited) ..... 289, 392  
 set\_query\_arguments ..... 425  
 set\_trigger\_value ..... 110  
 Signal Handling ..... 10  
 Snapshot ..... 492, 493  
 SPACE\_FooDataWriter\_dispose\_w\_timestamp .  
     301  
 SQL Examples ..... 521  
 SQL Grammar in BNF ..... 519  
 SQL Token Expression ..... 520  
 State Definitions ..... 494  
 State Masks ..... 494  
 State per Sample ..... 490  
 States ..... 449  
 Status Description Per Entity ..... 85  
 Status per Entity ..... 111  
 Status to Support Listeners and Conditions .. 447,  
     456  
 StatusCondition Trigger State ..... 510  
 Subscriber Statecraft for a Read Communication  
     Status ..... 505  
 SUBSCRIBER\_QOS\_DEFAULT ..... 441  
 SubscriberListener Interface ..... 405  
 SubscriberQos ..... 440  
 Subscription Module ..... 23, 311  
 Subscription Type Specific Classes ..... 333  
 SubscriptionMatchStatus ..... 101  
 suspend\_publications ..... 244  
 Synopsis ..... 431, 434, 438, 439, 440, 442

## T

take ..... 364, 392, 496  
 take\_instance ..... 364, 394, 497  
 take\_next\_instance ..... 364, 395  
 take\_next\_instance\_w\_condition ..... 365, 397  
 take\_next\_sample ..... 365, 399, 496  
 take\_w\_condition ..... 365, 399, 496  
 Thread Safety ..... 9  
 TimeBasedFilterQosPolicy ..... 80  
 Topic Definition Example ..... 513  
 TOPIC\_QOS\_DEFAULT ..... 443  
 TopicDataQosPolicy ..... 80  
 Topic-Definition Module ..... 21, 189  
 Topic-Definition type specific interfaces .... 216  
 TopicListener interface ..... 213  
 TopicQos ..... 442  
 TransportPriorityQosPolicy ..... 81

Typed Classes for Data Type “Foo” Pre-processor	Generation . . . . .	22
---	----------------------	----

U

unregister_instance . . . . .	289
unregister_instance (abstract) . . . . .	269
unregister_instance_w_timestamp . . . . .	292
unregister_instance_w_timestamp (abstract) . . . . .	269
UserDataQosPolicy . . . . .	82

V

View States to Support Reads . . . . .	449
view_state . . . . .	492
view_state for a Single Instance State Chart . . . . .	493

W

wait . . . . .	106
wait_for_historical_data . . . . .	366
wait_for_historical_data (inherited) . . . . .	400
write . . . . .	294
write (abstract) . . . . .	271
write_w_timestamp . . . . .	296
write_w_timestamp (abstract) . . . . .	271, 272
WriterDataLifecycleQosPolicy . . . . .	82

