



Intel® Internet Exchange Architecture Software Building Blocks

Reference Manual

November 2003

Order Number: [278665-005](#)



Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an ordering number and are referenced in this document, or other Intel literature may be obtained by calling 1-800-548-4725 or by visiting Intel's website at <http://www.intel.com>.

Copyright © Intel Corporation, 2003

AlertVIEW, i960, AnyPoint, AppChoice, BoardWatch, BunnyPeople, CablePort, Celeron, Chips, Commerce Cart, CT Connect, CT Media, Dialogic, DM3, EtherExpress, ETOX, FlashFile, GatherRound, i386, i486, iCat, iCOMP, Insight960, InstantIP, Intel, Intel logo, Intel386, Intel486, Intel740, IntelDX2, IntelDX4, IntelSX2, Intel ChatPad, Intel Create&Share, Intel Dot.Station, Intel GigaBlade, Intel InBusiness, Intel Inside, Intel Inside logo, Intel NetBurst, Intel NetStructure, Intel Play, Intel Play logo, Intel Pocket Concert, Intel SingleDriver, Intel SpeedStep, Intel StrataFlash, Intel TeamStation, Intel WebOutfitter, Intel Xeon, Intel XScale, Itanium, JobAnalyst, LANDesk, LanRover, MCS, MMX, MMX logo, NetPort, NetportExpress, Optimizer logo, OverDrive, Paragon, PC Dads, PC Parents, Pentium, Pentium II Xeon, Pentium III Xeon, Performance at Your Command, ProShare, RemoteExpress, Screamlane, Shiva, SmartDie, Solutions960, Sound Mark, StorageExpress, The Computer Inside, The Journey Inside, This Way In, TokenExpress, Trillium, Vivonic, and VTune are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

*Other names and brands may be claimed as the property of others.

Contents

1	Introduction.....	27
1.1	About this Document	27
1.2	Audience.....	27
1.3	Other Sources of Information.....	28
2	Core Components Overview	29
2.1	Overview	29
2.1.1	Functional and Data Flow	31
2.1.2	Functional APIs Design Concept	32
2.2	APIs for Dynamic Property Updates	33
2.2.1	Dynamic Properties and Clients.....	33
2.2.2	Property Updates API and Data Structures	33
2.2.2.1	Properties Data Structure	33
2.2.3	Property ID.....	38
2.2.4	Generic Prototype of Property API	40
2.2.4.1	ix_cc_<name of the cc>_set_property	40
2.2.4.2	Current Behavior of Property API	40
2.3	Handler Registration	41
2.3.1	Advantages.....	41
2.3.2	Core Component Handler Registration.....	41
2.3.3	Handler Configuration	42
2.3.4	API	42
2.3.4.1	ix_cc_add_packet_handler_list()	42
2.3.4.2	ix_cci_cc_add_message_handler_list()	44
2.3.5	Support for the IXA Portability Framework and Core Components Infrastructure	45
2.3.5.1	Usage of init() and fini() function	47
2.3.6	OS Independence of Core Components.....	47
2.4	High-Level Overview of the Core Components	48
2.4.1	System Application	49
2.4.2	POS RX	49
2.4.3	CSIX RX	49
2.4.4	Ethernet RX	50
2.4.5	CSIX TX.....	50
2.4.6	ATM/POS TX	50
2.4.7	Ethernet TX.....	50
2.4.7.1	Ethernet ARP Module	51
2.4.8	Queue Manager (QM).....	51
2.4.9	Queue Manager for DiffServ	52
2.4.10	Scheduler.....	52
2.4.11	Scheduler for DiffServ.....	52
2.4.12	IPv4 Forwarder	53
2.4.13	IPv6 Forwarder	53
2.4.14	IPv6 to IPv4 Tunneling.....	54
2.4.15	Six-Tuple Classifier.....	54
2.4.16	Three Color Meter.....	55
2.4.17	Weighted Random Early Detection (WRED)	55

2.4.18	DSCP Classifier	56
2.4.19	Route Table Manager	56
2.4.19.1	Next Hop Database.....	56
2.4.19.2	TCAM.....	57
2.4.19.3	Software LPM	57
2.4.20	Route Table Manager for IPv6.....	57
2.4.21	L2 Table Manager.....	57
2.4.22	Message Helper and Support Library	58
2.4.23	Stack Driver	58
2.4.24	SoftSAR Core Components.....	58
2.4.24.1	SAR Core Components	59
2.4.24.2	ATM RX Core Components	59
2.4.24.3	ATM TX Core Components.....	59
2.4.24.4	TM4.1 Core Components.....	60
2.4.25	MPLS Forwarder Core Component	60
3	System Application	61
3.1	Start and Shut Down the System Application.....	61
3.1.1	ix_sa_create()	62
3.1.2	_ix_sa_entry()	62
3.1.3	ix_sa_shutdown()	63
3.2	Loading Microcode and Starting Microengines.....	63
3.2.1	ix_sa_start_microengines()	64
3.3	User Initialization and Shutdown Hooks	64
3.3.1	ix_sa_init_hook_first()	65
3.3.2	ix_sa_init_hook_pre_ee()	65
3.3.3	ix_sa_init_hook_ee()	66
3.3.4	ix_sa_init_hook_pre_me()	66
3.3.5	ix_sa_init_hook_last()	67
3.3.6	ix_sa_shutdown_hook_first()	67
3.3.7	ix_sa_shutdown_hook_post_me()	68
3.3.8	ix_sa_shutdown_hook_post_ee()	68
3.3.9	ix_sa_shutdown_hook_ee()	69
3.3.10	ix_sa_shutdown_hook_last()	69
Receive		
4	POS RX API.....	73
4.1	Core Component Infrastructure API	73
4.1.1	ix_cc_pos_rx_init()	73
4.1.2	ix_cc_pos_rx_fini()	75
4.1.3	ix_cc_pos_rx_msg_handler()	76
4.1.4	ix_cc_pos_rx_pkt_handler()	77
4.2	Messaging API.....	78
4.2.1	ix_cc_pos_rx_async_get_statistics_info()	78
4.2.1.1	ix_s_cc_pos_rx_statistics_info_context	79
4.2.1.2	ix_cc_pos_rx_cb_get_statistics_info	80
4.2.2	ix_cc_pos_rx_async_get_interface_state()	80
4.2.2.1	ix_cc_pos_rx_cb_get_interface_state	81
4.2.2.2	ix_s_cc_pos_rx_if_state_context	81
4.2.2.3	ix_cc_pos_rx_if_state	82

4.3	Library API	82
4.3.1	ix_cc_pos_rx_get_statistics_info()	82
4.3.1.1	ix_e_cc_pos_rx_statistics_info	83
4.3.1.2	ix_s_cc_statistics_info_data	84
4.3.2	ix_cc_pos_rx_get_interface_state()	84
5	CSIX RX	85
5.4	Core Component Infrastructure API	85
5.4.1	ix_cc_csix_rx_init()	85
5.4.2	ix_cc_csix_rx_fini()	86
5.4.3	ix_cc_csix_rx_msg_handler()	87
5.5	Messaging API.....	88
5.5.1	ix_cc_csix_rx_async_get_statistics_info()	88
5.5.1.1	ix_s_cc_csix_rx_statistics_info_context	89
5.5.1.2	ix_cc_csix_rx_cb_get_statistics_info	90
5.6	Library API.....	90
5.6.1	ix_cc_csix_rx_get_statistics_info()	90
5.6.1.1	ix_e_cc_csix_rx_statistics_info Enumeration.....	91
5.6.1.2	ix_cc_statistics_info_data	92
6	Ethernet RX	93
6.1	Core Component Infrastructure API	93
6.1.1	ix_cc_eth_rx_init()	94
6.1.2	ix_cc_eth_rx_fini()	95
6.1.3	ix_cc_eth_rx_msg_handler()	95
6.1.4	ix_cc_eth_rx_high_priority_pkt_handler()	96
6.1.5	ix_cc_eth_rx_low_priority_pkt_handler()	97
6.2	Messaging API.....	98
6.2.1	ix_cc_eth_rx_async_get_statistics_info()	98
6.2.1.1	ix_s_cc_eth_rx_statistics_info_context	99
6.2.1.2	ix_cc_eth_rx_cb_get_statistics_info	100
6.2.2	ix_cc_eth_rx_async_get_interface_state()	101
6.2.2.1	ix_s_cc_eth_rx_if_state_context	101
6.2.2.2	ix_cc_eth_rx_cb_get_interface_state	103
6.2.3	ix_cc_eth_rx_async_add_mac_addr()	103
6.2.3.1	ix_cc_eth_rx_cb_mac_addr_op	104
6.2.4	ix_cc_eth_rx_async_delete_mac_addr()	105
6.2.4.1	ix_cc_eth_rx_cb_mac_addr_op	105
6.2.5	ix_cc_eth_rx_async_lookup_port()	106
6.2.5.1	ix_cc_eth_rx_cb_lookup_port	107
6.3	Library API.....	107
6.3.1	ix_cc_eth_rx_get_interface_state()	107
6.3.1.1	enum ix_e_cc_eth_rx_if_state Enumeration	108
6.3.2	ix_cc_eth_rx_set_property()	109
6.3.3	ix_cc_eth_rx_add_mac_addr()	109
6.3.4	ix_cc_eth_rx_del_mac_addr()	110
6.3.5	ix_cc_eth_rx_lookup_port()	111

Transmit

7	CSIX TX	115
7.1	Core Component Infrastructure API	115
7.1.1	ix_cc_csix_tx_init()	115
7.1.2	ix_cc_csix_tx_fini()	116
7.1.3	ix_cc_csix_tx_msg_handler()	117
7.2	Messaging API	118
7.2.1	ix_cc_csix_tx_async_get_statistics_info()	118
7.2.1.1	ix_s_cc_csix_tx_statistics_info_context	119
7.2.1.2	ix_e_cc_csix_tx_statistics_info Enumeration	119
7.2.1.3	ix_cc_csix_tx_cb_get_statistics_info	120
7.3	Library API	120
7.3.1	ix_cc_csix_tx_get_statistics_info()	120
7.3.1.1	ix_s_cc_statistics_info_data	121
8	ATM/POS TX	123
8.1	Core Component Infrastructure API	123
8.1.1	ix_cc_atm_pos_tx_init()	123
8.1.2	ix_cc_atm_pos_tx_fini()	124
8.1.3	ix_cc_atm_pos_tx_msg_handler()	125
8.1.4	ix_cc_atm_pos_tx_property_msg_handler()	126
8.2	Messaging API	126
8.2.1	ix_cc_atm_pos_tx_async_get_statistics_info()	127
8.2.1.1	ix_s_cc_atm_pos_tx_statistics_info_context	127
8.2.1.2	ix_e_cc_atm_pos_tx_statistics_info Enumeration	128
8.2.1.3	ix_cc_atm_pos_tx_cb_get_statistics_info	129
8.2.2	ix_cc_atm_pos_tx_async_get_interface_state()	130
8.2.2.1	ix_s_cc_atm_pos_tx_if_state_context	130
8.2.2.2	ix_cc_atm_pos_tx_cb_get_interface_state Callback	132
8.2.2.3	ix_e_cc_atm_pos_tx_if_state	132
8.3	Library API	132
8.3.1	ix_cc_atm_pos_tx_get_statistics_info()	133
8.3.1.1	ix_s_cc_atm_pos_tx_statistics_info_data	134
8.3.2	ix_cc_atm_pos_tx_get_interface_state()	134
8.3.3	ix_cc_atm_pos_tx_set_property()	135
9	Ethernet TX	137
9.1	Data Structures	137
9.1.1	ix_cc_eth_tx_next_hop_info	138
9.1.2	ix_ether_addr	138
9.1.3	ix_cc_eth_tx_if_state	139
9.2	Core Component Infrastructure API	139
9.2.1	ix_cc_eth_tx_init()	139
9.2.2	ix_cc_eth_tx_fini()	140
9.2.3	ix_cc_eth_tx_msg_handler()	141
9.2.4	ix_cc_eth_tx_property_msg_handler()	142
9.2.5	ix_cc_eth_tx_pkt_handler()	143
9.3	Messaging API	144
9.3.1	ix_cc_eth_tx_async_get_statistics_info()	145
9.3.1.1	ix_cc_eth_tx_statistics_info_context	145
9.3.1.2	ix_cc_eth_tx_statistics_info	146
9.3.1.3	ix_cc_eth_tx_cb_get_statistics_info()	148

9.3.2	<code>ix_cc_eth_tx_async_get_interface_state()</code>	149
9.3.2.1	<code>ix_cc_eth_tx_if_state_context</code>	150
9.3.2.2	<code>ix_cc_eth_tx_cb_get_interface_state()</code>	150
9.3.3	<code>ix_cc_eth_tx_async_create_arp_entry()</code>	151
9.3.3.1	<code>ix_cc_eth_tx_cb_create_arp_entry()</code>	152
9.3.4	<code>ix_cc_eth_tx_async_add_arp_entry()</code>	152
9.3.4.1	<code>ix_cc_eth_tx_cb_add_arp_entry()</code>	153
9.3.5	<code>ix_cc_eth_tx_async_del_arp_entry()</code>	154
9.3.5.1	<code>ix_cc_eth_tx_cb_del_arp_entry()</code>	154
9.3.6	<code>ix_cc_eth_tx_async_purge_arp_cache()</code>	155
9.3.7	<code>ix_cc_eth_tx_async_dump_arp_cache()</code>	155
9.4	Library API	156
9.4.1	<code>ix_cc_eth_tx_get_statistics_info()</code>	156
9.4.1.1	<code>ix_s_cc_statistics_info_data</code>	157
9.4.2	<code>ix_cc_eth_tx_get_interface_state()</code>	158
9.4.3	<code>ix_cc_eth_tx_create_arp_entry()</code>	158
9.4.4	<code>ix_cc_eth_tx_add_arp_entry()</code>	159
9.4.5	<code>ix_cc_eth_tx_del_arp_entry()</code>	160
9.4.6	<code>ix_cc_eth_tx_purge_arp_cache()</code>	160
9.4.7	<code>ix_cc_eth_tx_dump_arp_cache()</code>	161
9.4.8	<code>ix_cc_eth_tx_set_property()</code>	161
10	Ethernet ARP Module	163
10.1	Error Types	164
10.2	Library API	165
10.2.1	<code>ix_cc_arp_init()</code>	165
10.2.2	<code>ix_cc_arp_fini()</code>	166
10.2.3	<code>ix_cc_arp_create_entry()</code>	166
10.2.4	<code>ix_cc_arp_add_entry()</code>	167
10.2.5	<code>ix_cc_arp_update_entry()</code>	168
10.2.6	<code>ix_cc_arp_del_entry()</code>	169
10.2.7	<code>ix_cc_arp_purge()</code>	169
10.2.8	<code>ix_cc_arp_dump()</code>	170
10.2.9	<code>ix_cc_arp_resolve_l2_addr()</code>	170
10.2.10	<code>ix_cc_arp_process_arp_pkts()</code>	171
10.2.11	<code>ix_cc_arp_create_gratuitous_arp()</code>	173
Queue Manager		
11	Queue Manager	177
11.1	Core Component Infrastructure API	177
11.1.1	<code>ix_cc_qm_init()</code>	177
11.1.2	<code>ix_cc_qm_fini()</code>	178
11.1.3	<code>ix_cc_qm_pkt_handler()</code>	179
11.1.4	Sending Packets	179
11.1.5	<code>ix_cc_qm_msg_handler()</code>	180
11.2	Messaging API	181
11.2.1	<code>ix_cc_qm_async_get_packet_count()</code>	181
11.2.1.1	<code>ix_cc_qm_cb_pkt_count</code>	181
11.3	Library API	182
11.3.1	<code>ix_cc_qm_get_packet_count()</code>	182

12	Egress Queue Manager (DiffServ)	183
Scheduler		
13	Scheduler	187
13.1	Core Component Infrastructure API	188
13.1.1	ix_cc_scheduler_init()	188
13.1.2	ix_cc_scheduler_fini()	189
14	Egress Scheduler (DiffServ)	191
Forwarder		
15	IPv4 Forwarder	195
15.1	Data Structures, Types and Macros	195
15.1.1	IX_CC_RTMV4_DUMP_ROUTE_SIZE	196
15.1.2	IX_CC_RTMV4_DUMP_NEXT_HOP_SIZE	196
15.1.3	ix_cc_rtmv4_nhid	196
15.1.4	Reserved Next Hop Ids	196
15.1.4.1	IX_CC_RTMV4_NHID_NO_ROUTE	196
15.1.5	ix_cc_rtmv4_next_hop_info	196
15.1.6	ix_cc_ipv4_dump_data	196
15.1.7	ix_cc_ipv4_stats_data	197
15.2	Core Component Infrastructure API	198
15.2.1	ix_cc_ipv4_init()	198
15.2.2	ix_cc_ipv4_fini()	200
15.2.3	ix_cc_ipv4_msg_handler()	201
15.2.4	ix_cc_ipv4_microblock_high_priority_pkt_handler()	202
15.2.5	ix_cc_ipv4_microblock_low_priority_pkt_handler()	203
15.2.6	ix_cc_ipv4_stackdrv_pkt_handler()	204
15.2.7	ix_cc_ipv4_common_pkt_handler()	204
15.3	Message Helper API	205
15.3.1	ix_cc_ipv4_async_add_route()	206
15.3.1.1	ix_cc_ipv4_cb_route_op	207
15.3.2	ix_cc_ipv4_async_delete_route()	207
15.3.2.1	ix_cc_ipv4_cb_route_op	208
15.3.3	ix_cc_ipv4_async_update_route()	209
15.3.3.1	ix_cc_ipv4_cb_route_op	210
15.3.4	ix_cc_ipv4_async_lookup_route()	210
15.3.4.1	ix_cc_ipv4_cb_lookup_route	211
15.3.5	ix_cc_ipv4_async_purge_routes()	211
15.3.6	ix_cc_ipv4_async_dump_routes()	212
15.3.6.1	ix_cc_ipv4_cb_dump_data	212
15.3.7	ix_cc_ipv4_async_add_next_hop()	213
15.3.7.1	ix_cc_ipv4_cb_route_op	213
15.3.8	ix_cc_ipv4_async_delete_next_hop()	214
15.3.8.1	ix_cc_ipv4_cb_route_op	215
15.3.9	ix_cc_ipv4_async_update_next_hop()	215
15.3.9.1	ix_cc_ipv4_cb_route_op	216
15.3.10	ix_cc_ipv4_async_get_next_hop()	216
15.3.10.1	ix_cc_ipv4_cb_get_next_hop	217

15.3.11	<code>ix_cc_ipv4_async_dump_next_hops()</code>	217
15.3.11.1	<code>ix_cc_ipv4_cb_dump_data</code>	218
15.3.12	<code>ix_cc_ipv4_async_purge_rtm()</code>	219
15.3.13	<code>ix_cc_ipv4_async_set_mtu()</code>	219
15.3.14	<code>ix_cc_ipv4_cb_route_op</code>	220
15.3.15	<code>ix_cc_ipv4_async_set_flags()</code>	220
15.3.15.1	<code>ix_cc_ipv4_cb_route_op</code>	221
15.3.16	<code>ix_cc_ipv4_async_get_sleep_time()</code>	222
15.3.16.1	<code>ix_cc_ipv4_cb_get_sleep_time</code>	222
15.3.17	<code>ix_cc_ipv4_async_set_sleep_time()</code>	223
15.3.18	<code>ix_cc_ipv4_async_get_queue_depth()</code>	223
15.3.18.1	<code>ix_cc_ipv4_cb_get_queue_depth</code>	224
15.3.19	<code>ix_cc_ipv4_async_get_packets_to_drain()</code>	224
15.3.19.1	<code>ix_cc_ipv4_cb_get_packets_to_drain</code>	225
15.3.20	<code>ix_cc_ipv4_async_set_packets_to_drain()</code>	225
15.3.21	<code>ix_cc_ipv4_async_get_statistics()</code>	226
15.3.21.1	<code>ix_cc_ipv4_cb_get_statistics</code>	227
15.4	Library API	227
15.4.1	<code>ix_cc_ipv4_add_route()</code>	229
15.4.2	<code>ix_cc_ipv4_delete_route()</code>	230
15.4.3	<code>ix_cc_ipv4_update_route()</code>	230
15.4.4	<code>ix_cc_ipv4_lookup_route()</code>	231
15.4.5	<code>ix_cc_ipv4_purge_routes()</code>	232
15.4.6	<code>ix_cc_ipv4_dump_routes()</code>	232
15.4.7	<code>ix_cc_ipv4_add_next_hop()</code>	233
15.4.8	<code>ix_cc_ipv4_delete_next_hop()</code>	234
15.4.9	<code>ix_cc_ipv4_update_next_hop()</code>	235
15.4.10	<code>ix_cc_ipv4_get_next_hop()</code>	235
15.4.11	<code>ix_cc_ipv4_dump_next_hops()</code>	236
15.4.12	<code>ix_cc_ipv4_purge_rtm()</code>	238
15.4.13	<code>ix_cc_ipv4_set_mtu()</code>	238
15.4.13.1	<code>ix_cc_ipv4_set_flags()</code>	239
15.4.14	<code>ix_cc_ipv4_get_rtm_handle()</code>	240
15.4.15	<code>ix_cc_ipv4_get_sleep_time()</code>	240
15.4.16	<code>ix_cc_ipv4_set_sleep_time()</code>	241
15.4.17	<code>ix_cc_ipv4_get_queue_depth()</code>	241
15.4.17.1	<code>ix_cc_ipv4_get_packets_to_drain()</code>	242
15.4.18	<code>ix_cc_ipv4_set_packets_to_drain()</code>	242
15.4.19	<code>ix_cc_ipv4_get_statistics()</code>	243
15.4.20	<code>ix_cc_ipv4_set_property()</code>	244
16	IPv6 Forwarder	245
16.1	Data Structures, Types and Macros	245
16.1.1	<code>IX_CC_RTMV6_DUMP_ROUTE_SIZE</code>	245
16.1.2	<code>ix_cc_rtmv6_nhid</code>	246
16.1.3	Reserved Next Hop Ids	246
16.1.4	<code>ix_cc_rtmv6_next_hop_info</code>	246
16.1.5	<code>ix_cc_ipv6_dump_data</code>	246
16.1.6	<code>ix_cc_in_ipv6_stats_data</code>	247
16.1.7	<code>ix_cc_out_ipv6_stats_data</code>	247

16.1.8	<code>ix_cc_ipv6_stats_data</code>	248
16.1.9	<code>ix_cc_ipv6_icmp_err_type</code>	248
16.1.10	<code>ix_cc_ipv6_icmp_err_code</code>	248
16.2	Core Component Infrastructure API	249
16.2.1	<code>ix_cc_ipv6_init()</code>	249
16.2.2	<code>ix_cc_ipv6_fini()</code>	251
16.2.3	<code>ix_cc_ipv6_msg_handler()</code>	252
16.2.4	<code>ix_cc_ipv6_microblock_high_priority_pkt_handler()</code>	254
16.2.5	<code>ix_cc_ipv6_microblock_low_priority_pkt_handler()</code>	254
16.2.6	<code>ix_cc_ipv6_stackdrv_pkt_handler()</code>	255
16.2.7	<code>ix_cc_ipv6_common_pkt_handler()</code>	256
16.3	Message Helper API	257
16.3.1	<code>ix_cc_ipv6_async_add_prefix()</code>	258
16.3.1.1	<code>ix_cc_ipv6_cb_route_op</code>	258
16.3.2	<code>ix_cc_ipv6_async_delete_prefix()</code>	259
16.3.3	<code>ix_cc_ipv6_async_update_prefix()</code>	260
16.3.4	<code>ix_cc_ipv6_async_lookup_prefix()</code>	260
16.3.4.1	<code>ix_cc_ipv6_cb_lookup_route</code>	261
16.3.5	<code>ix_cc_ipv6_async_purge_prefixes()</code>	262
16.3.6	<code>ix_cc_ipv6_async_dump_prefixes()</code>	262
16.3.6.1	<code>ix_cc_ipv6_cb_dump_data</code>	263
16.3.7	<code>ix_cc_ipv6_async_add_next_hop()</code>	263
16.3.8	<code>ix_cc_ipv6_async_delete_next_hop()</code>	264
16.3.9	<code>ix_cc_ipv6_async_update_next_hop()</code>	266
16.3.10	<code>ix_cc_ipv6_async_get_next_hop()</code>	266
16.3.10.1	<code>ix_cc_ipv6_cb_get_next_hop</code>	267
16.3.11	<code>ix_cc_ipv6_async_dump_next_hops()</code>	267
16.3.12	<code>ix_cc_ipv6_async_purge_rtm()</code>	268
16.3.13	<code>ix_cc_ipv6_async_set_mtu()</code>	270
16.3.14	<code>ix_cc_ipv6_async_set_flags()</code>	270
16.3.15	<code>ix_cc_ipv6_async_get_rate_limit_time()</code>	271
16.3.15.1	<code>ix_cc_ipv6_cb_get_rate_limit_time</code>	272
16.3.16	<code>ix_cc_ipv6_async_set_rate_limit_time()</code>	272
16.3.17	<code>ix_cc_ipv6_async_get_queue_depth()</code>	273
16.3.17.1	<code>ix_cc_ipv6_cb_get_queue_depth</code>	273
16.3.18	<code>ix_cc_ipv6_async_get_statistics()</code>	274
16.3.18.1	<code>ix_cc_ipv6_cb_get_statistics</code>	274
16.3.19	<code>ix_cc_ipv6_async_perform_addr_resolution()</code>	275
16.3.19.1	<code>ix_cc_ipv6_cb_route_op</code>	275
16.3.20	<code>ix_cc_ipv6_async_add_neighbor()</code>	276
16.3.21	<code>ix_cc_ipv6_async_del_neighbor()</code>	277
16.3.22	<code>ix_cc_ipv6_sync_add_neighbor()</code>	278
16.3.23	<code>ix_cc_ipv6_sync_del_neighbor()</code>	279
16.3.24	<code>ix_cc_ipv6_async_send_icmp_error()</code>	279
16.3.25	<code>ix_cc_ipv6_async_send_icmp_info_message()</code>	281
16.4	Library API	282
16.4.1	<code>ix_cc_ipv6_add_prefix()</code>	283
16.4.2	<code>ix_cc_ipv6_delete_prefix()</code>	284
16.4.3	<code>ix_cc_ipv6_update_prefix()</code>	284
16.4.4	<code>ix_cc_ipv6_lookup_prefix()</code>	285

16.4.5	<code>ix_cc_ipv6_purge_prefixes()</code>	286
16.4.6	<code>ix_cc_ipv6_dump_prefixes()</code>	286
16.4.7	<code>ix_cc_ipv6_add_next_hop()</code>	287
16.4.8	<code>ix_cc_ipv6_delete_next_hop()</code>	287
16.4.9	<code>ix_cc_ipv6_update_next_hop()</code>	288
16.4.10	<code>ix_cc_ipv6_get_next_hop()</code>	289
16.4.11	<code>ix_cc_ipv6_dump_next_hops()</code>	290
16.4.12	<code>ix_cc_ipv6_purge_rtm()</code>	290
16.4.13	<code>ix_cc_ipv6_set_mtu()</code>	291
16.4.14	<code>ix_cc_ipv6_set_flags()</code>	291
16.4.15	<code>ix_cc_ipv6_get_rate_limit_time()</code>	292
16.4.16	<code>ix_cc_ipv6_set_rate_limit_time()</code>	293
16.4.17	<code>ix_cc_ipv6_get_queue_depth()</code>	293
16.4.18	<code>ix_cc_ipv6_get_statistics()</code>	294
16.4.19	<code>ix_cc_ipv6_set_property()</code>	294
16.4.20	<code>ix_cc_ipv6_perform_addr_resolution()</code>	295
16.4.21	<code>ix_cc_ipv6_add_neighbor()</code>	297
16.4.22	<code>ix_cc_ipv6_del_neighbor()</code>	298
16.4.23	<code>ix_cc_ipv6_send_icmp_error()</code>	298
16.4.24	<code>ix_cc_ipv6_send_icmp_info_message()</code>	299
17	IPv6 to IPv4 Tunneling	301
17.1	Data Structures, Types and Macros	301
17.1.1	<code>ix_cc_v6v4_tunnel_handle</code>	302
17.1.2	<code>ix_cc_v6v4_end_tunnel_config</code>	302
17.1.3	<code>ix_cc_v6v4_end_tunnel_info</code>	302
17.1.4	End Tunnel Flags	303
17.1.5	<code>ix_cc_v6v4_ingress_source_entry</code>	303
17.1.6	<code>ix_cc_v6v4_interface_id</code>	303
17.1.7	<code>ix_cc_v6v4_start_tunnel_config</code>	303
17.1.7.1	<code>IX_CC_V6V4_PATH_MTU_INTERFACE</code>	304
17.1.7.2	<code>IX_CC_V6V4_BROADCAST_INTERFACE</code>	304
17.1.8	<code>ix_cc_v6v4_start_tunnel_info</code>	305
17.1.9	Start Tunnel Flags	305
17.1.10	<code>ix_cc_v6v4_statistics</code>	305
17.1.11	Option Values	306
17.2	Core Component Infrastructure API	306
17.2.1	<code>ix_cc_v6v4_init()</code>	306
17.2.2	<code>ix_cc_v6v4_fini()</code>	307
17.2.3	<code>ix_cc_v6v4_msg_handler()</code>	308
17.2.4	<code>ix_cc_v6v4_microblock_pkt_handler()</code>	310
17.2.5	<code>ix_cc_v6v4_ipv6_pkt_handler()</code>	310
17.2.6	<code>ix_cc_v6v4_ipv4_pkt_handler()</code>	311
17.3	Message Helper API	312
17.3.1	<code>ix_cc_v6v4_async_add_end_tunnel()</code>	313
17.3.1.1	<code>ix_cc_v6v4_cb_add_tunnel</code>	314
17.3.2	<code>ix_cc_v6v4_async_delete_end_tunnel()</code>	314
17.3.2.1	<code>ix_cc_v6v4_cb</code>	315
17.3.3	<code>ix_cc_v6v4_async_set_decap_tos_option()</code>	316
17.3.4	<code>ix_cc_v6v4_async_set_src_validation()</code>	317

17.3.5	<code>ix_cc_v6v4_async_get_end_tunnel()</code>	317
17.3.5.1	<code>ix_cc_v6v4_cb_end_tunnel</code>	318
17.3.6	<code>ix_cc_v6v4_async_add_allowed_source()</code>	319
17.3.7	<code>ix_cc_v6v4_async_delete_allowed_source()</code>	319
17.3.8	<code>ix_cc_v6v4_async_get_allowed_sources()</code>	320
17.3.8.1	<code>ix_cc_v6v4_cb_get_sources</code>	321
17.3.9	<code>ix_cc_v6v4_async_dump_end_tunnels()</code>	321
17.3.9.1	<code>ix_cc_v6v4_cb_dump_end_tunnels</code>	322
17.3.10	<code>ix_cc_v6v4_async_clear_allowed_sources()</code>	322
17.3.11	<code>ix_cc_v6v4_async_add_start_tunnel()</code>	323
17.3.12	<code>ix_cc_v6v4_async_delete_start_tunnel()</code>	325
17.3.13	<code>ix_cc_v6v4_async_set_ttl()</code>	325
17.3.14	<code>ix_cc_v6v4_async_set_encap_tos_option()</code>	326
17.3.15	<code>ix_cc_v6v4_async_set_tos()</code>	327
17.3.16	<code>ix_cc_v6v4_async_set_mtu()</code>	329
17.3.17	<code>ix_cc_v6v4_async_set_subnet_broadcast()</code>	329
17.3.18	<code>ix_cc_v6v4_async_get_start_tunnel()</code>	331
17.3.18.1	<code>ix_cc_v6v4_cb_start_tunnel</code>	331
17.3.19	<code>ix_cc_v6v4_async_dump_start_tunnels()</code>	332
17.3.19.1	<code>ix_cc_v6v4_cb_dump_start_tunnels</code>	333
17.3.20	<code>ix_cc_v6v4_async_get_statistics()</code>	333
17.3.20.1	<code>ix_cc_v6v4_cb_statistics</code>	334
17.4	Library API	334
17.4.1	<code>ix_cc_v6v4_add_end_tunnel()</code>	335
17.4.2	<code>ix_cc_v6v4_delete_end_tunnel()</code>	336
17.4.3	<code>ix_cc_v6v4_set_decap_tos_option()</code>	336
17.4.4	<code>ix_cc_v6v4_set_src_validation()</code>	337
17.4.5	<code>ix_cc_v6v4_get_end_tunnel()</code>	339
17.4.6	<code>ix_cc_v6v4_add_allowed_source()</code>	340
17.4.7	<code>ix_cc_v6v4_delete_allowed_source()</code>	340
17.4.8	<code>ix_cc_v6v4_clear_allowed_sources()</code>	341
17.4.9	<code>ix_cc_v6v4_get_allowed_sources()</code>	342
17.4.10	<code>ix_cc_v6v4_dump_end_tunnels()</code>	343
17.4.11	<code>ix_cc_v6v4_add_start_tunnel()</code>	344
17.4.12	<code>ix_cc_v6v4_delete_start_tunnel()</code>	344
17.4.13	<code>ix_cc_v6v4_set_ttl()</code>	345
17.4.14	<code>ix_cc_v6v4_set_encap_tos_option()</code>	345
17.4.15	<code>ix_cc_v6v4_set_tos()</code>	346
17.4.16	<code>ix_cc_v6v4_set_mtu()</code>	348
17.4.17	<code>ix_cc_v6v4_set_subnet_broadcast()</code>	349
17.4.18	<code>ix_cc_v6v4_get_start_tunnel()</code>	350
17.4.19	<code>ix_cc_v6v4_dump_start_tunnels()</code>	350
17.4.20	<code>ix_cc_v6v4_get_statistics()</code>	351
17.4.21	<code>ix_cc_v6v4_set_property()</code>	352
18	NAT-PT Translation	353
18.1	Data Structures, Types and Macros	353
18.1.1	<code>ix_cc_natpt_config_params</code>	354
18.1.2	<code>ix_cc_natpt_v6addr</code>	354
18.1.3	<code>ix_cc_natpt_v4addr</code>	354

18.1.4	<code>ix_cc_natpt_static_v6v4map</code>	354
18.1.5	<code>ix_cc_natpt_v4v6portmap</code>	355
18.1.6	<code>ix_cc_natpt_naptmode</code>	355
18.1.7	<code>ix_cc_natpt_session</code>	355
18.1.8	Translation CC Specific Error Codes	356
18.2	Core Component Infrastructure API	357
18.2.1	<code>ix_cc_natpt_init()</code>	357
18.2.2	<code>ix_cc_natpt_fini()</code>	358
18.2.3	<code>ix_cc_natpt_msg_handler()</code>	359
18.2.4	<code>ix_cc_natpt_microblock_pkt_handler()</code>	360
18.3	Message Helper API	361
18.3.1	<code>ix_cc_natpt_async_set_config_parameters()</code>	362
18.3.1.1	<code>ix_cc_natpt_cb()</code>	363
18.3.2	<code>ix_cc_natpt_async_get_config_parameters()</code>	363
18.3.2.1	<code>ix_cc_natpt_cb_get_config_parameters</code>	364
18.3.3	<code>ix_cc_natpt_async_add_static_mapping()</code>	364
18.3.4	<code>ix_cc_natpt_async_remove_static_mapping()</code>	365
18.3.5	<code>ix_cc_natpt_async_get_static_mapping()</code>	366
18.3.5.1	<code>ix_cc_natpt_cb_get_static_mapping</code>	367
18.3.6	<code>ix_cc_natpt_async_add_v4addr_pool()</code>	367
18.3.7	<code>ix_cc_natpt_async_remove_v4addr_pool()</code>	368
18.3.8	<code>ix_cc_natpt_async_get_v4addr_pool()</code>	369
18.3.8.1	<code>ix_cc_natpt_cb_get_v4addr_pool</code>	369
18.3.9	<code>ix_cc_natpt_async_set_napt_mode()</code>	370
18.3.10	<code>ix_cc_natpt_async_add_v4v6port_mapping()</code>	371
18.3.11	<code>ix_cc_natpt_async_remove_v4v6port_mapping()</code>	371
18.3.12	<code>ix_cc_natpt_async_get_v4v6port_mapping()</code>	372
18.3.12.1	<code>ix_cc_natpt_cb_get_v4v6port_mapping</code>	373
18.3.13	<code>ix_cc_natpt_async_get_active_sessions()</code>	373
18.3.13.1	<code>ix_cc_natpt_cb_get_active_sessions</code>	374
18.4	Library API	375
18.4.1	<code>ix_cc_natpt_set_config_parameters()</code>	375
18.4.2	<code>ix_cc_natpt_get_config_parameters()</code>	377
18.4.3	<code>ix_cc_natpt_add_static_mapping()</code>	377
18.4.4	<code>ix_cc_natpt_remove_static_mapping()</code>	378
18.4.5	<code>ix_cc_natpt_get_static_mapping()</code>	379
18.4.6	<code>ix_cc_natpt_add_v4addr_pool()</code>	380
18.4.7	<code>ix_cc_natpt_remove_v4addr_pool()</code>	381
18.4.8	<code>ix_cc_natpt_get_v4addr_pool()</code>	382
18.4.9	<code>ix_cc_natpt_set_napt_mode()</code>	382
18.4.10	<code>ix_cc_natpt_add_v4v6port_mapping()</code>	383
18.4.11	<code>ix_cc_natpt_remove_v4v6port_mapping()</code>	384
18.4.12	<code>ix_cc_natpt_get_v4v6port_mapping()</code>	385
18.4.13	<code>ix_cc_natpt_get_active_sessions()</code>	385

DiffServ Components

19	Six-Tuple Exact Match Classifier	389
19.1	Data Structures	389
19.1.1	Classification Pattern Data Type	389

19.1.1.1	ix_s_cc_classifier_6t_pattern	390
19.1.2	Classification Result Data Type.....	390
19.1.2.1	ix_cc_classifier_6t_table_type Enumeration.....	390
19.1.2.2	ix_cc_classifier_6t_qos_output Enumeration.....	390
19.1.2.3	ix_s_cc_classifier_6t_qos_result	391
19.1.2.4	ix_s_cc_classifier_6t_fwd_result	391
19.1.2.5	ix_cc_classifier_6t_result	391
19.1.3	Statistics Data Type.....	392
19.1.3.1	ix_cc_classifier_6t_statistics	392
19.2	Core Component Infrastructure API	392
19.2.1	ix_cc_classifier_6t_init()	393
19.2.2	ix_cc_classifier_6t_fini()	394
19.2.3	ix_cc_classifier_6t_pkt_handler()	395
19.2.4	ix_cc_classifier_6t_msg_handler()	396
19.3	Message Helper API.....	398
19.3.1	ix_cc_classifier_6t_async_add_entry()	398
19.3.1.1	ix_cc_classifier_6t_cb_add_entry	399
19.3.2	ix_cc_classifier_6t_async_remove_entry()	400
19.3.2.1	ix_cc_classifier_6t_cb_remove_entry	401
19.3.3	ix_cc_classifier_6t_async_update_entry()	401
19.3.3.1	ix_cc_classifier_6t_cb_update_entry	402
19.3.4	ix_cc_classifier_6t_async_search_entry()	403
19.3.4.1	ix_cc_classifier_6t_cb_search_entry	404
19.3.5	ix_cc_classifier_6t_async_get_statistics()	405
19.3.5.1	ix_cc_classifier_6t_cb_get_statistics	406
19.3.6	ix_cc_classifier_6t_async_get_statistics_def()	407
19.3.6.1	ix_cc_classifier_6t_cb_get_statistics_def	407
19.4	Library API.....	408
19.4.1	ix_cc_classifier_6t_add_entry()	409
19.4.2	ix_cc_classifier_6t_remove_entry()	410
19.4.3	ix_cc_classifier_6t_update_entry()	411
19.4.4	ix_cc_classifier_6t_search_entry()	412
19.4.5	ix_cc_classifier_6t_get_statistics()	413
19.4.6	ix_cc_classifier_6t_get_statistics_def()	414
20	Three Color Meter.....	415
20.1	Data Structures.....	415
20.1.1	TCM Parameters Data Type.....	415
20.1.1.1	ix_cc_tc_meter_output Enumerration.....	415
20.1.1.2	ix_cc_tc_meter_algo_type Enumeration.....	416
20.1.1.3	ix_cc_tc_meter_parameters	416
20.1.2	TCM Statistics Data Type	417
20.1.2.1	ix_cc_tc_meter_statistics	417
20.2	Core Component Infrastructure API	417
20.2.1	ix_cc_tc_meter_init()	417
20.2.2	ix_cc_tc_meter_fini()	418
20.2.3	ix_cc_tc_meter_pkt_handler()	419
20.2.4	ix_cc_tc_meter_msg_handler()	420
20.3	Message Helper API.....	421
20.3.1	ix_cc_tc_meter_async_add_entry()	421
20.3.1.1	ix_cc_tc_meter_cb_add_entry	422

20.3.2	<code>ix_cc_tc_meter_async_remove_entry()</code>	423
20.3.2.1	<code>ix_cc_tc_meter_cb_remove_entry</code>	424
20.3.3	<code>ix_cc_tc_meter_async_update_entry()</code>	424
20.3.3.1	<code>ix_cc_tc_meter_cb_update_entry</code>	425
20.3.4	<code>ix_cc_tc_meter_async_get_statistics()</code>	426
20.3.4.1	<code>ix_cc_tc_meter_cb_get_statistics</code>	427
20.4	Library API	428
20.4.1	<code>ix_cc_tc_meter_add_entry()</code>	428
20.4.2	<code>ix_cc_tc_meter_remove_entry()</code>	431
20.4.3	<code>ix_cc_tc_meter_update_entry()</code>	432
20.4.4	<code>ix_cc_tc_meter_get_statistics()</code>	434
21	Weighted Random Early Detection	437
21.1	Data Structures	437
21.1.1	WRED Parameters Data Types	437
21.1.1.1	<code>ix_s_cc_red_instance</code>	437
21.1.1.2	<code>ix_s_cc_wred_parameters</code>	438
21.1.2	WRED Statistics Data Type	438
21.1.2.1	<code>ix_cc_wred_statistics</code>	438
21.2	Core Component Infrastructure API	439
21.2.1	<code>ix_cc_wred_init()</code>	439
21.2.2	<code>ix_cc_wred_fini()</code>	441
21.2.3	<code>ix_cc_wred_pkt_handler()</code>	442
21.2.4	<code>ix_cc_wred_msg_handler()</code>	442
21.3	Message Helper API	444
21.3.1	<code>ix_cc_wred_async_add_entry()</code>	444
21.3.1.1	<code>ix_cc_wred_cb_add_entry</code>	445
21.3.2	<code>ix_cc_wred_async_remove_entry()</code>	446
21.3.2.1	<code>ix_cc_wred_cb_remove_entry</code>	447
21.3.3	<code>ix_cc_wred_async_update_entry()</code>	447
21.3.3.1	<code>ix_cc_wred_cb_update_entry</code>	448
21.3.4	<code>ix_cc_wred_async_get_statistics()</code>	449
21.3.4.1	<code>ix_cc_wred_cb_get_statistics</code>	450
21.4	Library API	451
21.4.1	<code>ix_cc_wred_add_entry()</code>	451
21.4.2	<code>ix_cc_wred_remove_entry()</code>	454
21.4.3	<code>ix_cc_wred_update_entry()</code>	454
21.4.4	<code>ix_cc_wred_get_statistics()</code>	457
22	DSCP Classifier	459
22.1	Data Structures in Functional APIs	459
22.1.1	Classification Result Data Type	460
22.1.1.1	<code>ix_s_cc_classifier_dscp_result</code>	460
22.1.2	Statistics Data Type	460
22.1.2.1	<code>ix_s_cc_classifier_dscp_statistics</code>	460
22.2	Core Component Infrastructure API	460
22.2.1	<code>ix_cc_classifier_dscp_init()</code>	461
22.2.2	<code>ix_cc_classifier_dscp_fini()</code>	462
22.2.3	<code>ix_cc_classifier_dscp_pkt_handler()</code>	463
22.2.4	<code>ix_cc_classifier_dscp_msg_handler()</code>	464

22.3	Message Helper API.....	466
22.3.1	ix_cc_classifier_dscp_async_add_if_config()	466
22.3.1.1	ix_cc_classifier_dscp_cb_add_if_config	467
22.3.2	ix_cc_classifier_dscp_async_remove_if_config()	468
22.3.2.1	ix_cc_classifier_7t_cb_remove_if_config	468
22.3.3	ix_cc_classifier_dscp_async_update_if_config()	469
22.3.3.1	ix_cc_classifier_dscp_cb_update_if_config	470
22.3.4	ix_cc_classifier_dscp_async_get_if_config()	471
22.3.4.1	ix_cc_classifier_dscp_cb_get_if_config	472
22.3.5	ix_cc_classifier_dscp_async_set_def_rules()	472
22.3.5.1	ix_cc_classifier_dscp_cb_set_def_rules	473
22.3.6	ix_cc_classifier_dscp_async_get_def_rules()	474
22.3.6.1	ix_cc_classifier_dscp_cb_get_def_rules	474
22.3.7	ix_cc_classifier_dscp_async_get_statistics()	475
22.3.7.1	ix_cc_classifier_dscp_cb_get_statistics	476
22.4	Library API.....	477
22.4.1	ix_cc_classifier_dscp_add_if_config()	477
22.4.2	ix_cc_classifier_dscp_remove_if_config()	479
22.4.3	ix_cc_classifier_dscp_update_if_config()	479
22.4.4	ix_cc_classifier_dscp_set_def_rules()	480
22.4.5	ix_cc_classifier_dscp_get_def_rules()	481
22.4.6	ix_cc_classifier_dscp_get_if_config()	482
22.4.7	ix_cc_classifier_dscp_get_statistics()	483

Support Libraries

23	Route Table Manager	487
23.1	Data Structures, Types, and Macros	487
23.1.1	Data Structures, Types	487
23.1.1.1	ix_cc_rtmv4	488
23.1.1.2	ix_cc_rtmv4_nhid	488
23.1.1.3	ix_cc_rtmv4_next_hop_info	488
23.1.1.4	ix_cc_rtmv4_symbols	489
23.1.1.5	ix_cc_rtmv4_statistics	490
23.1.1.6	ix_cc_rtmv4_lkup_type	490
23.1.1.7	ix_cc_rtmv4_mem_type	490
23.1.1.8	ix_cc_rtmv4_config	490
23.1.2	Macros	491
23.1.2.1	IX_CC_RTMV4_DUMP_ROUTE_SIZE()	491
23.1.2.2	IX_CC_RTMV4_DUMP_NEXT_HOP_SIZE	492
23.1.2.3	IX_CC_RTMV4_NHID_NO_ROUTE	492
23.1.2.4	IX_CC_RTMV4_L2INDEX_NO_ROUTE	492
23.2	Core Component Infrastructure API	494
23.2.1	ix_cc_rtmv4_init()	495
23.2.2	ix_cc_rtmv4_fini()	496
23.2.3	ix_cc_rtmv4_add_next_hop()	497
23.2.4	ix_cc_rtmv4_delete_next_hop()	498
23.2.5	ix_cc_rtmv4_update_next_hop()	499
23.2.6	ix_cc_rtmv4_get_next_hop()	500
23.2.7	ix_cc_rtmv4_set_mtu()	501
23.2.8	ix_cc_rtmv4_set_flags()	502

23.2.9	<code>ix_cc_rtmv4_add_route()</code>	503
23.2.10	<code>ix_cc_rtmv4_update_route()</code>	504
23.2.11	<code>ix_cc_rtmv4_delete_route()</code>	506
23.2.12	<code>ix_cc_rtmv4_get_route</code>	507
23.2.13	<code>ix_cc_rtmv4_lookup()</code>	508
23.2.14	<code>ix_cc_rtmv4_dump_next_hops()</code>	509
23.2.15	<code>ix_cc_rtmv4_dump_routes()</code>	510
23.2.16	<code>ix_cc_rtmv4_purge()</code>	511
23.2.17	<code>ix_cc_rtmv4_purge_routes()</code>	512
23.2.18	<code>ix_cc_rtmv4_get_symbols()</code>	513
23.2.19	<code>ix_cc_rtmv4_get_statistics()</code>	514
24	Route Table Manager for IPV6	515
24.1	Data Structures, Types, and Macros	515
24.1.1	<code>ix_cc_rtmv6</code>	516
24.1.2	<code>ix_cc_rtmv6_nhid</code>	516
24.1.3	<code>ix_cc_rtmv6_next_hop_info</code>	516
24.1.4	<code>ix_cc_rtmv6_symbols</code>	517
24.1.5	<code>ix_cc_rtmv6_statistics</code>	517
24.1.6	<code>ix_cc_rtmv6_lkup_type</code>	517
24.1.7	<code>ix_cc_rtmv6_mem_type</code>	518
24.1.8	<code>IX_CC_RTMV6_DUMP_ROUTE_SIZE</code>	518
24.1.9	<code>IX_CC_RTMV6_DUMP_NEXT_HOP_SIZE</code>	518
24.1.10	<code>IX_CC_RTMV6_NHID_NO_ROUTE</code>	519
24.2	Core Component Infrastructure API	519
24.2.1	<code>ix_cc_rtmv6_init()</code>	520
24.2.2	<code>ix_cc_rtmv6_fini()</code>	521
24.2.3	<code>ix_cc_rtmv6_add_next_hop()</code>	521
24.2.4	<code>ix_cc_rtmv6_delete_next_hop()</code>	522
24.2.5	<code>ix_cc_rtmv6_update_next_hop()</code>	523
24.2.6	<code>ix_cc_rtmv6_get_next_hop()</code>	524
24.2.7	<code>ix_cc_rtmv6_set_mtu()</code>	524
24.2.8	<code>ix_cc_rtmv6_set_flags()</code>	525
24.2.9	<code>ix_cc_rtmv6_add_route()</code>	526
24.2.10	<code>ix_cc_rtmv6_delete_route()</code>	527
24.2.11	<code>ix_cc_rtmv6_update_route()</code>	528
24.2.12	<code>ix_cc_rtmv6_lookup()</code>	529
24.2.13	<code>ix_cc_rtmv6_dump_next_hops()</code>	530
24.2.14	<code>ix_cc_rtmv6_dump_routes()</code>	531
24.2.15	<code>ix_cc_rtmv6_purge()</code>	531
24.2.16	<code>ix_cc_rtmv6_purge_routes()</code>	532
24.2.17	<code>ix_cc_rtmv6_get_symbols()</code>	533
24.2.18	<code>ix_cc_rtmv6_get_statistics()</code>	534
25	L2 Table Manager	535
25.1	Data Structures, Types and Definitions	535
25.1.1	<code>ix_s_cc_l2tm_atm_header</code>	536
25.1.2	<code>ix_s_cc_l2tm_config</code>	536
25.1.3	<code>ix_s_cc_l2tm_entry</code>	537
25.1.4	<code>ix_s_cc_l2tm_ether_header</code>	538

25.1.5	<code>ix_s_cc_l2tm_stats</code>	538
25.1.6	<code>ix_s_cc_l2tm_symbols</code>	539
25.1.7	<code>ix_u_cc_l2tm_ipaddr</code>	539
25.1.8	<code>ix_cc_l2tm</code>	540
25.1.9	<code>ix_cc_l2tm_ipaddr_type</code> Enumeration.....	540
25.1.10	<code>ix_cc_l2tm_l2addr_type</code> Enumeration.....	540
25.1.11	<code>ix_e_cc_l2tm_error</code> Enumeration.....	541
25.1.12	<code>ix_cc_l2tm_memory_type</code> Enumeration.....	541
25.1.13	<code>ix_cc_l2tm_state</code> Enumeration.....	541
25.2	Library API	542
25.2.1	<code>ix_cc_l2tm_create()</code>	542
25.2.2	<code>ix_cc_l2tm_destroy()</code>	543
25.2.3	<code>ix_cc_l2tm_init()</code>	544
25.2.4	<code>ix_cc_l2tm_fini()</code>	545
25.2.5	<code>ix_cc_l2tm_add_entry()</code>	546
25.2.6	<code>ix_cc_l2tm_update_entry()</code>	547
25.2.7	<code>ix_cc_l2tm_delete_entry()</code>	547
25.2.8	<code>ix_cc_l2tm_get_entry()</code>	548
25.2.9	<code>ix_cc_l2tm_add_l3_info()</code>	549
25.2.10	<code>ix_cc_l2tm_update_l3_info()</code>	550
25.2.11	<code>ix_cc_l2tm_assign_l2_info()</code>	551
25.2.12	<code>ix_cc_l2tm_clear_l2_info()</code>	552
25.2.13	<code>ix_cc_l2tm_get_l2_index()</code>	552
25.2.14	<code>ix_cc_l2tm_get_symbols()</code>	553
25.2.15	<code>ix_cc_l2tm_get_statistics()</code>	554
25.2.16	<code>ix_cc_l2tm_purge()</code>	554
26	Message Helper and Support Library	555
26.1	Message Support Library API.....	555
26.1.1	<code>ix_cc_msup_init()</code>	555
26.1.2	<code>ix_cc_msup_fini()</code>	556
26.1.3	<code>ix_cc_msup_send_msg()</code>	557
26.1.4	<code>ix_cc_msup_send_async_msg()</code>	557
26.1.5	<code>ix_cc_msup_send_bcast_msg()</code>	558
26.1.6	<code>ix_cc_msup_send_sync_msg()</code>	559
26.1.7	<code>IX_MSUP_EXTRACT_MSG()</code>	560
26.1.8	<code>ix_cc_msup_send_reply_msg()</code>	560
Stack Driver		
27	Stack Driver.....	563
27.1	Core Component Module Data Structures and Types.....	563
27.1.1	<code>ix_cc_stkdrv_packet_type</code>	563
27.1.2	<code>ix_cc_stkdrv_handler_id</code>	564
27.1.3	<code>ix_cc_stkdrv_handler_func</code>	565
27.1.4	<code>ix_cc_stkdrv_virtual_if</code>	566
27.1.5	<code>ix_cc_stkdrv_physical_if_info</code>	567
27.1.6	<code>ix_cc_stkdrv_physical_if_node</code>	568
27.1.7	<code>ix_cc_stkdrv_handler_module</code>	568
27.1.8	<code>ix_cc_stkdrv_fp_node</code>	569

27.1.9	<code>ix_cc_stkdrv_ctrl</code>	570
27.2	Stack Driver Callback Prototypes	571
27.2.1	Communication Handler Packet Processing.....	571
27.2.1.1	<code>ix_cc_stkdrv_packet_cb()</code>	571
27.2.2	Communication Handler Message Processing.....	572
27.2.2.1	<code>ix_cc_stkdrv_msg_str_cb()</code>	572
27.2.2.2	<code>ix_cc_stkdrv_msg_int_cb()</code>	573
27.2.3	Shutdown.....	573
27.2.3.1	<code>ix_cc_stkdrv_handler_module_fini_cb()</code>	573
27.3	Core Component API.....	574
27.3.1	Core Component Infrastructure API	574
27.3.1.1	<code>ix_cc_stkdrv_init()</code>	575
27.3.1.2	<code>ix_cc_stkdrv_fini()</code>	576
27.3.1.3	<code>ix_cc_stkdrv_high_priority_pkt_handler()</code>	576
27.3.1.4	<code>ix_cc_stkdrv_low_priority_pkt_handler()</code>	577
27.3.1.5	<code>ix_cc_stkdrv_pkt_to_remote_handler()</code>	578
27.3.1.6	<code>ix_cc_stkdrv_msg_handler()</code>	579
27.3.2	Core Component Infrastructure Separation.....	580
27.3.2.1	<code>_ix_cc_stkdrv_process_pkt()</code>	580
27.3.2.2	<code>_ix_cc_stkdrv_process_pkt_to_remote()</code>	581
27.3.3	Packet and Message Processing API.....	581
27.3.3.1	<code>ix_cc_stkdrv_send_packet()</code>	582
27.3.3.2	<code>ix_cc_stkdrv_send_msg_str()</code>	582
27.3.3.3	<code>ix_cc_stkdrv_send_msg_int()</code>	583
27.3.4	Properties API.....	584
27.3.4.1	<code>ix_cc_stkdrv_async_get_property()</code>	584
27.3.4.2	<code>ix_cc_stkdrv_async_get_num_ports()</code>	585
27.4	Packet Classifier.....	588
27.4.1	Packet Classifier Data Structures	588
27.4.1.1	<code>ix_cc_stkdrv_filter_type</code>	588
27.4.1.2	<code>ix_cc_stkdrv_filter_priority</code>	589
27.4.1.3	<code>ix_cc_stkdrv_ipv4_address_range</code>	589
27.4.1.4	<code>ix_cc_stkdrv_ipv6_address_range</code>	589
27.4.1.5	<code>ix_cc_stkdrv_port_range</code>	590
27.4.1.6	<code>ix_cc_stkdrv_ipv4_range_filter</code>	590
27.4.1.7	<code>ix_cc_stkdrv_ipv6_range_filter</code>	591
27.4.1.8	<code>ix_cc_stkdrv_filter</code>	591
27.4.1.9	<code>ix_cc_stkdrv_filter_index_node</code>	592
27.4.1.10	<code>ix_cc_stkdrv_filter_handle</code>	592
27.4.1.11	<code>ix_cc_stkdrv_filter_ctrl</code>	592
27.4.2	Core Component Infrastructure API	593
27.4.2.1	<code>ix_cc_stkdrv_cb_filter_ops()</code>	593
27.4.2.2	<code>ix_cc_stkdrv_init_filters()</code>	594
27.4.2.3	<code>ix_cc_stkdrv_fini_filters()</code>	594
27.4.2.4	<code>ix_cc_stkdrv_classify_pkt()</code>	596
27.4.3	Message Helper API.....	596
27.4.3.1	<code>ix_cc_stkdrv_async_add_filter()</code>	597
27.4.3.2	<code>ix_cc_stkdrv_async_remove_filter()</code>	597
27.4.3.3	<code>ix_cc_stkdrv_async_remove_all_filters()</code>	598
27.4.3.4	<code>ix_cc_stkdrv_async_modify_filter()</code>	598
27.4.4	Library API.....	599
27.4.4.1	<code>ix_cc_stkdrv_add_filter()</code>	600

27.4.4.2	ix_cc_stkdrv_remove_filter()	600
27.4.4.3	ix_cc_stkdrv_remove_all_filters()	601
27.4.4.4	ix_cc_stkdrv_async_modify_filter()	601
27.5	Outgoing Packet Classifier Design	602
27.5.1	Outgoing Packet Classifier Data Structures	602
27.5.1.1	ix_cc_stkdrv_og_pkt_type	603
27.5.1.2	ix_cc_stkdrv_og_filter_type	603
27.5.1.3	ix_cc_stkdrv_og_cc_type	603
27.5.1.4	ix_cc_stkdrv_port_range	604
27.5.1.5	ix_cc_stkdrv_cb_og_chk_filter	604
27.5.1.6	ix_cc_stkdrv_og_filter	604
27.5.1.7	ix_cc_stkdrv_og_comm_cc_map	605
27.5.1.8	ix_cc_stkdrv_og_filter_ctrl	605
27.5.2	Outgoing Packet Classifier Internal API Functions	606
27.5.2.1	ix_cc_stkdrv_init_og_filters()	606
27.5.2.2	ix_cc_stkdrv_fini_og_filters()	606
27.5.2.3	ix_cc_stkdrv_classify_output_id()	607
27.6	VIDD for VxWorks*	608
27.6.1	VIDD System Data Structures for VxWorks	608
27.6.1.1	DEV_OBJ	608
27.6.1.2	END_ERR	609
27.6.1.3	END_OBJ	610
27.6.1.4	M2_INTERFACETBL	612
27.6.2	VIDD Local Data Structures for VxWorks	613
27.6.2.1	ix_cc_stkdrv_vidd_physical_if_node	614
27.6.2.2	ix_cc_stkdrv_vidd_fp_node	614
27.6.3	ix_cc_stkdrv_vidd_ctrl	615
27.7	MUX Interface API	616
27.7.1	NET_FUNCS	617
27.7.2	VIDD System Function Calls	618
27.7.2.1	ix_cc_stkdrv_vidd_npt_load()	619
27.7.2.2	ix_cc_stkdrv_vidd_npt_unload()	620
27.7.2.3	ix_cc_stkdrv_vidd_npt_start()	620
27.7.2.4	ix_cc_stkdrv_vidd_npt_stop()	621
27.7.2.5	ix_cc_stkdrv_vidd_npt_ioctl()	621
27.7.2.6	ix_cc_stkdrv_vidd_npt_send()	622
27.7.2.7	ix_cc_stkdrv_vidd_npt_mCastAddrAdd()	623
27.7.2.8	ix_cc_stkdrv_vidd_npt_mCastAddrDel()	624
27.7.2.9	ix_cc_stkdrv_vidd_npt_mCastAddrGet()	624
27.7.2.10	ix_cc_stkdrv_vidd_npt_pollSend()	625
27.7.2.11	ix_cc_stkdrv_vidd_npt_pollRcv()	626
27.7.3	MUX API used by the VIDD	627
27.7.3.1	muxTkReceive()	627
27.7.3.2	muxError()	628
27.7.3.3	muxTxRestart()	628
27.8	VIDD for Linux*	628
27.8.1	VIDD System Data Structures for Linux	629
27.8.1.1	sk_buff	629
27.8.1.2	net_device	630
27.8.1.3	ifreq	630
27.8.1.4	ix_cc_stkdrv_vidd_physical_if_node	631
27.8.2	VIDD System API for Linux	632
27.8.2.1	ix_cc_stkdrv_vidd_ifd_open	632

27.8.2.2	ix_cc_stkdrv_vidd_ifd_stop	632
27.8.2.3	ix_cc_stkdrv_vidd_ifd_tx	633
27.8.2.4	ix_cc_stkdrv_vidd_ifd_set_config	633
27.8.2.5	ix_cc_stkdrv_vidd_ifd_do_ioctl	634
27.8.2.6	ix_cc_stkdrv_vidd_ifd_get_stats	634
27.8.2.7	ix_cc_stkdrv_vidd_ifd_set_multicast_list	635
27.8.2.8	ix_cc_stkdrv_vidd_ifd_init	635
27.8.3	VIDD Linux Driver Support API.....	636
27.8.3.1	ix_cc_stkdrv_vidd_init	636
27.8.3.2	ix_cc_stkdrv_vidd_fini	637
27.8.3.3	ix_cc_stkdrv_vidd_if_devinet_ioctl	637
27.8.3.4	ix_cc_stkdrv_vidd_if_dev_ioctl	638
27.8.3.5	ix_cc_stkdrv_vidd_if_up	638
27.8.3.6	ix_cc_stkdrv_vidd_if_down	639
27.8.3.7	ix_cc_stkdrv_vidd_receive_pkt	639
27.8.3.8	ix_cc_stkdrv_set_ip_mask	640
27.9	Transport Module.....	641
27.9.1	Transport Data Structures	641
27.9.1.1	ix_cc_stkdrv_tm_ctrl	641
27.9.2	Transport API.....	641
27.9.2.1	ix_cc_stkdrv_tm_receive_pkt()	641
27.9.2.2	ix_cc_stkdrv_tm_pkt_handler()	643

SoftSAR

28	SoftSAR	647
28.1	SAR Core Components	647
28.1.1	Data Structures	647
28.1.1.1	VC-related Data Structures.....	648
28.1.1.2	Port-Related Data Structures.....	650
28.1.2	Core Component Infrastructure API	651
28.1.2.1	ix_cc_atmsar_init()	651
28.1.2.2	ix_cc_atmsar_fini()	652
28.1.2.3	ix_cc_atmsar_msg_handler()	653
28.1.2.4	ix_cc_atmsar_pkt_handler()	654
28.1.3	Messaging API.....	655
28.1.3.1	ix_cc_atmsar_async_vc_create()	655
28.1.3.2	ix_cc_atmsar_async_vc_update()	657
28.1.3.3	ix_cc_atmsar_async_vc_remove()	658
28.1.3.4	ix_cc_atmsar_async_port_create()	660
28.1.3.5	ix_cc_atmsar_async_port_remove()	661
28.1.3.6	ix_cc_atmsar_async_get_vc_stats()	663
28.1.3.7	ix_cc_atmsar_async_get_port_stats()	664
28.1.4	Library API	666
28.1.4.1	ix_cc_atmsar_vc_create()	666
28.1.4.2	ix_cc_atmsar_vc_update()	667
28.1.4.3	ix_cc_atmsar_vc_remove()	667
28.1.4.4	ix_cc_atmsar_port_create()	668
28.1.4.5	ix_cc_atmsar_port_remove()	669
28.1.4.6	ix_cc_atmsar_get_vc_stats()	669
28.1.4.7	ix_cc_atmsar_get_port_stats()	670
28.1.5	Plug-in API.....	671
28.1.5.1	ix_cc_atmsar_plugin_get_cfg_params()	671

28.1.5.2	<code>ix_cc_atmsar_plugin_reg_vc_service()</code>	672
28.1.5.3	<code>ix_cc_atmsar_plugin_reg_port_service()</code>	673
28.1.5.4	<code>ix_cc_atmsar_plugin_reg_vc_handle_service()</code>	674
28.1.5.5	<code>ix_cc_atmsar_plugin_reg_port_handle_service()</code>	674
28.1.5.6	<code>ix_cc_atmsar_plugin_reg_done()</code>	675
28.2	ATM RX Core Components	676
28.2.1	Core Component Infrastructure API	676
28.2.1.1	<code>ix_cc_atmr_x_init()</code>	677
28.2.1.2	<code>ix_cc_atmr_x_fini()</code>	677
28.2.1.3	<code>ix_cc_atmr_x_msg_handler()</code>	678
28.2.1.4	<code>ix_cc_atmr_x_pkt_handler()</code>	679
28.3	ATM TX Core Components	680
28.3.1	Core Component Infrastructure API	680
28.3.1.1	<code>ix_cc_atmt_x_init()</code>	680
28.3.1.2	<code>ix_cc_atmt_x_fini()</code>	681
28.4	TM4.1 Core Components	682
28.4.1	Core Component Infrastructure API	682
28.4.1.1	<code>ix_cc_atmtm41_init()</code>	683
28.4.1.2	<code>ix_cc_atmtm41_fini()</code>	685
28.4.1.3	<code>ix_cc_atmtm41_msg_handler()</code>	685
28.4.1.4	<code>ix_cc_atmtm41_pkt_handler()</code>	686

MPLS Core Component

29	MPLS Forwarder	689
29.1	Data Structures and Types	689
29.1.1	<code>ix_cc_mpls_fec</code>	691
29.1.2	<code>ix_cc_mpls_label</code>	692
29.1.3	<code>ix_cc_mpls_ilm</code>	692
29.1.4	<code>ix_cc_mpls_params</code>	693
29.1.5	<code>ix_cc_mpls_nhlfe</code>	694
29.1.6	<code>ix_cc_mpls_nhlfe_handle</code>	695
29.1.7	<code>ix_cc_mpls_cc_nhlfe</code>	695
29.1.8	<code>ix_cc_mpls_nhlfe_stats</code>	695
29.1.9	<code>ix_cc_mpls_cc_lsp</code>	696
29.1.10	<code>ix_cc_mpls_lsp_param</code>	696
29.1.11	<code>ix_cc_mpls_lsp</code>	697
29.1.12	<code>ix_cc_mpls_lsp_stats</code>	697
29.1.13	<code>ix_cc_mpls_port_stats</code>	698
29.2	Core Component Infrastructure API	698
29.2.1	<code>ix_cc_mpls_init()</code>	698
29.2.2	<code>ix_cc_mpls_fini()</code>	699
29.2.3	<code>ix_cc_mpls_msg_handler()</code>	700
29.2.4	<code>ix_cc_mpls_microblock_pkt_handler()</code>	701
29.3	Message Helper API	703
29.3.1	<code>ix_cc_mpls_cb_general()</code>	704
29.3.2	<code>ix_cc_mpls_async_lsp_create()</code>	704
29.3.2.1	<code>ix_cc_mpls_cb_lsp_create()</code>	705
29.3.3	<code>ix_cc_mpls_async_lsp_delete()</code>	706
29.3.4	<code>ix_cc_mpls_async_lsp_modify()</code>	706
29.3.4.1	<code>ix_cc_mpls_cb_lsp_modify()</code>	707

29.3.5	<code>ix_cc_mpls_async_lsp_query()</code>	708
29.3.5.1	<code>ix_cc_mpls_cb_lsp_query()</code>	708
29.3.6	<code>ix_cc_mpls_async_lsp_stats_query()</code>	709
29.3.6.1	<code>ix_cc_mpls_cb_lsp_stats_query()</code>	709
29.3.7	<code>ix_cc_mpls_async_lsp_purge()</code>	710
29.3.8	<code>ix_cc_mpls_async_nhlfe_create()</code>	711
29.3.8.1	<code>ix_cc_mpls_cb_nhlfe_create()</code>	711
29.3.9	<code>ix_cc_mpls_async_nhlfe_delete()</code>	712
29.3.10	<code>ix_cc_mpls_async_nhlfe_query()</code>	712
29.3.10.1	<code>ix_cc_mpls_cb_nhlfe_query()</code>	713
29.3.11	<code>ix_cc_mpls_async_nhlfe_stats_query()</code>	713
29.3.11.1	<code>ix_cc_mpls_cb_nhlfe_stats_query()</code>	714
29.3.12	<code>ix_cc_mpls_async_nhlfe_purge()</code>	714
29.3.13	<code>ix_cc_mpls_async_nhlfe_set_create()</code>	715
29.3.13.1	<code>ix_cc_mpls_cb_nhlfe_set_create()</code>	716
29.3.14	<code>ix_cc_mpls_async_nhlfe_set_delete()</code>	716
29.3.15	<code>ix_cc_mpls_async_nhlfe_set_modify()</code>	717
29.3.16	<code>ix_cc_mpls_async_nhlfe_set_query()</code>	718
29.3.17	<code>ix_cc_mpls_async_param_query()</code>	718
29.3.17.1	<code>ix_cc_mpls_cb_param_query()</code>	719
29.3.18	<code>ix_cc_mpls_async_port_stats_query()</code>	719
29.3.18.1	<code>ix_cc_mpls_cb_port_stats_query()</code>	720
29.4	Library API	721
29.4.1	<code>ix_cc_mpls_lsp_create()</code>	721
29.4.2	<code>ix_cc_mpls_lsp_delete()</code>	723
29.4.3	<code>ix_cc_mpls_lsp_modify()</code>	724
29.4.4	<code>ix_cc_mpls_lsp_query()</code>	725
29.4.5	<code>ix_cc_mpls_lsp_stats_query()</code>	726
29.4.6	<code>ix_cc_mpls_lsp_purge()</code>	726
29.4.7	<code>ix_cc_mpls_nhlfe_create()</code>	727
29.4.8	<code>ix_cc_mpls_nhlfe_delete()</code>	728
29.4.9	<code>ix_cc_mpls_nhlfe_query()</code>	728
29.4.10	<code>ix_cc_mpls_nhlfe_stats_query()</code>	729
29.4.11	<code>ix_cc_mpls_nhlfe_purge()</code>	730
29.4.12	<code>ix_cc_mpls_nhlfe_set_create()</code>	731
29.4.13	<code>ix_cc_mpls_nhlfe_set_delete()</code>	732
29.4.14	<code>ix_cc_mpls_nhlfe_set_modify()</code>	733
29.4.15	<code>ix_cc_mpls_nhlfe_set_query()</code>	733
29.4.16	<code>ix_cc_mpls_param_query()</code>	734
29.4.17	<code>ix_cc_mpls_port_stats_query()</code>	735
A	Glossary	737

Figures

2-1	Software Core Components	30
29-1	MPLS Core Component Data Structures.....	690

Tables

2-1	Dynamic Properties and Clients	33
2-2	Properties Data Structure	33
2-3	Property API	40
2-4	Core Component Handler Registration API	42
2-5	Core Component Functionality Mapped to Different Framework Layers	48
3-1	Starting and Shutting Down System Application	61
3-2	Loading Microcode and Starting Engines	63
3-3	User Initialization and Shutdown Hooks	64
4-1	POS RX Core Component Infrastructure API	73
4-2	POS RX Messaging API	78
4-3	POS RX Library API	82
5-1	Core Components of the CSIX API	85
5-2	Messaging API Function Calls of the CSIX RX Core Component	88
5-3	Library API Function Calls of the CSIX RX Core Component	90
6-1	Ethernet RX Core Component Functions	93
6-2	Ethernet Interface RX Messaging Functions	98
6-3	Ethernet Interface RX Library Functions	107
7-1	CSIX TX Core Component Infrastructure API	115
7-2	Messaging API for the CSIX TX Core Component	118
7-3	CSIX Library API	120
8-1	ATM/POS Interface TX Core Component Infrastructure Functions	123
8-2	ATM/POS Interface TX Messaging Functions	126
8-3	ATM/POS Interface Tx Library Functions	132
9-1	Ethernet TX Core Component Data Structures	137
9-2	Ethernet TX Core Component Infrastructure API	139
9-3	Ethernet TX Messaging Functions	144
9-4	Ethernet TX Library API	156
10-1	Ethernet ARP Library API	165
11-1	Queue Manager Core Component Infrastructure API	177
11-2	Queue Manager Core Component Messaging API	181
11-3	Queue Manager Core Component Library API	182
13-1	Scheduler Core Component Infrastructure API	188
14-1	New Patching Symbols in Scheduler (DiffServ) Core Component	191
15-1	IPV4 Forwarder Data Structures, Types, and Macros	195
15-2	IPV4 Forwarder Core Component Infrastructure API	198
15-3	IPV4 Forwarder Core Component Messages	201
15-4	IPV4 Forwarder Message Helper API	205
15-5	IPV4 Forwarder Library API	227
16-1	IPv6 Forwarder Data Structures, Types and Macros	245
16-2	IPv6 Forwarder Core Component Infrastructure API	249
16-3	IPv6 Forwarder Messages	252
16-4	IPv6 Forwarder Message Helper API	257
16-5	IPv6 Forwarder Library API	282
17-1	IPv6-IPV4 Tunneling Data Structures, Types and Macros	301
17-2	IPv6 to IPV4 Tunnelling Core Component Infrastructure AP	306
17-3	Message Types for the Tunneling Core Component	309
17-4	IPv6 to IPV4 Tunnelling Core Component Message Helper API	312
17-5	IPv6 to IPV4 Tunnelling Library API	334

18-1	Data Structures, Types and Macros in Translation Core Components.....	353
18-2	Translation-specific Error Codes.....	356
18-3	Translation Core Component Infrastructure API.....	357
18-4	Message Types for the Translation Core Component.....	359
18-5	Message Helper API in the Translation Core Component.....	361
18-6	Library API in the Translation Core Component.....	375
19-1	Data Structures for Configuring Exact-match Rules.....	389
19-2	6-tuple Classifier Core Components Infrastructure API.....	392
19-3	Microblock and Core Component Mapping Rules.....	395
19-4	6-tuple Core Component Supported Message Types.....	397
19-5	6-tuple Core Component Message Helper API.....	398
19-6	6-tuple Core Component Library API.....	408
20-1	SRTCM Core Component Data Structures.....	415
20-2	SRTCM Core Component Infrastructure APIs.....	417
20-3	SRTCM Message Types supported.....	420
20-4	SRTCM Message Helper APIs.....	421
20-5	SRTCM Library API.....	428
20-6	SRAM Entry Field Values Set by the ix_cc_tc_meter_add API.....	429
20-7	SRAM Entry Field Values Set by the ix_cc_tc_meter_update API.....	433
21-1	WRED Core Component Data Structures for Message Helper and Library APIs.....	437
21-2	WRED Core Component Infrastructure API.....	439
21-3	WRED Empty Entry Pattern.....	440
21-4	Supported Messages (ix_cc_wred_msg_handler).....	443
21-5	WRED Message Helper API.....	444
21-6	WRED Library API.....	451
21-7	WRED Table Entry Fields Set by ix_cc_wred_add_entry.....	452
21-8	WRED Table Entry Fields Set by ix_cc_wred_update_entry.....	455
22-1	Data Structures Defined by the DSCP Classifier Core Component.....	459
22-2	DSCP Classifier Core Component Infrastructure API.....	460
22-3	Messages Supported by DSCP Classifier.....	465
22-4	DSCP Classifier Message Helper API.....	466
22-5	DSCP Classifier Library API.....	477
23-1	Route Table Manager Data Structures and Types.....	487
23-2	Route Table Manager Macros.....	491
23-3	Route Table Manager Core Component Infrastructure API.....	494
24-1	RTMv6 Data Structures, Types and Macros.....	515
24-2	RTMv6 Core Component Infrastructure API.....	519
25-1	Data Structures, Types, Definitions and Enumerations in L2TM.....	535
25-2	L2 Table Manager Library API.....	542
26-1	Message Support Library API.....	555
27-1	Stack Driver Core Component Data Structures and Types.....	563
27-2	Stack Driver Callback Prototypes.....	571
27-3	Stack Driver Core Component Infrastructure API.....	574
27-4	Stack Driver Core Component Infrastructure Separation API.....	580
27-5	Stack Driver Packet and Message Processing API.....	581
27-6	Stack Driver Properties API.....	584
27-7	Stack Driver Packet Classifier Data Structures.....	588
27-8	Stack Driver Core Component Infrastructure API.....	593
27-9	Stack Driver Message Helper API.....	596
27-10	Stack Driver Library API.....	599

27-11 Stack Driver Outgoing Packet Classifier Data Structures.....	602
27-12 Stack Driver Outgoing Packet Classifier Internal API	606
27-13 Stack Driver VIDD System Data Structures.....	608
27-14 <code>END_ERR</code> Error Codes	609
27-15 Stack Driver VIDD Local Data Structures	613
27-16 Required Functions for the Stack Driver MUX Interface.....	616
27-17 VIDD System API	618
27-18 IOCTL Commands.....	621
27-19 VIDD MUX API	627
27-20 VIDD System Data Structures for Linux	629
27-21 VIDD System API for Linux.....	632
27-22 VIDD Linux Driver Support API.....	636
27-23 Transport Module Data Structures.....	641
27-24 Transport Module External API.....	641
28-1 SAR Data Structures and Data Type Definitions.....	647
28-2 SAR Core Component Infrastructure API	651
28-3 SAR Messaging API	655
28-4 SAR Library API.....	666
28-5 SAR Control Plug-in API.....	671
28-6 Core Component Infrastructure API	676
28-7 Message Type defined in ATM RX Core Components.....	678
28-8 ATM TX Core Component Infrastructure API	680
28-9 TM4.1 Core Component supported Core Component Infrastructure API.....	682
29-1 MPLS Forwarder Core Component Data Types and Structures	691
29-2 MPLS Core Component Infrastructure API.....	698
29-3 Messages Supported by MPLS Forwarder Core Components.....	701
29-4 MPLS Forwarder Core Component Message Helper API	703
29-5 MPLS Forwarder Core Component Library API	721

Revision History

Date	Revision	Description
May 2002	001	SDK 3.0 Pre-Release 3
Aug 2002	002	SDK 3.0 Pre-Release 4
Oct 2002	003	SDK 3.0 Pre-Release 5
Feb 2003	004	SDK 3.0 Pre-Release 6
April 2003	005	SDK 3.0 Pre-Release 6 - FCS
July 2003	006	SDK 3.1 Field Trial Release
November 2003	IXS SDK 3.5	Added new chapter for MPLS Core Component. Updated Stack Driver chapter with Linux* support.

1.1 About this Document

This Reference Manual introduces you to the Intel Exchange Architecture (IXA) Software Building Blocks, which is a part of the Intel[®] Internet Exchange Architecture Software Development Kit (Intel[®] IXA SDK). This software enables you to develop and deliver network applications that utilize the Intel[®] IXP2400 Network Processor and Intel[®] IXP2400 Network Processor.

There are two types of building blocks:

- Microblocks running on the MEv2 microengines
- Core Components running on the Intel XScale[®] core

These building blocks make use of the Intel Exchange Architecture (IXA) Portability Framework, which is a part of the IXA Software Development Kit (IXA SDK). The IXA Portability Framework is a network application framework and infrastructure for writing modular and portable code, which:

- Saves time by providing robust infrastructure software and APIs
- Saves time by providing re-configurable building blocks
- Permits portability across IXA network processors
- Provides an ideal structure for third-party plug-in application modules

1.2 Audience

This guide is intended for software developers who will design, develop, and deliver network applications that must process packets at high speed. It assumes that you are familiar with the following:

- C Programming
- Realtime network applications

1.3 Other Sources of Information

This manual is part of the Intel® Internet Exchange Architecture Portability Framework documentation set, which also includes the following documents:

- *Intel® Internet Exchange Architecture Portability Framework Developer's Manual*
- *Intel® Internet Exchange Architecture Portability Framework Reference Manual*
- *Intel® Internet Exchange Architecture Software Building Blocks Developer's Manual*
- *Intel® Internet Exchange Architecture Software Development Kit Software Framework Getting Started Guide*
- *IXP2400/IXP2800 Development Tools User's Guide*
- *Help Topics: Developer Workbench*
- *Intel® IXP2400/IXP2800 Network Processor Microengine C Compiler Language Support Reference*
- *Intel® IXP2400/IXP2800 Network Processor Microengine C Compiler LIBC Reference*
- *Intel® IXP2400/IXP2800 Network Processor Programmer's Reference Manual*
- *Intel® IXP2800 Network Processor Datasheet*
- *Intel® IXP2800 Network Processor Hardware Reference Manual*
- *Intel® IXP2400/IXP2800 Network Processor Programmer's Reference Manual*
- *Intel® IXP2400 Network Processor Datasheet*
- *Intel® IXP2400 Network Processor Hardware Reference Manual*

2.1 Overview

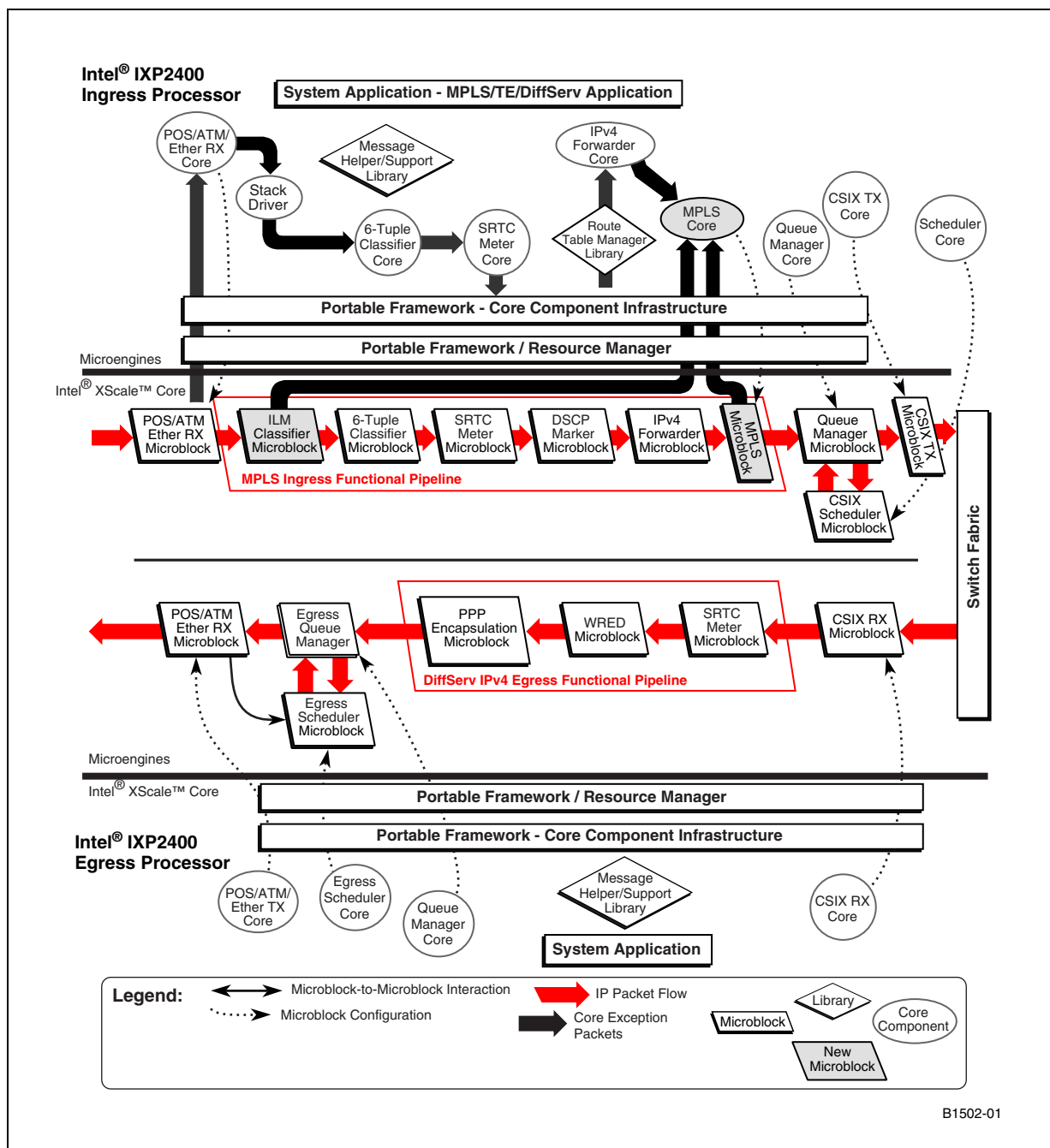
This chapter provides a software overview and describes the data flow for core components of the IXA SDK 3.1.

The core components are executed on the Xscale core processor and are counterparts of the microblocks. Core components help microblocks to do IPv4 forwarding, and provide a programming interface to higher level applications such as the Control Plane PDK. Core Components are built on top of IXA Portability Framework and Core Component Infrastructure, as described in the *Intel® Internet Exchange Architecture Portability Framework Developer's Manual* and the *IXA Portability Framework Reference Manual*.

Core components conform to the rules and APIs of the Core Component Infrastructure. The data flow of core components is designed to reflect the packet pipeline built by the microblocks on the microengines. Packet processing components have packet inputs and packets output connected to other core components or to the microblocks.

[Figure 2-1](#) illustrates the core components, libraries, and system application for the IXA SDK 3.1.

Figure 2-1. Software Core Components



2.1.1 Functional and Data Flow

The system application is responsible for initializing all modules of the system. For core components, the system application initializes the Core Component Infrastructure and starts the execution engines. It maps each core component, or a number of core components to a particular execution engine. The execution engines in turn call initialization functions for each core component running on that execution engine. The order of initialization of core components is determined by the system application.

Microcode must be downloaded to the microengines by the time the core components are started. During initialization, each core component does the following:

- Allocates memory for microblocks
- Sets up memory for shared tables
- Patches symbols
- Configures the control block of the corresponding microblock through the IXA Portability Framework APIs

The main functionality of some core components—such as the CSIX TX, CSIX RX, and Scheduler core components—is to configure the microblocks. Other core components also perform packet processing in addition to configuration. Several types of packets—treated as exception packets by the microblocks—determine the packet flow through the core components. These packets include:

- Non-IP Packets
- Packets with no route information
- Packets that require fragmentation
- Packets for local IP addresses
- Packets with IP options

As a general rule, microblocks designate a packet as an exception when the packet requires more processing than the microblock has allocated to the packet based on the line rate that the microblock must maintain.

Non-IP exception packets are delivered to the Ethernet RX core component. All such packets are sent to a core component output defined in the file `bindings.h`. By default, this output is bound to `IX_DROP`; any other core component or application that needs these packets can bind to this output. However, it is extremely easy to modify any interface RX core component to send non-IP packets to the core application—a PPP stack, for example.

Exception IP packets are delivered by the IPv4 microblock to the IPv4 Forwarder core component. If the packet is for local delivery it gets sent to the Stack Driver which, in turn, sends it to the local or remote control plane.

The IPv4 Forwarder core component processes all other packets and does one of the following:

- Forwards the packet to the Queue Manger core component
- Discards the packet

In addition, the IPv4 Forwarder core component generates ICMP messages as needed.

Outbound packets are delivered to the microblocks through the Queue Manager core component. The Queue Manager core component enqueues packets for transmission out to the switch fabric or to the POS interface.

In addition to the core components, several libraries are included in the system. On the ingress side, there are:

- Route Table Manager library
- Message Helper and Support library

On the egress side, there are:

- Message Helper and Support library
- L2 Table library
- ARP library

These standard C-language libraries provide services to the core components and higher-level applications. Note that the Route Table Manager is tightly coupled to the IPv4 Forwarder core component and cannot be initialized from other clients due to the shared routing and next hop database tables.

The ingress and egress sides of the Intel® IXMB2400 Dual Network Processor Base Card are connected through a PCI interface. The Core Component Infrastructure and IXA Portability Framework support packet and message passing over this PCI interface. In case the switch fabric does not support loopback, packets going out from the same blade output port must be transmitted over PCI from the ingress to the egress side.

Each core component is designed to conform to rules and APIs of the Core Component Infrastructure that are described in the *IXA Portability Framework Reference Manual*. The Core Component Infrastructure provides:

- A single thread of control
- Handlers for messages and packets
- Initialization and termination functions

The IXA Portability Framework's registry data structure and API is used to manage static configuration information for individual core components.

2.1.2 Functional APIs Design Concept

Two types of functional APIs are supported by all core components:

- Messaging API
- Library API

The Messaging API is for clients who communicate with the core components through the messaging mechanism of the Core Component Infrastructure and, at the same time, want access to direct C-language APIs. These APIs are provided with the help of Message Helper Library described in [Chapter 63, “Message Helper and Support Library”](#) of the *Intel® Internet Exchange Architecture Software Building Blocks Developer's Manual*.

The Library API is for clients who are not using the Core Component Infrastructure to access services of the core component. In that case, the core component can be used as a C-language library. It is assumed that clients using Library APIs have their own implementation of a packet and message passing mechanism on the Intel XScale® core.

2.2 APIs for Dynamic Property Updates

The Client core components must implement message handlers to receive and process property updates, and The Master core components must implement functions that reference the list of client `commIDs` and send a property update message to each client. The Master needs to define a generic update message that is be sent to each Client, and this message structure should be exported to the Clients.

2.2.1 Dynamic Properties and Clients

Table 2-1 lists the dynamic properties and clients.

Table 2-1. Dynamic Properties and Clients

Property	Clients
MAC address	Ethernet TX, Stack Driver
IP address, subnet mask, broadcast, gateway	Stack Driver, Ethernet TX, IPv4
Physical Interface status (up/down)	Stack Driver, IPv4, POS TX, ATM TX, Ethernet TX, POX RX, ATM RX, Ethernet RX
Link Speed	Stack Driver
MTU	Stack Driver

2.2.2 Property Updates API and Data Structures

Additional APIs are provided by the core components to complement master/client relationships. Each core component that is master or client of the property implements the generic `ix_cc_<name>_set_property` API. This API is implemented as a Messaging and Library APIs as described in section [Section 2.1.2, “Functional APIs Design Concept”](#) on page 32.

2.2.2.1 Properties Data Structure

These IP data structure defined properties data for IPv4 and IPv6 interface information.

Table 2-2. Properties Data Structure

Data Structure	Description
<code>ix_cc_media_type</code>	Media type for a port
<code>ix_ipv4_properties</code>	Structure to hold IPv4 interface properties
<code>ix_ipv6_properties</code>	Structure to hold IPv6 interface properties
<code>ix_cc_ip_version</code>	Identification for IP version
<code>ix_cc_ip_properties</code>	Container for IP properties of a virtual interface

Table 2-2. Properties Data Structure

Data Structure	Description
ix_cc_physical_if_status	Enumeration for physical interface state
ix_cc_properties	Structure to hold interface properties, used to update properties or get properties
ix_cc_properties_node	Element of linked list of property structures
ix_cc_properties_msg	Structure to hold property update messages
ix_cc_init_context	Generic context passed into a CC during initialization

2.2.2.1.1 [ix_cc_media_type](#)

Media type for a port.

C Syntax

```
typedef enum ix_e_media_type
{
    IX_MEDIA_TYPE_FIRST = 0,
    IX_MEDIA_TYPE_FAST_ETHERNET = IX_MEDIA_TYPE_FIRST,
    IX_MEDIA_TYPE_1GB_ETHERNET,
    IX_MEDIA_TYPE_ATM,
    IX_MEDIA_TYPE_POS,
    IX_MEDIA_TYPE_UNKNOWN,
    IX_MEDIA_TYPE_LAST
} ix_media_type;
```

2.2.2.1.2 [ix_ipv4_properties](#)

Structure to hold IPv4 interface properties.

C Syntax

```
typedef struct ix_s_ipv4_properties
{
    ix_uint32 ipv4_address;
    ix_uint32 ipv4_subnet_mask;
    ix_uint32 ipv4_broadcast;
    ix_uint32 ipv4_gateway;
} ix_ipv4_properties;
```

2.2.2.1.3 ix_ipv6_properties

Structure to hold IPv6 interface properties.

C Syntax

```
typedef struct ix_s_ipv6_properties
{
    ix_uint128 ipv6_address;
    ix_uint128 ipv6_subnet_mask;
    ix_uint128 ipv6_broadcast;
    ix_uint128 ipv6_gateway;
} ix_ipv6_properties;
```

2.2.2.1.4 ix_cc_ip_version

Identification for IP version.

C Syntax

```
typedef enum ix_e_ip_version
{
    IX_CC_STKDRV_IP_VERSION_IPV4,
    IX_CC_STKDRV_IP_VERSION_IPV6,
} ix_cc_ip_version;
```

2.2.2.1.5 ix_cc_ip_properties

Container for IP properties of a virtual interface.

C Syntax

```
typedef union ix_u_cc_ip_properties
{
    ix_ipv4_properties ipv4_prop;
    ix_ipv6_properties ipv6_prop;
} ix_cc_ip_properties;
```

2.2.2.1.6 ix_cc_physical_if_status

Enumeration for physical interface state. This may be expanded as necessary.

C Syntax

```
typedef enum ix_e_cc_physical_if_status
{
    IX_CC_PHYSICAL_IF_STATUS_FIRST = 0,
    IX_CC_PHYSICAL_IF_STATUS_DOWN = IX_CC_PHYSICAL_IF_STATUS_FIRST,
    IX_CC_PHYSICAL_IF_STATUS_UP,
    IX_CC_PHYSICAL_IF_STATUS_LAST,
} ix_cc_physical_if_status;
```

2.2.2.1.7 `ix_cc_properties`

Structure to hold interface properties, used to update properties or get properties.

C Syntax

```
typedef struct ix_cc_s_properties
{
    /* ID of the blade (FP) */
    ix_uint32 blade_id;

    /* ID of physical interface(port) */
    ix_uint32 port_id;

    /* IP properties. */
    ix_cc_ip_version ip_version;
    ix_cc_ip_properties ip_prop;

    ix_uint16 mtu;

    /* physical if state */
    ix_cc_physical_if_status physical_if_state;

    /* link speed in megabits-per-second. */
    ix_uint32 link_speed;

    ix_uint8 mac_addr[IX_CC_MAC_ADDR_LEN];

    /* Media type (Fast Ethernet, 1GB Ethernet, ATM, POS) */
    ix_media_type media_type;
} ix_cc_properties;
```

2.2.2.1.8 `ix_cc_properties_node`

Element of linked list of property structures.

C Syntax

```
typedef struct ix_s_cc_properties_node ix_cc_properties_node;
struct ix_s_cc_properties_node
{
    /* Pointer to interface properties data. */
    ix_cc_properties* pPropertyInfo;

    /* Pointer to next node in linked list. */
    ix_cc_properties_node* pNextPropNode;
};
```

2.2.2.1.9 ix_cc_properties_msg

Structure to hold property update messages.

C Syntax

```
typedef struct ix_s_cc_properties_msg
{
    ix_uint32 propID; /* ID of property update. */
    ix_cc_properties propData; /* properties structure with properties data.
*/
} ix_cc_properties_msg;
```

2.2.2.1.10 ix_cc_init_context

Generic context passed into a CC during initialization.

C Syntax

```
typedef struct ix_s_cc_init_context
{
    /* Handle to shared freelist of buffers for all core components. */
    ix_buffer_free_list_handle hFreeList;

    /**
     * Pointer to head of linked list of interface properties
     * for all ports configured by the system app.
     */
    ix_cc_properties_node* pPropertyList;
} ix_cc_init_context;
```

2.2.3 Property ID

Property Ids are used to identify the property to update in *set property* call. The following Ids are for the first argument in the *set property* call, and it identifies whether the fields from *ix_cc_properties* structure (second argument) was changed.

```

/* Property IDs used to set properties. */
/**
 * Used to add a logical interface for a given
 * port. In this release, it will simply
 * modify the IPv4 or IPv6 address and subnet
 * mask for the existing logical interface for
 * the port, as we support only 1 logical
 * interface per port in this release.
 */
#define IX_CC_SET_PROPERTY_ADD_INTERFACE_IPV4 0x00000001
#define IX_CC_SET_PROPERTY_ADD_INTERFACE_IPV6 0x00000002

/**
 * Used to delete a logical interface for a
 * given port. In this release, it will simply
 * set the IPv4 or IPv6 address and subnet mask
 * for the existing logical interface for the
 * port to 0.0.0.0, as we support only 1
 * logical interface per port in this release.
 */
#define IX_CC_SET_PROPERTY_DEL_INTERFACE_IPV4 0x00000004
#define IX_CC_SET_PROPERTY_DEL_INTERFACE_IPV6 0x00000008

/**
 * Used to add a gateway address for a given
 * port.
 */
#define IX_CC_SET_PROPERTY_ADD_GATEWAY_IPV4 0x00000010
#define IX_CC_SET_PROPERTY_ADD_GATEWAY_IPV6 0x00000020

/**
 * Used to delete a gateway address for a
 * given port - sets gateway to 0.0.0.0.
 */
#define IX_CC_SET_PROPERTY_DEL_GATEWAY_IPV4 0x00000040
#define IX_CC_SET_PROPERTY_DEL_GATEWAY_IPV6 0x00000080

/**
 * Used to add a broadcast address for a given
 * port.
 */
#define IX_CC_SET_PROPERTY_ADD_BROADCAST_IPV4 0x00000100
#define IX_CC_SET_PROPERTY_ADD_BROADCAST_IPV6 0x00000200

```

```

/**
 * Used to delete a broadcast address for a
 * given port - sets broadcast to 0.0.0.0.
 */
#define IX_CC_SET_PROPERTY_DEL_BROADCAST_IPV4    0x00000400
#define IX_CC_SET_PROPERTY_DEL_BROADCAST_IPV6    0x00000800

/* Used to modify the MTU of a port. */
#define IX_CC_SET_PROPERTY_MTU                    0x00001000

/* Used to set the physical port status (up/down, etc.). */
#define IX_CC_SET_PROPERTY_PHYSICAL_IF_STATUS    0x00002000

/* Used to set the virtual port status (up/down, etc.). */
/* Not supported in this release. */
#define IX_CC_SET_PROPERTY_VIRTUAL_IF_STATUS     0x00008000

/**
 * Used to set the link status (up/down, etc.). This should
 * only be used by a component/application that can detect
 * when the link goes up or down. For example, the proper
 * use of this would be when a cable is unplugged, and the
 * media driver would then trigger an interrupt to an
 * Interface CC. The Interface CC would then invoke the
 * async_set_property API into the Stack Driver, which
 * would notify the appropriate property clients that the
 * link has gone down.
 */
/* Not supported in this release. */
#define IX_CC_SET_PROPERTY_LINK_STATUS           0x00010000

/* Used to set the MAC address for a port. */
#define IX_CC_SET_PROPERTY_MAC_ADDR              0x00020000

/**
 * Used to set link speed for a port -
 * unsupported in this release.
 */
#define IX_CC_SET_PROPERTY_LINK_SPEED            0x00008000

```

2.2.4 Generic Prototype of Property API

Table 2-3 shows the generic prototype of property API.

Table 2-3. Property API

Data Structures	Description
<code>ix_cc_<name of the cc>_set_property</code>	Sets the property, defined by <code>arg_PropId</code> to the core component.

2.2.4.1 `ix_cc_<name of the cc>_set_property`

The API sets the property, defined by `arg_PropId` to the core component. Each core component implements Messaging API—`ix_cc_<name of the cc>_async_set_property()` and Library API—`ix_cc_<name of the cc>_set_property()`. The first argument corresponds to the specific field in the properties data structure (second argument).

C Syntax

```
ix_cc_<name of the cc>_set_property (
    ix_uint32 arg_PropId,
    ix_cc_properties* arg_pProperty);
```

Input

`arg_PropId` Id of the property to set, as defined in `ix_cc_<name of the cc>_set_property`.

`arg_pProperty` The pointer to `ix_cc_properties` data structure.

Output/Returns

Return Value Returns a valid `ix_error`.

- `IX_SUCCESS`—the operation succeeded.
- A valid `ix_error`—the operation failed.—each component specific error code

2.2.4.2 Current Behavior of Property API

Each component has two types of APIs—Messaging API and Library API. The Messaging API is designed to provide functions to the calling application that does not require usage of messaging mechanism of the Core Component Infrastructure. The Messaging API can be considered as external interface of core components. The Library API provide internal interface to the core components, it does not require message support, and does not depend on the Core Component Infrastructure.

For property interfaces, each component that deals with the property implements messaging and library API, that takes property ID and properties data structure as an argument. Certain rules apply to behavior of these APIs.

2.3 Handler Registration

This section describes the method used by the core components to register handlers.

The core components need to register a variety of handlers:

- **Packet-Handlers**—A core component may need to register one or more packet handlers for various types of packet flow.
- **Message-Handlers**—Message handlers may need to be registered for different types of message input. For example, a handler for messages from “Message Helper” functions and another for properties subscribed to.

2.3.1 Advantages

Core components should not simply register handlers for the communication ID that it is interested in. All sources would send data to that single communication ID. The problem with doing this is that there is a performance benefit to providing a separate communication ID for each data source (message or packet). This alleviates the need for the Core Component Infrastructure to provide locking mechanisms due to the multiple sources.

The problem is that a core-component's code would need to be changed manually to add handler registration for each new communication ID to be served.

This design provides a way for core-components to remain unchanged while allowing a system designer to specify the communication IDs that are feeding the core-component.

2.3.2 Core Component Handler Registration

For each handler, message or packet, provided by a core component, a list of communication IDs is specified as a configuration parameter. At initialization time, the core component registers the message handler for each communication ID in the list.

A system designer may still choose to use a single communication ID for each handler, however this system provides a standard configuration method used by all core components.

2.3.3 Handler Configuration

The configuration for each handler is a list of communication IDs specified within a configuration header file. These lists are terminated by a “0” entry. Below are two examples.

Example 1—Communication ID in a Configuration Header File

```
/*A packet handler list using communication ID numbers*/
#define IX_CC_MYCC_PACKET_HANDLER_CNF {800, 817, ..., 0}
```

Example 2—Communication ID in a Configuration Header File

```
/*A message handler list using predefined communication ID names*/
#define IX_CC_MYCC_MSGHLP_HANDLER_CNF {COMID_1, COMID_2, ..., 0}
```

Note: The above names are examples and the handlers should follow an appropriate naming conventions. The convention shown in second example, that is communication ID names, should be used. Hard-coding communication ID numbers is forbidden.

2.3.4 API

Two support functions are provided to support the handler configuration method, one for message handlers and one for packet handlers. These functions are identical to the registration functions provided by the Core Component Infrastructure except for the addition of a parameter consisting of the list of communication IDs. They register the specified handler for each of the communication IDs in the list.

Table 2-4. Core Component Handler Registration API

<code>ix_cc_add_packet_handler_list()</code>	Adds a packet handler to a core component and registers it with each ID in the list.
<code>ix_cci_cc_add_message_handler_list()</code>	Adds a message handler to a core component and registers it with each ID in the list.

2.3.4.1 `ix_cc_add_packet_handler_list()`

This function is a wrapper for `ix_cci_cc_add_packet_handler`.

Adds a packet handler to a core component and the specified handler is registered with each ID in the list (parameter 2) as opposed to one ID for `ix_cci_cc_add_packet_handler()`. This function should be called in the `ix_exe_init()` function of the core component's execution engine. Attempting to associate a handler to an ID that already is associated with a handler returns an error.

C Syntax

```
ix_error ix_cc_add_packet_handler_list(
    ix_cc_handle arg_hComponent,
    ix_uint32 arg_InputID_List[MAX_IID_LIST_SIZE],
    ix_pkt_handler arg_Handler,
    void* arg_pContext,
    ix_input_type arg_SourceType);
```

Input

<code>arg_hComponent</code>	A handle to the execution engine used to gain access to the execution engine's default policy. The message handler pointed to by <code>arg_Handler</code> is added to this policy.
<code>arg_InputID_List</code>	This parameter is a list of communication Ids (input Ids) for which to register the handler for.
<code>arg_Handler</code>	Pointer to packet handler. See ix_pkt_handler for details.
<code>arg_pcontext</code>	Pointer to the context associated with the handler.
<code>arg_SourceType</code>	Indicates whether the input receives messages from one source or multiple sources. Valid values are: <ul style="list-style-type: none"> <code>IX_INPUT_TYPE_SINGLE_SRC</code> <code>IX_INPUT_TYPE_MULTI_SRC</code>. <p>NOTE: The framework uses the type value to determine whether or not to employ locking mechanism when writing data into the queue.</p>

Output

Return Value	Returns zero if successful or one of the following <code>ix_error</code> types for failure: <ul style="list-style-type: none"> <code>IX_CCI_ERR_OS_MEM_ALLOC</code>—not enough memory Any errors returned by the <code>ix_cc_init()</code> callback function <code>IX_CCI_ASSERTION</code>—invalid function parameter—debug compilations only
<code>arg_phComponent</code>	A pointer to the location for return of the core-component handle.

2.3.4.1.1 [ix_pkt_handler](#)

This is the pointer to the packet handler with following signature:

C Syntax

```
ix_error (* ix_pkt_handler) (
    ix_buffer_handle arg_hDataToken,
    ix_uint32 arg_ExceptionCode,
    void* arg_pComponentContext);
```

Input

<code>arg_hDataToken</code>	Data being received by the handler.
<code>arg_ExceptionCode</code>	Data being received by the handler.
<code>arg_pComponentContext</code>	The core component context created by the application-defined core component <code>ix_cc_init()</code> function passed to the core component's creation function, <code>ix_cci_cc_create()</code> .

2.3.4.2 `ix_cci_cc_add_message_handler_list()`

Adds a message handler to a core component and the specified handler is registered with each ID in the list (parameter 2) as opposed to one ID for `ix_cci_cc_add_message_handler()`. This function should be called in the `ix_exe_init()` function of the core component's execution engine. Attempting to associate a handler to an ID that already is associated with a handler returns an error.

C Syntax

```
ix_error
ix_cci_cc_add_message_handler_list (
    ix_cc_handle arg_hComponent,
    ix_uint32 arg_InputID_List[MAX_IID_LIST_SIZE],
    ix_pkt_handler arg_Handler,
    void* arg_pContext,
    ix_input_type arg_SourceType);
.
```

Input

<code>arg_hComponent</code>	A handle to the execution engine used to gain access to the execution engine's default policy. The message handler pointed to by <code>arg_Handler</code> is added to this policy.
<code>arg_InputID_List</code>	This parameter is a list of communication Ids (input Ids) for which to register the handler for.
<code>arg_Handler</code>	A pointer to a message handler. See ix_msg_handler .
<code>arg_pContext</code>	The context associated with the handler.
<code>arg_SourceType</code>	Indicates whether the input receives messages from one source or multiple sources. Valid values are: <ul style="list-style-type: none"> <code>IX_INPUT_TYPE_SINGLE_SRC</code> <code>IX_INPUT_TYPE_MULTI_SRC</code> <p>NOTE: The framework uses this type value to determine whether or not to employ locking when writing data into the queue.</p>

Output

Return Value	<p>Returns zero if successful or one of the following valid <code>ix_error</code> types for failure:</p> <ul style="list-style-type: none"> • <code>IX_CCI_ERR_OS_MEM_ALLOC</code>—could not allocate enough memory for internal handler structure • <code>IX_CCI_TOKEN_PROCESSOR_CONTAINER_FULL</code>—attempted to add handler to a full policy that cannot grow • <code>IX_CCI_ERR_TKP_CONTAINER_MEM_ALLOC</code>—memory allocation error when attempting to add handler to a full policy • <code>IX_CCI_ERR_MSG_SET_MODE</code>—error reported by Resource Manager when setting handler's mode of operation • <code>IX_CCI_ASSERTION</code>—invalid function parameter—debug compilations only
--------------	---

2.3.4.2.1 `ix_msg_handler`

A pointer to a message handler with the following signature.

C Syntax

```
ix_error (* ix_msg_handler) (
    ix_buffer_handle arg_hDataToken,
    ix_uint32 arg_UserData,
    void* arg_pComponentContext);
```

Input

<code>arg_hDataToken</code>	The data being received by the handler
<code>arg_UserData</code>	The data being received by the handler
<code>arg_pComponentContext</code>	The core component context created by the application-defined core component <code>ix_cc_init_context</code> function passed to the core component's creation function.

2.3.5 Support for the IXA Portability Framework and Core Components Infrastructure

There are 2 possible configurations for using the IXA Portability Framework and the Core Component Infrastructure.

- Using the Resource Manager
- Using the Resource Manager and the Core Components Infrastructure

The core components are designed to be used in both configurations. The internal functionality of the core components does not depend on Core Component Infrastructure support. This section describes the minor changes in usage when porting core components between the two systems.

The core components provide three types of APIs:

- Core Component Infrastructure APIs
- Messaging APIs
- Library APIs

For the Resource Manager-only configuration, the Core Component Infrastructure and Library APIs are used. For the configuration where the Core Component Infrastructure is also used, the Core Component Infrastructure and Messaging APIs are used. The use of the Core Component Infrastructure APIs is the same for both configurations except for the `init()` functions provided by each core component.

2.3.5.1 Usage of `init()` and `fini()` Function

The `init()` function initializes the core component. A typical `init()` function has the following prototype:

C Syntax

```
ix_error ix_cc_<component-name>_init(
    ix_cc_handle arg_CcHandle,
    void **arg_ppContext);
```

The only difference between the two configurations is the value assigned to the first argument, `arg_CcHandle`. For a Resource Manager-only configuration, this argument is assigned the value of `IX_CC_HANDLE_L0` which is `#def-ed` in a system header file. For a configuration using the Core Component Infrastructure, this argument is assigned by the Core Component Infrastructure. The value `IX_CC_HANDLE_L0` is an invalid value for `arg_CcHandle` under this configuration.

The use of second argument `arg_ppContext` remains the same in both the configurations.

The `fini()` function terminates the core component. A typical `fini()` function has the following prototype:

C Syntax

```
ix_error ix_cc_<component-name>_fini(
    ix_cc_handle arg_CcHandle,
    void *arg_pContext);
```

The usage remains the same in both configurations. For `arg_CcHandle`, the value `IX_CC_HANDLE_L0` is passed in the Resource Manager-only configuration. When the Core Component Infrastructure is used, the handle that was assigned to the core component in `init()` is passed.

2.3.6 OS Independence of Core Components

This section discusses OS - independence of the core components.

Any software system is dependent on the OS in several areas - memory management, execution context, threads synchronization, inter-process communication and access to the system hardware resources. In terms of file accesses, assumption is that targeted OSs does not provide file system

support and functionality of the core components does not rely on the presence of a file system. The combination of three uniform layers of APIs, described below, gives core components the ability to be portable across VxWorks, Linux kernel, and Linux user mode.

The independence of the core components is achieved by using following layers of abstractions from OS services:

- OSSL, described in the *Intel® Internet Exchange Architecture Portability Framework Developer's Manual*, provides OS-independent API for timers, synchronization primitives, thread allocation and memory management for XScale memory.
- Resource Manager, described in the *Intel® Internet Exchange Architecture Portability Framework Developer's Manual*, provides a set of API for hardware resource allocation, initialization and configuration including:
 - Memory - SRAM, DRAM, Scratch and local memory
 - Hardware Queues and Rings
 - RBUF's, TBUFS's
 - Microengine Management (loading, patching symbols, enable, disable etc.)
 - Buffer management - hardware and software buffers shared between microblocks and core components.
 - Communication with microblocks
- Core Component Infrastructure, described in the *Intel® Internet Exchange Architecture Portability Framework Developer's Manual*, provides a set of APIs for defining the thread of execution for the core component, initialization and termination of the core component. It also provides a mechanism for component to component (inter-process and inter-task) packet and message handling.

Table 2-5 shows how typical core component functionality maps to the services of the these three layers.

Table 2-5. Core Component Functionality Mapped to Different Framework Layers

Typical Functionality of Core Component	Framework Layer	OS services
Initialization, termination, message and packet processing, events and policy handling	Core Component Infrastructure	Execution context, inter-process and inter-task communication, signaling and event support.
Synchronization of data access, allocation and freeing of core memory, low-level timers	OSSL	Thread creation and synchronization, mutexes and semaphores, OS memory allocation and timer services.
Microblock and microengine configuration, creation and accessing shared data between the core components' and microblocks' regions of DRAM and SRAM, Core component/ microblock packet passing, allocation and organization of shared buffers	Resource Manager	Access to the system hardware resources.

These layers needs to be ported to the each OS on which the system is running, and the core component needs to conform to APIs of the different layers in order to remain OS-independent and portable. By using OSSL, Resource Manager and Core Component Infrastructure APIs and data structures, the porting of core components is a simple recompilation step.

There are few components that have more dependency on the OS, including the Stack Driver and the System Application. The System Application is responsible for system start-up and it is OS-dependent in terms how the system is invoked in VxWorks or Linux. The Stack Driver provides network support to the OS-specific local TCP/IP stack.

2.4 High-Level Overview of the Core Components

This section provides a brief overview of the core components described in subsequent chapters of this document. The core components are listed in the order in which they appear in subsequent chapters in this document.

2.4.1 System Application

The system application is responsible for configuring the complete system, including:

- Downloading microcode to the microengines
- Initializing the IXA Portability Framework
- Initializing the Core Components Infrastructure
- Starting the Execution Engines and corresponding core components

In addition, the system application determines and creates configuration information for all core components. If the IXA Portability Framework's registry is used for configuration, the system application populates and configures the registry. The system application runs on both the ingress and egress sides of the reference board.

For complete details, see [Chapter 40, “System Application”](#) of the *Intel® Internet Exchange Architecture Software Building Blocks Developer's Manual*.

2.4.2 POS RX

The POS RX core component does the following:

- Receives and handles any exception packets from the POS RX (receive) microblock.
- Configures the POS RX microblock.
- Configures the IXF6048 POS framer driver. This allows a single OC-48 or quad OC-12 configuration.

For complete details, see [Chapter 41, “POS RX Core Component”](#) of the *Intel® Internet Exchange Architecture Software Building Blocks Developer's Manual*.

2.4.3 CSIX RX

The CSIX RX core component performs the initialization and configuration for the CSIX RX microblock through patching imported symbols and a memory block into the microblock. Patched symbols include, for example, statistics counters such as received packets/frames, dropped packets, received bytes, and so on.

The CSIX RX core component will be designed based on Core Component Infrastructure as described in the *Intel® Internet Exchange Architecture Portability Framework Developer's Manual*. The main functions implemented for supporting this infrastructure are initialization, termination, and message handler functions. In addition to these functions, the CSIX RX core component also provides an interface to the system application for setting and retrieving configuration data.

For complete details, see [Chapter 42, “CSIX RX Core Component”](#) of the *Intel® Internet Exchange Architecture Software Building Blocks Developer's Manual*.

2.4.4 Ethernet RX

The Ethernet RX core component performs the following functions:

- Configures and initializes the Packet RX microblock
- Receives ARP and other exception packets from the Packet RX microblock
- Sends ARP packets over the PCI Bus to the Ethernet TX core component
- Sends other exception packets to another user-defined output core component

For complete details, see [Chapter 43, “Ethernet RX Core Component”](#) of the *Intel® Internet Exchange Architecture Software Building Blocks Developer's Manual*.

2.4.5 CSIX TX

The CSIX TX core component performs the initialization and configuration for the CSIX TX microblock through patching imported symbols and a memory block into the microblock. Patched symbols include, for example, ingress blade ID, Transmit Context (TxC), statistics counters, and so on.

The CSIX TX core component is designed based on the Core Component Infrastructure as described in the *Intel® Internet Exchange Architecture Portability Framework Developer's Manual*. The main functions implemented for supporting this infrastructure are initialization, termination, and message handler functions. In addition to these functions, the CSIX TX core component also provides an interface to the system application for setting and retrieving configuration data.

For complete details, see [Chapter 44, “CSIX TX Core Component”](#) of the *Intel® Internet Exchange Architecture Software Building Blocks Developer's Manual*.

2.4.6 ATM/POS TX

The ATM/POS TX core component performs the initialization and configuration for the ATM TX microblock and the POS TX microblock by patching imported symbols and memory blocks into the microblocks. Patched symbols include, for example, Transmit Context (TxC), statistics counters, and MEv2 port number mask, etc.

The ATM/POS TX core component is designed based on Core Component Infrastructure as detailed in the *Intel® Internet Exchange Architecture Portability Framework Developer's Manual*. The main functions to be implemented for supporting this infrastructure are initialization, termination, and message handler functions.

For complete details, see [Chapter 45, “ATM/POS TX Core Component”](#) of the *Intel® Internet Exchange Architecture Software Building Blocks Developer’s Manual*.

2.4.7 Ethernet TX

The Ethernet TX core component performs transmit functions for the Ethernet interface.

For complete details, see [Chapter 46, “Ethernet TX Core Component”](#) of the *Intel® Internet Exchange Architecture Software Building Blocks Developer’s Manual*.

2.4.7.1 Ethernet ARP Module

The Ethernet ARP module is a library within the Ethernet TX core component. It provides ARP functionality to the Ethernet TX and other core components.

The Ethernet ARP module provides the following services specified in RFC 826, RFC 1122, and RFC 3220:

- Processing of ARP packets
- Layer-2 address resolution
- Creation of dynamic ARP entries
- ARP entry aging
- ARP flooding prevention
- ARP packet queue

For complete details of these and additional functionality of the Ethernet ARP module, see [Chapter 47, “Ethernet ARP Module”](#).

2.4.8 Queue Manager (QM)

Two Queue Manager core components exist in the system—one on the ingress side and another on the egress side—with slight differences in typical data processing. These differences are mentioned explicitly throughout this section and in [Chapter 48, “Queue Manager Core Component”](#) of the *Intel® Internet Exchange Architecture Software Building Blocks Developer’s Manual*. The Ingress Queue Manager core component corresponds with the Cell Based Queue Manager microblock, and the Egress Queue Manager Core component corresponds with the Packet Based Queue Manager microblock. However, on reference boards that provide full duplex capability, the number of Queue Manager core components depends on the system configuration since there may be either one or two Queue Manager core components.

The Queue Manager core component provides the following functionality that is common between both the Ingress and Egress Queue Managers:

- Configuration module for Queue Manager microblock
- Centralized packet enqueueing from core to the microblocks
- Handling of packets enqueued for local output ports in case the switch fabric does not support loopback

As a configuration module, the Queue Manager core component patches symbols and sets up the memory block for the Queue Manager microblock. These configuration values are obtained either statically from the system registry at system startup or during shared memory initialization—that is, during allocation of a memory block and patching of that memory block’s base address. Once configuration values are set for the microblock they do not change during execution.

As a packet enqueueing component, the Queue Manager receives packets from other core components for transmission to the media. The Queue Manager core component uses the Core Component Infrastructure Microblock Communication ID to transfer packets to microblocks. On the ingress side, packets are queued for transmission over the switch fabric. On the egress side, packets are queued for transmission over the POS media.

For packets that are scheduled for transmission on the output port of the same blade, the Queue Manager checks to see if the switch fabric supports loopback. If it does, then packets travel the standard path from the Queue Manager core component to the Queue Manager microblock to the CSIX TX microblock to the switch fabric. If the switch fabric does not do loopback, then the Queue Manager sends packets over PCI to the egress Queue Manager for enqueueing to the output port.

2.4.9 Queue Manager for DiffServ

The Egress Queue Manager core component for DiffServ is similar to the Queue Manager core component detailed in [Chapter 48, “Queue Manager Core Component”](#) of the *Intel® Internet Exchange Architecture Software Building Blocks Developer’s Manual*. The only changes are:

- The queue descriptor size is eight longwords. The core component must take this into account while allocating SRAM memory for the queue descriptor table.
- During initialization, the core component inserts into the system repository the `\\QM\QD_SRAM_BASE` property. The property contains the address of the queue descriptor table in SRAM. The WRED core component reads the property and uses the value to patch `QD_SRAM_BASE` symbol in the WRED microblock. Note that this requires that the Queue Manager core component is initialized before the WRED core component.

2.4.10 Scheduler

Similar to the Queue Manager, two Scheduler core components exist in the system—one on the ingress side and another on the egress side. The differences between the Ingress and the Egress Scheduler are outlined in the [Chapter 50, “Scheduler Core Component”](#) of the *Intel® Internet Exchange Architecture Software Building Blocks Developer’s Manual*. The Ingress Scheduler core component corresponds to the CSIX Scheduler microblock, and the Egress Scheduler core component corresponds to the Egress Packet WRR/DRR Scheduler microblock. In full duplex systems, the number of Scheduler core components depends on system configuration.

- The Scheduler core component is a configuration module for the Scheduler microblock. All configurations are done during system initialization. The Scheduler core component uses the Resource Manager APIs to patch symbols and allocate memory blocks for the Scheduler microblock. These configuration values are obtained either statically from the system registry at system startup or during shared memory initialization—that is, during allocation of a memory block and the patching that memory block’s base address.

2.4.11 Scheduler for DiffServ

The Scheduler core component for DiffServ is virtually identical to the Scheduler core component described in [Chapter 50, “Scheduler Core Component”](#) of the *Intel® Internet Exchange Architecture Software Building Blocks Developer’s Manual*. The only difference is that the Scheduler (DiffServ) core component adds an additional patching symbol.

For details see [Chapter 51, “Scheduler \(DiffServ\) Core Component”](#) of the *Intel® Internet Exchange Architecture Software Building Blocks Developer’s Manual*.

2.4.12 IPv4 Forwarder

The IPv4 Forwarder core component assists the IPv4 microblock to forward packet buffers. The IPv4 Forwarder core component together with the microblock are an implementation of a near-RFC1812 compliant unicast capability on the Intel® IXP2400 and IXP2800 Network Processors. The IPv4 Forwarder core component uses the IXA Portability Framework to interact with its microblock. The IPv4 microblock sends packets which require special handling to the core in the following cases:

- No route information is found for the packet
- When the packet is destined for a local interface
- If IP options are present in header of the packet
- If packet needs to be fragmented

In addition to processing packets which require special handling, the IPv4 Forwarder core component is also responsible for generating ICMP error messages.

The IPv4 Forwarder implements packet and message passing primitives compliant with the Core Component Infrastructure and provides an external API for configuring network interfaces (physical ports). It implements data structures and internal APIs for building and sending ICMP packets.

For complete details, see [Chapter 52, “IPv4 Forwarder Core Component”](#) of the *Intel® Internet Exchange Architecture Software Building Blocks Developer’s Manual*.

2.4.13 IPv6 Forwarder

The IPv6 core component performs the following functions:

- Configures the IPv6 microblock—static configuration.
- Provides message and packet handlers to receive messages and packets from other core components and the IPV6 microblock.
- Generates ICMPv6 error messages.
- Validates an IP header.
- Handles extension headers.
- Supports *neighbor discovery*.
- Supports stateless address automatic configuration.

The microblock performs fast-path forwarding of IPv6 packets. Exception packets—such as packets requiring options processing—are sent to the IPv6 Forwarder core component for special handling.

For details see [Chapter 53, “IPv6 Forwarder Core Component”](#) of the *Intel® Internet Exchange Architecture Software Building Blocks Developer’s Manual*.

2.4.14 IPv6 to IPv4 Tunneling

The tunneling core component helps the tunneling microblocks implement the following types of tunneling:

- Configured tunnels as defined in RFC 2893.
- Automatic tunnels as defined in RFC 2893.
- IPv6toIPv4 tunnels as defined in RFC 3056.

The following functionality is provided:

- Configures the tunneling microblocks.
- Sets up and manages tunneling next-hop information.
- Provides packet handlers to receive packets from the tunneling microblocks and from other core components.
- Provides message handlers to allow configuration of tunneling information.
- Provides reassembly and decapsulation of fragmented tunneled packets.
- Sends ICMPv6 *Packet Too Big* error messages if packet length exceeds the recorded path MTU for a tunnel.
- Reflects ICMP error messages to IPv6 the sender.
- Reports tunneling statistics.

For details see [Chapter 54, “IPv6 To IPv4 Tunneling Core Component”](#) of the *Intel® Internet Exchange Architecture Software Building Blocks Developer’s Manual*.

2.4.15 Six-Tuple Classifier

The six-tuple Exact Match Classifier core component provides the following functionality:

- Initializes and configures the Six-Tuple Classifier microblock by patching symbols.
- Provides an API interface to add and remove exact-match rules. The rules are stored in a hash table shared between the core component and the microblock.
- Implements exact-matching capability. There is no range matching classification in a core component (slow path). Range matching will be available with TCAM support.
- Implements the basic classification function of IP packets without header options in the same way as is done in the microblock. The core component classifies local IP packets generated by the Stack Driver.
- Implements an extended classification function for exception packets thrown by the Six-Tuple microblock. The exception packets are the packets with IP header options.

For complete details see [Chapter 56, “Six-Tuple Classifier Core Component”](#) of the *Intel® Internet Exchange Architecture Software Building Blocks Developer’s Manual*.

2.4.16 Three Color Meter

The Rate Three Color Meter (TCM) core component provides the following functionality:

- Initializes and configures the Three Color Meter microblock by patching symbols.
- Provides an API interface to add, remove and update Three Color Meter instances. The instance parameters are stored in a meter table shared between the Three Color Meter core component and the microblock.

For complete details see [Chapter 57, “Three Color Meter Core Component”](#) of the *Intel® Internet Exchange Architecture Software Building Blocks Developer’s Manual*.

2.4.17 Weighted Random Early Detection (WRED)

The Weighted Random Early Detection (WRED) core component provides the following functionality:

- Initializes and configures the WRED microblock by patching symbols.
- Provides an API interface to add, remove, and update WRED instances. The instance parameters are shared between the core component and the microblock.
- Provides an API interface to read statistics associated with a selected WRED instance.

For details see [Chapter 58, “Weighted Random Early Detection \(WRED\) Core Component”](#) of the *Intel® Internet Exchange Architecture Software Building Blocks Developer’s Manual*.

2.4.18 DSCP Classifier

The core component provides the following functionality:

- Initializes and configures the DSCP classifier microblock by patching symbols.
- Provides an API functions to set classification rules for interfaces and read the current classification rules for interfaces. The rules are stored in a configuration table shared between the core component and the microblock.
- Provides API functions to read per-rule statistics.
- Implements DSCP classification for slow path traffic received from other components (for example local packet from the Stack Driver). If the configuration entry requires remarking DSCP value, the core component changes the DSCP value in the packet.

For complete details see [Chapter 59, “DSCP Classifier Core Component”](#) of *Intel® Internet Exchange Architecture Software Building Blocks Developer’s Manual*.

2.4.19 Route Table Manager

The Route Table Manager (RTM) provides services to the other core components for maintaining route tables. It provides a way to create a new table, delete an existing table, add routes to a table, remove routes from a table, and look up a route in a table.

A route consists of two elements: a network identifier and next hop information. The network identifier, made up of an IP address and network mask, is used in the Route Table Manager's lookup algorithm. The next hop information—which is only stored by the Route Table Manager and is not used internally—is defined by the IPv4 Forwarder microblock (see [Chapter 24, “IPv4 Forwarder Microblock”](#) of the *Intel® Internet Exchange Architecture Software Building Blocks Developer's Manual*).

After routes have been added to the Route Table they may be looked up using an IP address as a key. The algorithm used to find the correct route based on the IP address is not defined at the Route Table Manager level, but rather is a function of the lower-level implementation. The Route Table Manager implements a hardware-based TCAM lookup and a software memory-based longest prefix match (LPM) lookup.

The TCAM and software LPM algorithms will implement a common interface so that at initialization time, the client may choose which to use.

For complete details, see [Chapter 60, “Route Table Manager”](#) of the *Intel® Internet Exchange Architecture Software Building Blocks Developer's Manual*.

2.4.19.1 Next Hop Database

The RTM uses the Next Hop Database (NHDB) as a repository for next hop information. The NHDB provides interfaces to the Route Table Manager—and indirectly to the Route Table Manager's clients—for adding, removing and updating next hop entries. The NHDB is internal to the Route Table Manager, and is not exposed to any core components. However, the number of next hops able to be stored in the NHDB may be configured externally. The NHDB maintains a portion of the memory used by the microengines.

2.4.19.2 TCAM

Ternary Content Addressable Memory (TCAM) is a hardware device for providing high speed search acceleration and can be used for route table look up. The TCAM provides an index for a key (IP address) provided by its client. Using this index the gateway address, next hop ID, and other relevant information can be found in the route table. The communication between TCAMS and clients is based on an Intel®-defined API.

Note: This API model requires no modification of the client's software to work with different TCAMs—for example, IDT, SiberCore, and so on.

2.4.19.3 Software LPM

Only the Longest Prefix Match (LPM) is implemented for the Intel® IXA SDK. This algorithm provides trie tables matching the tables described by the IPv4 Forwarder microblock—see [Chapter 24, “IPv4 Forwarder Microblock”](#) of the *Intel® Internet Exchange Architecture Software Building Blocks Developer's Manual*. Because the currently planned Software LPM is designed for IPv4, all exposed function names are defined in such a way that they will not conflict with IPv6 function names.

2.4.20 Route Table Manager for IPv6

The Route Table Manager for IPv6 (RTMv6) stores and retrieves data on behalf of the IPv6 Forwarder core component. In addition, the RTMv6 stores the data in a format consistent with the requirements of the IPv6 Forwarder microblock. RTMv6 uses the Lookup library API to access the route tables for IPv6. To manage next hops, RTMv6 uses a Next Hop Database (NHDB)—see [Section 2.4.19](#). The RTMv6 exposes an API allowing the IPv6 core component to add to, remove from, or lookup items in the route table. This API is implemented as a library supporting a single client.

This implementation of RTMv6 is specific to the IP, version 6. Because an RTMv6 designed for IPv6 would require a different interface—for example, 128-bit parameters instead of 32-bit—it exposes its entry points in such a way as to indicate that it only deals with IPv6 data types and algorithms. This avoids naming conflicts with an RTMv6 designed for IPv4. For details on the Route Table Manager for IPv4 refer to [Chapter 60, “Route Table Manager”](#) of the *Intel® Internet Exchange Architecture Software Building Blocks Developer’s Manual*.

For details on the Route Table Manager for IPv6 see [Chapter 61, “Route Table Manager for IPV6 Core Component”](#) of the *Intel® Internet Exchange Architecture Software Building Blocks Developer’s Manual*.

2.4.21 L2 Table Manager

The L2 Table Manager provides an API to manage, add, and delete entries in the L2 Table. The L2 Table corresponds to the Route Table and Next Hop Database on the ingress side. See [Chapter 60, “Route Table Manager”](#) of the *Intel® Internet Exchange Architecture Software Building Blocks Developer’s Manual* for details on the Next Hop Database.

For complete details see [Chapter 62, “L2 Table Manager”](#) of the *Intel® Internet Exchange Architecture Software Building Blocks Developer’s Manual*.

2.4.22 Message Helper and Support Library

The Message Helper and Support library is the layer of translation between messages and core component APIs. This library specifies the handling of function arguments and the returning result back to the client. The Message Helper and Support Library provides a C-language interface to the clients of the core components and use IXA Portability Framework’s message mechanism to invoke internal core component API function. The Message Helper library translates the message API into the messages and enforces the mechanism of delivering messages to the Library APIs of each core component and sending the result back to the clients. It helps clients to simplify the programming model by hiding the marshaling of data inside the messages.

For complete details, see [Chapter 63, “Message Helper and Support Library”](#) of the *Intel® Internet Exchange Architecture Software Building Blocks Developer’s Manual*.

2.4.23 Stack Driver

The Stack Driver provides a core component compatible abstraction of the operating system's network stack. It is a specialized type of core component that allows input to and output from any network stack—IP, Appletalk*, IPX—and other core components. It appears to the network stack as a media driver and appears to the IXA SDK as a core component. The main functions of the Stack Driver are:

- To send data received by the Stack Driver from an upstream core component up to the operating system's network stack.
- To send data received by the Stack Driver from the operating system's network stack to the bounded downstream core component.

The operating system's network stack with which the Stack Driver interacts can be either local or remote. The current design describes its interface with an operating system specific IP stack.

For complete details, see [Chapter 64, “Stack Driver”](#) of the *Intel® Internet Exchange Architecture Software Building Blocks Developer's Manual*.

2.4.24 SoftSAR Core Components

The SoftSAR core components include the following sub-components:

- [Section 2.4.24.1, “SAR Core Components”](#)
- [Section 2.4.24.2, “ATM RX Core Components”](#)
- [Section 2.4.24.3, “ATM TX Core Components”](#)
- [Section 2.4.24.4, “TM4.1 Core Components”](#)

For more information on the ATM RX microblock and core component design see [Chapter 6, “ATM AAL5 RX Microblock.”](#) and [Chapter 65, “SoftSAR Core Components”](#) of *Intel® Internet Exchange Architecture Software Building Blocks Developer's Manual*.

2.4.24.1 SAR Core Components

The SoftSAR core components provide the following functionalities:

- Utilization of SAR Plug-in API
- Provides front-end SAR Control API for user modules/applications
- Provides mechanism for registration of SAR Control plug-ins
- Controls SAR Control Agent (if it exists in the system)
- Distributes a user request to local SAR Control plug-ins and to SAR Control Agent (if it exists in the system)

2.4.24.2 ATM RX Core Components

This section describes the high level design for the ATM Receive Core Component. The ATM RX Core Component runs on the ingress side and performs the following functions:

- Performs initialization/configuration of ATM RX microblock and patch symbols.

- Provides an interface to a system application for setting and retrieving configuration and statistical parameters.
- Keeps track of the Virtual Circuit (VC) establish/teardown. This information is used to maintain the RXC table and the hash lookup table for looking up the Receive Context (RXC) for a particular VC.
- Handles exception packets from the ATM RX microblock.
- Supports port (up to 2048) and VCQ#.

For more information on the ATM RX microblock and core component design see [Chapter 6, “ATM AAL5 RX Microblock.”](#) of *Intel® Internet Exchange Architecture Software Building Blocks Developer’s Manual*.

2.4.24.3 ATM TX Core Components

The ATM TX Core Component performs the following functions:

- Initializes and configures the ATM TX microblock through patching symbols
- Provides interfaces for setting and retrieving the ATM TX interface state and statistical parameters
- Initializes and configures the ATM framer device

For information on ATM TX microblocks, see [Chapter 11, “ATM AAL5 TX Microblock.”](#) of the *Intel® Internet Exchange Architecture Software Building Blocks Developer’s Manual*.

2.4.24.4 TM4.1 Core Components

This section describes the high level design of the TM4.1 core component. The following functionalities are supported:

- Provides an API interface to add and remove ports and VCs. VCs description and ports description are stored in tables shared between the core component and the microblocks.
- Defines algorithms for setting TM4.1 microblocks data that provide even cell transmit for ports and for VC with real-time related conformance parameters.

For information on the TM4.1 microblock, see [Chapter 23, “TM4.1 Shaper and Scheduler Microblock”](#) of the *Intel® Internet Exchange Architecture Software Building Blocks Developer’s Manual*.

2.4.25 MPLS Forwarder Core Component

The MPLS Forwarder core component receives packets and messages from the MPLS Forwarder microblocks (ILM Forwarder and FTN Forwarder).

The MPLS Forwarder core component performs the following functions on behalf of the ILM Forwarder and FTN Forwarder microblocks:

- Initializes and maintains the data structures for the ILM Forwarder and FTN Forwarder microblocks
- Configures the ILM Forwarder and FTN Forwarder microblocks (static configuration)

- Provides message and packet handlers to receive messages from the control plane and packets from the ILM Forwarder and FTN Forwarder microblocks
- Handles fragmentation of MPLS packets
- Implements "slow path" forwarding for fragmented MPLS packets
- Handles packets with special MPLS label values

For complete details see [Chapter 66, "MPLS Forwarder Core Component"](#) of the *Intel® Internet Exchange Architecture Software Building Blocks Developer's Manual*.

The System Application configures the system and provides the following services:

- Initializes the Core Component Infrastructure.
- Provides a set of hooks into various stages of initialization to allow easy addition of user-provided initialization code.
- Loads microcode and starts the microengines.
- Allocates a set of scratch rings based on the system configuration.
- Allocates various free lists and provides a message helper function to allow core components to access them.
- Provides 'Property Master' services for system properties that do not have a clear core component master.
- Creates execution engines based on a user provided configuration.
- Instantiates core components within execution engines based on a user provided configuration.
- Initializes support facilities.
- Provides proper system shutdown.
- Allows re-initialization of the system.

For complete details see [Chapter 40, “System Application”](#) of the *Intel® Internet Exchange Architecture (IXA) Software Building Blocks Developer's Manual*.

3.1 Start and Shut Down the System Application

Table 3-1 shows the function calls that start and shuts down the system application.

Table 3-1. Starting and Shutting Down System Application

API	Description
<code>ix_sa_create()</code>	Starts the system application by way of creating a thread with an entry point of <code>_ix_sa_entry()</code> .
<code>_ix_sa_entry()</code>	Starts the system application.
<code>ix_sa_shutdown()</code>	Signals the system application to shutdown the system.

3.1.1 `ix_sa_create()`

Starts the system application by way of creating a thread with an entry point of `_ix_sa_entry`.

C Syntax

```
int ix_sa_create(ix_uint32 arg_bladeID);
```

Input

<code>arg_bladeID</code>	Blade ID of this blade.
--------------------------	-------------------------

Output

None.

Returns

None.

3.1.2 `_ix_sa_entry()`

Starts the system application. This function does not return until the system is shutdown. The argument is a `void *` because the prototype is set as a thread entry point. Internally the blade ID is cast to the correct type, `ix_uint32`. Calling this function directly from the VxWorks Shell is useful for debugging.

C Syntax

```
ix_error _ix_sa_entry (void *arg_bladeID);
```

Input

<code>arg_bladeID</code>	Blade ID of this blade.
--------------------------	-------------------------

Output

None.

Returns:

None.

3.1.3 ix_sa_shutdown()

Signals the system application to shutdown the system. If a special override is provided, the application calling `ix_sa_shutdown` can specify whether the system should automatically restart.

C Syntax

```
ix_error ix_sa_shutdown (ix_sa_restart_override arg_restart);
```

Input

`arg_restart` A value from the following enumeration:

```
typedef enum
{
    IX_SA_RESTART_NO = 0 /*Do NOT restart */
    ,IX_SA_RESTART_FORCE /* Force a restart */
    ,IX_SA_RESTART_DEFAULT /* use system default */
} ix_sa_restart_override;
```

Output

None.

Returns

None.

3.2 Loading Microcode and Starting Microengines

Table 3-2 shows the function call for the loading microcode and starting the microengines.

Table 3-2. Loading Microcode and Starting Engines

API	Description
<code>ix_sa_start_microengines()</code>	Makes the resource manager call(s) to start the microengines when the first logical interface is configured with an IP address.

3.2.1 ix_sa_start_microengines()

This function is called by an external source—in this release, the Stack Driver—when the system is ready to start the microengines. In this release, the microengines is started until at least one logical interface is configured with an IP address. Thus the Stack Driver, which is the Property Master of IP addresses, calls this function once the first logical interface is configured with an IP address. Then this function makes the resource manager call(s) to start the microengines.

C Syntax

```
ix_error ix_sa_start_microengines (void);
```

Input

none

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • A valid <code>ix_error</code>—the operation failed.
--------------	---

3.3 User Initialization and Shutdown Hooks

A system designer may have a variety of initialization tasks beyond what the system application provides by default. A series of initialization and shutdown hooks are provided at various stages of initialization. These hooks allow a system designer to easily add additional custom code without the need to modify the core system application. The use of these hooks is by no means required but does provide a well defined, tested way to add additional initialization and shutdown code.

Table 3-3 shows the user initialization and shutdown hooks.

Table 3-3. User Initialization and Shutdown Hooks

API	Description
<code>ix_sa_init_hook_first()</code>	Called before the Core Component Infrastructure is initialized.
<code>ix_sa_init_hook_pre_ee()</code>	Called before starting any execution engine or core components
<code>ix_sa_init_hook_ee()</code>	Called from the execution engine context.
<code>ix_sa_init_hook_pre_me()</code>	Called before the Microengines are started.
<code>ix_sa_init_hook_last()</code>	Called after all the initialization is complete.
<code>ix_sa_shutdown_hook_first()</code>	Called before the shutdown begins but before the system application performs any shutdown.
<code>ix_sa_shutdown_hook_post_me()</code>	Called after the microengines are stopped.
<code>ix_sa_shutdown_hook_post_ee()</code>	Called before the Core Component Infrastructure is shutdown.
<code>ix_sa_shutdown_hook_ee()</code>	Called from the context of each execution engine before any core components are shutdown.
<code>ix_sa_shutdown_hook_last()</code>	Called after all shutdown activity is completed.

3.3.1 ix_sa_init_hook_first()

This function is called before anything is initialized—that is, before the Core Component Infrastructure is initialized.

C Syntax

```
ix_error ix_sa_init_hook_stage (void **context);
```

Input

none

Output/Returns

context	A calling application-provided context. This pointer is stored and passed to the last stage shutdown hook.
Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none">• <code>IX_SUCCESS</code>—the operation succeeded.• A valid <code>ix_error</code>—the operation failed.

3.3.2 ix_sa_init_hook_pre_ee()

This function is called before any execution engines or Core Components are started.

C Syntax

```
ix_error ix_sa_init_hook_pre_ee (void **context);
```

Input

none

Output

context	A calling application-provided context. This pointer is stored and passed to the <code>post_ee</code> stage shutdown hook.
Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none">• <code>IX_SUCCESS</code>—the operation succeeded.• A valid <code>ix_error</code>—the operation failed.

3.3.3 `ix_sa_init_hook_ee()`

This function is called from the execution engine context from the 'init' function. This happens before any core components are created.

C Syntax

```
ix_error ix_sa_init_hook_ee (void **context);
```

Input

none

Output

context	A calling application-provided context. This pointer is stored and passed to the <code>ee</code> stage shutdown hook.
Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • A valid <code>ix_error</code>—the operation failed.

3.3.4 `ix_sa_init_hook_pre_me()`

This function is called before the Microengines are started.

C Syntax

```
ix_error ix_sa_init_hook_pre_me (void **context);
```

Input

none

Output

context	A calling application-provided context. This pointer is stored and passed to the <code>post_me</code> stage shutdown hook.
Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • A valid <code>ix_error</code>—the operation failed.

3.3.5 ix_sa_init_hook_last()

This function is called after all initialization is complete.

C Syntax

```
ix_error ix_sa_init_hook_stage (void **context);
```

Input

none

Output

context	A calling application-provided context. This pointer is stored and passed to the first stage shutdown hook.
Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none">• <code>IX_SUCCESS</code>—the operation succeeded.• A valid <code>ix_error</code>—the operation failed.

3.3.6 ix_sa_shutdown_hook_first()

This function is called when shutdown begins but before the system application performs any shutdown tasks. (Pre-execution engine)

C Syntax

```
ix_error ix_sa_shutdown_hook_first (void *context);
```

Input

context	The context returned from the last stage initialization hook.
---------	---

Output

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none">• <code>IX_SUCCESS</code>—the operation succeeded.• A valid <code>ix_error</code>—the operation failed.
--------------	--

3.3.7 ix_sa_shutdown_hook_post_me()

This function is called after the microengines are stopped.

C Syntax

```
ix_error ix_sa_shutdown_hook_post_me (void *context);
```

Input

context	The context returned from the <code>pre_me</code> stage initialization hook.
---------	--

Output

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none">• <code>IX_SUCCESS</code>—the operation succeeded.• A valid <code>ix_error</code>—the operation failed.
--------------	--

3.3.8 ix_sa_shutdown_hook_post_ee()

This function is called before core-component infrastructure is shut down.

C Syntax

```
ix_error ix_sa_shutdown_hook_post_ee (void *context);
```

Input

context	The context returned from the <code>pre_ee</code> stage initialization hook.
---------	--

Output

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none">• <code>IX_SUCCESS</code>—the operation succeeded.• A valid <code>ix_error</code>—the operation failed.
--------------	--

3.3.9 ix_sa_shutdown_hook_ee()

This function is called from the context of each execution engine—that is, from inside the fini function—and before any core-components are shutdown.

C Syntax

```
ix_error ix_sa_shutdown_hook_stage (void *context);
```

Input

context	The context returned from the ee stage initialization hook.
---------	---

Output

Return Value	Returns a valid ix_error. <ul style="list-style-type: none">• IX_SUCCESS—the operation succeeded.• A valid ix_error—the operation failed.
--------------	--

3.3.10 ix_sa_shutdown_hook_last()

This function is called after all shutdown activity is completed.

C Syntax

```
ix_error ix_sa_shutdown_hook_last (void *context);
```

Input

context	The context returned from the first stage initialization hook.
---------	--

Output

Return Value	Returns a valid ix_error. <ul style="list-style-type: none">• IX_SUCCESS—the operation succeeded.• A valid ix_error—the operation failed.
--------------	--

Receive

The following chapters are included in this section:

- [Chapter 4, “POS RX API”](#)
POS RX Core Component corresponds to the POS RX microblock. This component configures the POS RX microblock.
- [Chapter 5, “CSIX RX”](#)
CSIX RX Core Component performs initialization and configuration for the CSIX RX microblock through patching imported symbols and memory block into the microblock.
- [Chapter 6, “Ethernet RX”](#)
Ethernet RX Core Component handles the ARP exception packets received from the Ethernet RX microblock and also takes care of configuring the Packet RX microblock. In addition, Ethernet RX Core Component maintains the Ethernet MAC filtering table for MAC address filtering used by the microblock.

The POS RX core component performs the following functions:

- Initializes and configures the Packet RX microblock and patches symbols for shared use.
- Interfaces to a system application for setting and retrieving configuration and statistical parameters.
- Handles exception packets from the Packet RX microblock.

The Packet RX microblock on the Ingress IXP2400 performs frame reassembly on the mpackets coming in on the media interface. It uses eight threads on one single microengine. Each thread handles one mpacket at a time. The microblock is written such that it supports up to 16 virtual ports, one or more of which may be unused. This allows the microblock to support different configurations such as Quad-OC12, 16 OC-3 or a single OC-48 port.

For complete details, see [Chapter 41, “POS RX Core Component”](#) of the *Intel® Internet Exchange Architecture Software Building Blocks Developer's Manual*.

4.1 Core Component Infrastructure API

Table 4-1 summarizes the POS RX Core Component Infrastructure API.

Table 4-1. POS RX Core Component Infrastructure API

Name	Description
<code>ix_cc_pos_rx_init()</code>	Initializes the core component.
<code>ix_cc_pos_rx_fini()</code>	Terminates the core component.
<code>ix_cc_pos_rx_msg_handler()</code>	The message handler function for the POS RX core component.
<code>ix_cc_pos_rx_pkt_handler()</code>	The exception packet handler for packets sent to this core component by the POS RX microblock.

4.1.1 `ix_cc_pos_rx_init()`

Initializes the core component. It should be called and successfully returned before any other function in the core component interface is called. It performs static configuration for the POS RX Microblock by allocating and patching required variables and a memory block into the microblock. This is accomplished by calling the Resource Manager API. On return from the function, the POS RX interface is enabled.

C Syntax

```
ix_error ix_cc_pos_rx_init(
    ix_cc_handle arg_CcHandle,
    void **arg_ppContext);
```

Input

`arg_CcHandle` Handle to the core component.

Input/Output

`arg_ppContext` This is an INPUT and OUTPUT argument.

- As an input, this is a pointer to a generic initialization context which is used to extract system configuration information.
- As an output, this is a pointer to the location where the pointer to the control block is stored. This is allocated by the core component. The control block is internal to the core component and contains a RX Context memory and other variables and internal data structures. This pointer is later passed into the [`ix_cc_pos_rx_fini\(\)`](#) function by the Core Component Infrastructure to free memory when the core component is terminated.

Output/Returns

Return Value Returns a valid `ix_error`.

- `IX_SUCCESS`—the operation succeeded.
- A valid `ix_error`—the operation failed.

4.1.2 `ix_cc_pos_rx_fini()`

Terminates the core component. It is executed when the execution engine must shut down the core component. This function frees all allocated memory and resources obtained in the `ix_cc_pos_rx_init()` function. Before returning, the POS RX interface is disabled.

C Syntax

```
ix_error ix_cc_pos_rx_fini(  
    ix_cc_handle arg_CcHandle,  
    void *arg_pContext);
```

Input

<code>arg_CcHandle</code>	The handle to the core component.
<code>arg_pContext</code>	A pointer to the control block memory allocated earlier in <code>ix_cc_pos_rx_init()</code> . The termination routine uses this pointer to deallocate the control block memory.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none">• <code>IX_SUCCESS</code>—the operation succeeded.• A valid <code>ix_error</code>—the operation failed.
--------------	--

4.1.3 ix_cc_pos_rx_msg_handler()

This is the message handler function for the POS RX core component.

C Syntax

```
ix_error ix_cc_pos_rx_msg_handler(  
    ix_buffer_handle arg_Msg,  
    ix_uint32 arg_UserData,  
    void *arg_pContext);
```

Input

arg_Msg	Buffer handle embedding information for the invocation of an associated library API function.
arg_UserData	Message type. The currently defined messages include: <ul style="list-style-type: none">• IX_CC_POS_RX_MSG_GET_INTERFACE_STATE—request to obtain the interface state.• IX_CC_POS_RX_MSG_GET_STATISTICS_INFO—request to obtain statistics information.
arg_pContext	The handle to the core component created internal context structure.

Output/Returns

Return Value	Returns a valid ix_error. <ul style="list-style-type: none">• IX_SUCCESS—the operation succeeded.• A valid ix_error—the operation failed.
--------------	--

4.1.4 `ix_cc_pos_rx_pkt_handler()`

This function handles the exception packets sent to the core component by the POS RX Microblock. These exception packets are sent to the output defined for this component. Any application or component that needs to capture these packets can *bind* to this output using the appropriate symbolic constant in the `bindings.h` file.

C Syntax

```
ix_error ix_cc_pos_rx_pkt_hdlr (ix_buffer_handle buffer,
                               ix_uint32 exception_code, void *ctx)
```

Input

<code>buffer</code>	A handle to the packet buffer.
<code>exception_code</code>	<code>PPP_CONTROL_PACKET</code>
<code>ctx</code>	A pointer to a context. The context may be a user-defined structure—that is passed to the core component when a packet arrives. This context is defined by the core component and passed through the <code>ix_cc_add_packet_handler_list()</code> function—see Section 2.3.4.1 , “ <code>ix_cc_add_packet_handler_list()</code> ”—called in the initialization function of the core component’s execution engine.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. A valid <code>ix_error</code>—the operation failed.
--------------	---

4.2 Messaging API

This section describes the messaging API provided by the POS RX core component.

Table 4-2. POS RX Messaging API

Name	Description
<code>ix_cc_pos_rx_async_get_statistics_info()</code>	Returns statistical information.
<code>ix_cc_pos_rx_async_get_interface_state()</code>	Returns the POS interface state for a specific port.

4.2.1 `ix_cc_pos_rx_async_get_statistics_info()`

Returns statistical information for the POS RX Microblock and the POS RX core component.

C Syntax

```
ix_error ix_cc_pos_rx_async_get_statistics_info(
    ix_cc_pos_rx_statistics_info_context *arg_pMgInfoContext,
    ix_cc_pos_rx_cb_get_statistics_info arg_Callback);
```

Input

<code>arg_pMgInfoContext</code>	A pointer to a data structure of type <code>ix_s_cc_pos_rx_statistics_info_context</code> specifying the type of statistics information requested, an index to a virtual circuit, and a pointer identifying the caller.
<code>arg_Callback</code>	A pointer to the callback routine. The function prototype of the callback is defined by <code>ix_cc_pos_rx_cb_get_statistics_info</code> .

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • A valid <code>ix_error</code>—the operation failed.
--------------	---

4.2.1.1 `ix_s_cc_pos_rx_statistics_info_context`

This data structure specifies the type of statistics information requested, the index to a virtual circuit, and the requestor. It is used with `ix_cc_pos_rx_statistics_info`, an enumeration listing the supported types of POS RX statistics.

C Syntax

```
typedef struct ix_s_cc_pos_rx_statistics_info_context {
    ix_cc_pos_rx_statistics_info entity;
    ix_uint32 index,
    void *pUserContext;
} ix_cc_pos_rx_statistics_info_context;
```

Input

entity	Specifies the type of statistics information. For a list of the types of information currently supported see Section 4.2.1.1.1 , “ <code>ix_cc_pos_rx_statistics_info</code> .”
index	Specifies the index number for the virtual circuit.
pUserContext	A pointer to a user-provided context structure which is used by the core component in callback functions to identify the originator of the request.

4.2.1.1.1 `ix_cc_pos_rx_statistics_info`

The enumerated type that defines the supported types of statistics information for the Pos RX core component.

C Syntax

```
typedef enum ix_e_cc_pos_rx_statistics_info {
    IX_CC_POS_RX_PACKETS = 0,
    IX_CC_POS_RX_PACKETS_DROPPED,
    IX_CC_POS_RX_PACKETS_EXCEPTION,
    IX_CC_POS_RX_BYTES,
    IX_CC_POS_RX_ALL_COUNTERS,
} ix_cc_pos_rx_statistics_info;
```

Values

<code>IX_CC_POS_RX_PACKETS</code>	The RX packet count.
<code>IX_CC_POS_RX_PACKETS_DROPPED</code>	The dropped packet count.
<code>IX_CC_POS_RX_PACKETS_EXCEPTION</code>	The exception packet count.
<code>IX_CC_POS_RX_BYTES</code>	The RX byte count.
<code>IX_CC_POS_RX_ALL_COUNTERS</code>	Specifies all supported counters for a port.

4.2.1.2 `ix_cc_pos_rx_cb_get_statistics_info`

This is the prototype of callback functions for calls to `ix_cc_pos_rx_async_get_statistics_info()`.

C Syntax

```
ix_error (*ix_cc_pos_rx_cb_get_statistics_info)(
    ix_error arg_Result,
    void *arg_pContext,
    ix_uint64 *arg_pBuffer);
```

Input

<code>arg_pContext</code>	A pointer to the user-provided context.
<code>arg_pBuffer</code>	A pointer to the buffer that stores the returned statistics information.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • A valid <code>ix_error</code>—the operation failed.
--------------	---

4.2.2 `ix_cc_pos_rx_async_get_interface_state()`

Returns the current state of the POS RX interface. This is an asynchronous function and the current state of the interface is returned through the user-provided callback function.

C Syntax

```
ix_error ix_cc_pos_rx_async_get_interface_state(
    ix_cc_pos_rx_if_state_context *arg_pStateContext,
    ix_cc_pos_rx_cb_get_interface_state arg_Callback);
```

Input

<code>arg_pStateContext</code>	A pointer to a data structure specifying the port whose interface state information is to be obtained and a calling context. See ix_s_cc_pos_rx_if_state_context .
<code>arg_Callback</code>	A pointer to the callback routine. The function prototype is shown in Section 4.2.2.1 , “ <code>ix_cc_pos_rx_cb_get_interface_state</code> .”

Output/Returns

Return Value	Returns a valid <code>ix_error</code> .
	<ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • A valid <code>ix_error</code>—the operation failed.

4.2.2.1 `ix_cc_pos_rx_cb_get_interface_state`

The function prototype for the get interface state callback function.

C Syntax

```
ix_error (*ix_cc_pos_rx_cb_get_interface_state)(
    ix_error arg_Result,
    void *arg_pContext,
    ix_cc_pos_rx_if_state *arg_pState);
```

Input

<code>arg_Result</code>	The error code returned by the POS RX core component for the result of the interface state request—its value is either <code>IX_SUCCESS</code> for success or a valid <code>ix_error</code> for failure.
<code>arg_pContext</code>	A pointer to the user-provided context.
<code>arg_pState</code>	A pointer to the data structure used to store the returned interface state.

4.2.2.2 `ix_s_cc_pos_rx_if_state_context`

This data structure specifies an interface port and a calling context.

C Syntax

```
typedef struct ix_s_cc_pos_rx_if_state_context {
    ix_uint8 port;
    void *pUserContext;
} ix_cc_pos_rx_if_state_context;
```

Input

<code>port</code>	Specifies the interface port number whose interface state is to be returned.
<code>pUserContext</code>	A pointer to a user-provided context structure which is used by the core component in the callback function to identify the originator of the request.

4.2.2.3 ix_cc_pos_rx_if_state

The enumerated type specifying states of a POS RX interface.

C Syntax

```
typedef enum ix_e_cc_pos_rx_if_state {
    IX_CC_POS_RX_IF_STATE_DOWN = 0;
    IX_CC_POS_RX_IF_STATE_UP;
} ix_cc_pos_rx_if_state;
```

Values

IX_CC_POS_RX_IF_STATE_DOWN	The POS RX interface is down.
IX_CC_POS_RX_IF_STATE_UP	The POS RX interface is up.

4.3 Library API

This section describes the POS RX Library API. This API is summarized in [Table 4-3](#).

Table 4-3. POS RX Library API

Name	Description
ix_cc_pos_rx_get_statistics_info()	Returns the requested statistics information.
ix_cc_pos_rx_get_interface_state()	Returns the current state of the POS RX interface for a specified physical port.

4.3.1 ix_cc_pos_rx_get_statistics_info()

Returns the requested statistics information.

C Syntax

```
ix_error ix_cc_pos_rx_get_statistics_info(
    ix_cc_pos_rx_statistics_info arg_entity,
    ix_cc_statistics_info_data *arg_pBuffer,
    void *arg_pContext);
```

Input

arg_entity	The type of statistics information. See Section 4.3.1.1 , “ ix_e_cc_pos_rx_statistics_info ”.
arg_pBuffer	A pointer to the data structure used to return the requested entity.
arg_pContext	A pointer to the core component control block memory allocated earlier in the call to the ix_cc_pos_rx_init() function

Output/Returns

Return Value	Returns a valid <code>ix_error</code> .
	<ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • A valid <code>ix_error</code>—the operation failed.

4.3.1.1 `ix_e_cc_pos_rx_statistics_info`

This enumeration lists the types of statistics supported by `ix_cc_pos_rx_get_statistics_info()`.

C Syntax

```
typedef enum ix_e_cc_pos_rx_statistics_info {
    IX_CC_POS_RX_PACKETS = 0,
    IX_CC_POS_RX_PACKETS_DROPPED,
    IX_CC_POS_RX_PACKETS_EXCEPTION,
    IX_CC_POS_RX_BYTES,
    IX_CC_POS_RX_ALL_COUNTERS,
} ix_cc_pos_rx_statistics_info;
```

Values

<code>IX_CC_POS_RX_PACKETS</code>	The RX packet count.
<code>IX_CC_POS_RX_PACKETS_DROPPED</code>	The dropped packet count.
<code>IX_CC_POS_RX_PACKETS_EXCEPTION</code>	The exception packet count.
<code>IX_CC_POS_RX_BYTES</code>	The Rx byte count.
<code>IX_CC_POS_RX_ALL_COUNTERS</code>	All supported counters for the port.

4.3.1.2 `ix_s_cc_statistics_info_data`

A data structure specifying the data buffer used to return statistics information.

C Syntax

```
typedef struct ix_s_cc_statistics_info_data {
    ix_uint32 dataLength;
    void *pDataBuffer;
} ix_cc_statistics_info_data;
```

Data Members

<code>dataLength</code>	The length in bytes of the data buffer.
<code>pDataBuffer</code>	A pointer to a buffer that is used to return the statistics data. It is the responsibility of the calling application to allocate the memory required for the <code>pDataBuffer</code> pointer.

4.3.2 `ix_cc_pos_rx_get_interface_state()`

Returns the current state of the POS RX interface.

C Syntax

```
ix_error ix_cc_pos_rx_get_interface_state(
    ix_cc_pos_rx_if_state *arg_pState,
    void *arg_pContext);
```

Input

<code>arg_pContext</code>	A pointer to the core component control block memory allocated earlier in the call to the <code>ix_cc_pos_rx_init()</code> function
---------------------------	---

Output/Returns

<code>arg_pState</code>	A pointer to the structure which, on return from the call, reports the current state of the interface. See Section 4.2.2.3 , “ <code>ix_cc_pos_rx_if_state</code> ”.
Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. A valid <code>ix_error</code>—the operation failed.

The CSIX RX core component performs the following functions:

- Performs initialization and configuration of CSIX RX microblock through patching symbols.
- Provides an interface to the System Application for setting and retrieving configuration and statistical parameters.

The CSIX RX core component does not process data packets since it has no exception packets from the CSIX RX microblock or neighboring core components. For complete details see [Chapter 42, “CSIX RX Core Component”](#) of the *Intel® Internet Exchange Architecture (IXA) Software Building Blocks Developer's Manual*.

5.4 Core Component Infrastructure API

Table 5-1 lists the core component infrastructure API provided by CSIX RX core component.

Table 5-1. Core Components of the CSIX API

API	Description
<code>ix_cc_csix_rx_init()</code>	Initializes the core component
<code>ix_cc_csix_rx_fini()</code>	Terminates the core component
<code>ix_cc_csix_rx_msg_handler()</code>	Handles messages for interface state and statistics

5.4.1 `ix_cc_csix_rx_init()`

This function initializes the core component. It should be called and returned before any other function in the core component API can be called. It performs static configuration for the microblock by allocating and patching the required variables and memory blocks by calling the Resource Manager API.

C Syntax

```
ix_error ix_cc_csix_rx_init(  
    ix_cc_handle arg_CcHandle,  
    void **arg_ppContext);
```

Input

`arg_CcHandle` Handle to the core component.

Input/Output

<code>arg_ppContext</code>	<p>This is an INPUT and OUTPUT argument.</p> <ul style="list-style-type: none"> As an Input argument, it is a pointer to a generic init context used to extract system configuration information. As an Output argument, it is the location which stores the pointer to the control block allocated by the core component. The control block is internal to the core component and contains RX Context memory, their variables, and internal data structures. This pointer is used later to be passed into the <code>ix_cc_csix_rx_fini()</code> function by Core Component Infrastructure to free memory before the core component is terminated.
----------------------------	--

Output/Returns

Return Value	<p>Returns a valid <code>ix_error</code>.</p> <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. A valid <code>ix_error</code>—the operation failed.
--------------	---

5.4.2 `ix_cc_csix_rx_fini()`

The function terminates the core component. It is executed when the execution engine running the core component is being shut down. This function frees all the allocated memory and resources obtained in `ix_cc_csix_rx_init()` function.

C Syntax

```
ix_error ix_cc_csix_rx_fini(
    ix_cc_handle arg_CcHandle,
    void *arg_pContext);
```

Input

<code>arg_CcHandle</code>	Handle to the core component.
<code>arg_pContext</code>	Pointer to the allocated memory control block using <code>ix_cc_csix_rx_init()</code> . The termination routine uses the parameter to deallocate the control block memory.

Output/Returns

Return Value	<p>Returns a valid <code>ix_error</code>.</p> <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. A valid <code>ix_error</code>—the operation failed.
--------------	---

5.4.3 ix_cc_csix_rx_msg_handler()

The function is the message handler for the CSIX RX core component.

C Syntax

```
ix_error ix_cc_csix_rx_msg_handler(  
    ix_buffer_handle arg_Msg,  
    ix_uint32 arg_UserData,  
    void *arg_pContext);
```

Input

arg_Msg	The buffer handle embedding information for calling the associated library API function.
arg_UserData	Message type: <ul style="list-style-type: none">• IX_CC_CSIX_RX_MSG_GET_STATISTICS_INFO—obtain the statistics info
arg_pContext	Handle to the core component created using an internal context structure.

Output/Returns

Return Value	Returns a valid ix_error. <ul style="list-style-type: none">• IX_SUCCESS—the operation succeeded.• A valid ix_error—the operation failed.
--------------	--

5.5 Messaging API

Table 5-2 describes the function calls of the Messaging API provided by CSIX RX core component.

Table 5-2. Messaging API Function Calls of the CSIX RX Core Component

API	Description
<code>ix_cc_csix_rx_async_get_statistics_info()</code>	Obtains the statistical information

5.5.1 `ix_cc_csix_rx_async_get_statistics_info()`

The function returns the statistics information requested.

C Syntax

```
ix_error ix_cc_csix_rx_async_get_statistics_info (
    ix_cc_csix_rx_statistics_info_context *arg_pMgInfoContext,
    ix_cc_csix_rx_cb_get_statistics_info arg_Callback);
```

Input

`arg_pMgInfoContext` See [Section 5.5.1.1](#),
“`ix_s_cc_csix_rx_statistics_info_context`”.

`arg_Callback` Specifies the pointer to the callback routine. See [Section 5.5.1.2](#),
“`ix_cc_csix_rx_cb_get_statistics_info`”.

Output/Returns

Return Value Returns a valid `ix_error`.

- `IX_SUCCESS`—the operation succeeded.
- A valid `ix_error`—the operation failed.

5.5.1.1 **ix_s_cc_csix_rx_statistics_info_context**

The data structure defining the return storage for a statistics request from `ix_cc_csix_rx_async_get_statistics_info()`. This is used with `ix_e_cc_csix_rx_statistics_info` Enumeration.

C Syntax

```
typedef struct ix_s_cc_csix_rx_statistics_info_context
{
    ix_cc_csix_rx_statistics_info entity;
    ix_uint32 index,
    void *pUserContext;
} ix_cc_csix_rx_statistics_info_context;
```

Input

entity	The type of statistics information. see ix_e_cc_csix_rx_statistics_info Enumeration.
index	The index number to the statistics information. For the CSIX RX, the index number can be any number from 0 to 1023 for VoQ (Virtual Output Queue) 0 to 1023.
pUserContext	The pointer to the calling application-provided context structure which is used by the core component in its callback function for the caller to identify the instance of the request.

5.5.1.1.1 **ix_e_cc_csix_rx_statistics_info Enumeration**

The enumeration defines the types of information supported.

C Syntax

```
typedef enum ix_e_cc_csix_rx_statistics_info
{
    IX_CC_CSIX_RX_PACKETS = 0, /* RX packet count */
    IX_CC_CSIX_RX_BYTES, /* RX byte count */
    IX_CC_CSIX_RX_CFRAMES, /* RX C-Frame count */
    IX_CC_CSIX_RX_DROP_PACKETS, /* Dropped packets */
    IX_CC_CSIX_RX_ALL_COUNTERS, /* All the above counters
    of given VoQ */
} ix_cc_csix_rx_statistics_info;
```

5.5.1.2 ix_cc_csix_rx_cb_get_statistics_info

The function prototype of the callback for `ix_cc_csix_rx_async_get_statistics_info()`.

C Syntax

```
ix_error (*ix_cc_csix_rx_cb_get_statistics_info)(
    ix_error arg_Result,
    void *arg_pContext,
    ix_uint64 *arg_pBuffer);
```

Input

<code>arg_Result</code>	The error code returned by CSIX RX core component as a result of the statistics request.
<code>*arg_pContext</code>	The pointer to the calling application-provided context.
<code>*arg_pBuffer</code>	the pointer to the buffer that stores the obtained statistics information. For per VoQ single 64-bit counter, the buffer length is 8 bytes. For entity of type <code>IX_CC_CSIX_RX_ALL_COUNTERS</code> , the buffer length is $8 * 4 = 32$ bytes.

5.6 Library API

Table 5-3 lists the function calls available in the library API provided by the CSIX RX core component.

Table 5-3. Library API Function Calls of the CSIX RX Core Component

API	Description
<code>ix_cc_csix_rx_get_statistics_info()</code>	Obtains the statistics information.

5.6.1 ix_cc_csix_rx_get_statistics_info()

The function returns the statistics information requested.

C Syntax

```
ix_error ix_cc_csix_rx_get_statistics_info(
    ix_cc_csix_rx_statistics_info arg_entity,
    ix_uint32 arg_index,
    ix_cc_csix_rx_statistics_info_data *arg_pBuffer,
    void *arg_pContext);
```

Input

<code>arg_entity</code>	Type of statistics information. See Section 5.6.1.1 , “ ix_e_cc_csix_rx_statistics_info Enumeration ”.
<code>arg_index</code>	Index number to the statistics information. For the CSIX RX, the index number can be any number from 0—1023 for VoQ (Virtual Output Queue).
<code>arg_pBuffer</code>	Pointer to the ix_cc_statistics_info_data structure.
<code>arg_pContext</code>	The pointer to the control block memory allocated earlier in ix_cc_csix_tx_init() .

Note: It is the calling applications’s responsibility to allocate the required memory, 8 bytes, and use it as the `pDataBuffer` pointer.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. A valid <code>ix_error</code>—the operation failed.
--------------	---

5.6.1.1 [ix_e_cc_csix_rx_statistics_info Enumeration](#)

The enumeration defines the types of statistics information supported.

C Syntax

```
typedef enum ix_e_cc_csix_rx_statistics_info
{
    IX_CC_CSIX_RX_PACKETS = 0, /* Rx packet count */
    IX_CC_CSIX_RX_BYTES, /* Rx byte count */
    IX_CC_CSIX_RX_CFRAMES, /* Rx C-Frame count */
    IX_CC_CSIX_RX_DROP_PACKETS, /* Dropped packets */
    IX_CC_CSIX_RX_ALL_COUNTERS, /* All the above counters of a given VoQ */
} ix_cc_csix_rx_statistics_info;
```

5.6.1.2 ix_cc_statistics_info_data

Pointer to the data structure used in `ix_cc_csix_rx_get_statistics_info()`.

C Syntax

```
typedef struct ix_s_cc_statistics_info_data
{
    ix_uint32 dataLength; /*length in bytes of the data buffer*/
    void *pDataBuffer; /* pointer to a buffer where the statistics data will
                        be stored */
} ix_cc_statistics_info_data;
```

Note: For `IX_CC_CSIX_RX_RX_COUNT` and `IX_CC_CSIX_RX_DROP` attributes, the data length must be 8 bytes for index number from 0 to 1023.

The Ethernet RX core component performs the following functions:

- Configures and initializes the Packet RX microblock
- Receives ARP and other exception packets from the Packet RX microblock
- Sends ARP packets over the PCI Bus to the Ethernet TX core component
- Sends other exception packets to another calling application-defined output core component

For complete details see [Chapter 43, “Ethernet RX Core Component”](#) of the *Intel® Internet Exchange Architecture (IXA) Software Building Blocks Developer’s Manual*.

6.1 Core Component Infrastructure API

[Table 6-1](#) shows the Core Component Infrastructure API provided by the Ethernet RX core component.

Table 6-1. Ethernet RX Core Component Functions

API	Description
<code>ix_cc_eth_rx_init()</code>	Initializes the core component
<code>ix_cc_eth_rx_fini()</code>	Terminates the core component
<code>ix_cc_eth_rx_msg_handler()</code>	Handles messages for interface state and statistics
<code>ix_cc_eth_rx_high_priority_pkt_handler()</code>	High priority packet handler for processing ARP exception packets
<code>ix_cc_eth_rx_low_priority_pkt_handler()</code>	Low priority packet handler for processing non-IP exception packets received from microblock

6.1.1 `ix_cc_eth_rx_init()`

The function initializes the core component. It should be called and returned successfully before any other function in the core component can be called. It performs static configuration for the Ethernet RX microblock by allocating and patching the required variables and memory block into the microblock through calling Resource Manager APIs, `ix_rm_ueng_patch_symbols()`.

The `ix_cc_eth_rx_init()` function registers the message handlers and packet handlers for its defined message inputs and packet input. In addition, it uses the handler registration API to register message handler and packet handlers for the defined message and packet inputs.

C Syntax

```
ix_error ix_cc_eth_rx_init (
    ix_cc_handle arg_CcHandle,
    void **arg_ppContext);
```

Input

<code>arg_CcHandle</code>	The handle to the core component.
---------------------------	-----------------------------------

Input/Output

<code>arg_ppContext</code>	<p>This is an input/output argument.</p> <p>As Input, it is a pointer to a generic init context (<code>ix_cc_init_context</code>) which is used to extract system configuration information.</p> <p>As Output, it is the location that stores the pointer to the control block allocated by the core component. This pointer is later passed into the ix_cc_eth_rx_fini() function by the Core Component Infrastructure for freeing the memory when the core component is terminated.</p>
----------------------------	---

Output/Returns

Return Value	<p>Returns a valid <code>ix_error</code>.</p> <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • A valid <code>ix_error</code>—the operation failed.
--------------	---

6.1.2 `ix_cc_eth_rx_fini()`

The function terminates the core component and is executed when the execution engine running the core component is being shut down. This function frees all allocated memory and resources allocated by `ix_cc_eth_rx_init()` function as well as other core component.functions.

C Syntax

```
ix_error ix_cc_eth_rx_fini (
    ix_cc_handle arg_CcHandle,
    void *arg_pContext);
```

Input

<code>arg_CcHandle</code>	The handle to the core component.
<code>arg_pContext</code>	The pointer to the control block memory allocated earlier in <code>ix_cc_eth_rx_init()</code> . The termination routine uses it to deallocate the control block memory.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. A valid <code>ix_error</code>—the operation failed.
--------------	---

6.1.3 `ix_cc_eth_rx_msg_handler()`

The Ethernet RX core component defines an input for handling messages sent from messaging API. The input id is defined in the system file, `bindings.h` as the symbolic constant, `IX_ETH_RX_MSG_INPUT`.

This function is the message handler associated with the input. For each message received, the function calls an associated library API defined in [Section 6.3, “Library API” on page 107](#) to process the message.

C Syntax

```
ix_error ix_cc_eth_rx_msg_handler(
    ix_buffer_handle arg_Msg,
    ix_uint32 arg_UserData
    void *arg_pContext);
```

`IX_ETH_RX_MSG_INPUT` Symbolic Constant

```
#define IX_ETH_RX_MSG_INPUT.
```

Input

arg_Msg	The buffer handle that has message specific information embedded. The embedded information is the parameter for calling the associated library API function.
arg_UserData	The message type. The following are defined message types: <ul style="list-style-type: none"> IX_CC_ETH_RX_MSG_GET_STATISTICS_INFO—obtains the statistical information IX_CC_ETH_RX_MSG_GET_INTERFACE_STATE—obtains the interface state IX_CC_COMMON_MSG_ID_PROP_UPDATE—sets the interface UP/DOWN IX_CC_ETH_RX_MSG_ADD_MAC_ADDR—add MAC address to SRAM filter table IX_CC_ETH_RX_MSG_DELETE_MAC_ADDR—delete MAC address from SRAM filter table IX_CC_ETH_RX_MSG_LOOKUP_PORT—lookup forwarding port information in SRAM filter table
arg_pContext	The handle to the core component-created internal context structure.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> IX_SUCCESS—the operation succeeded. A valid <code>ix_error</code>—the operation failed.
--------------	--

6.1.4 ix_cc_eth_rx_high_priority_pkt_handler()

This function handles the exception packets sent “up” by the Ethernet RX microblock. The handler function processes ARP exception packets.

When Ethernet RX microblock receives an ARP packet from the network, it sends it to the Ethernet RX core component as an exception packet. The Ethernet RX core component sends the ARP packet to Ethernet TX core component (through PCI bus) on the egress side for the ARP module of Ethernet TX core component to process.

C Syntax

```
ix_error ix_cc_eth_rx_high_priority_pkt_handler (
    ix_buffer_handle arg_hDataBuffer,
    ix_uint32 arg_ExceptionCode,
    void *arg_pContext)
```


Input

<code>arg_hDataBuffer</code>	Buffer handle to the exception packet.
<code>arg_ExceptionCode</code>	ARP packet types.
<code>arg_pContext</code>	Pointer to a context (may be a calling application-defined structure) that is passed to the core component when a packet arrives.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. A valid <code>ix_error</code>—the operation failed.
--------------	---

6.1.5 `ix_cc_eth_rx_low_priority_pkt_handler()`

This function handles the exception packets sent “up” by the Ethernet RX microblock. The handler function processes non-IP exception packets like PPP LCP and IPCP protocol packet. The Ethernet RX core component passes these packets to the other output, that is, `IX_ETH_RX_OTHER_OUTPUT`, for other component to handle. Any application/component that needs to capture these packets can bind to this output in `bindings.h` file. Currently, this output is bound to `IX_CC_PKT_DROP`.

C Syntax

```
ix_error ix_cc_eth_rx_low_priority_pkt_handler (
    ix_buffer_handle arg_hDataBuffer,
    ix_uint32 arg_ExceptionCode,
    void *arg_pContext)
```

Input

<code>arg_hDataBuffer</code>	Buffer handle to the exception packet.
<code>arg_ExceptionCode</code>	Non-IP packet types.
<code>arg_pContext</code>	Pointer to a context (may be a calling application-defined structure) that will be passed to the core component when a packet arrives.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. A valid <code>ix_error</code>—the operation failed.
--------------	---

6.2 Messaging API

Table 6-2 shows the messaging API provided by the Ethernet RX core component.

Table 6-2. Ethernet Interface RX Messaging Functions

API	Description
<code>ix_cc_eth_rx_async_get_statistics_info()</code>	Obtains the statistics info
<code>ix_cc_eth_rx_async_get_interface_state()</code>	Gets the Ethernet interface state
<code>ix_cc_eth_rx_async_add_mac_addr()</code>	Adds MAC address for filtering to SRAM MAC filter table
<code>ix_cc_eth_rx_async_delete_mac_addr()</code>	Deletes MAC address from SRAM MAC filter table
<code>ix_cc_eth_rx_async_lookup_port()</code>	Looks up forwarding port information in SRAM MAC filter table

6.2.1 `ix_cc_eth_rx_async_get_statistics_info()`

This function gets the statistical information from the Ethernet RX core component. It is an asynchronous function and the requested statistics counter is returned through the calling application-provided callback function. The function generates an `IX_CC_ETH_RX_MSG_GET_STATISTICS_INFO` message and passes it to the Ethernet RX core component by calling the send message API of the Message Support Library. The message handler function of the Ethernet TX core component `ix_cc_eth_tx_msg_handler()`, calls the corresponding library API, `ix_cc_eth_tx_get_statistics_info()`, for handling the message.

C Syntax

```
ix_error ix_cc_eth_rx_async_get_statistics_info (
    ix_cc_eth_rx_statistics_info_context *arg_pStatInfoContext,
    ix_cc_eth_rx_cb_get_statistics_info arg_Callback);
```

Input

`arg_pStatInfoContext` The data type as defined in [Section 6.2.1.1](#), “`ix_s_cc_eth_rx_statistics_info_context`.”

`arg_Callback` The pointer to the callback routine [Section 6.2.1.2](#), “`ix_cc_eth_rx_cb_get_statistics_info`.”

Output/Returns

Return Value Returns a valid `ix_error`.

- `IX_SUCCESS`—the operation succeeded.
- A valid `ix_error`—the operation failed.

6.2.1.1 ix_s_cc_eth_rx_statistics_info_context

This data structure specifies the type of statistics information requested, the index to the statistical information type, and the requestor. It is used with [ix_e_cc_eth_rx_statistics_info Enumeration](#), an enumeration listing the supported types of Ethernet RX statistics.

C Syntax

```
typedef struct ix_s_cc_eth_rx_statistics_info_context
{
    ix_cc_eth_rx_statistics_info entity;
    ix_uint32 index,
    void *pUserContext;
} ix_cc_eth_rx_statistics_info_context;
```

Input

entity	The type of statistical information. See ix_e_cc_eth_rx_statistics_info Enumeration .
index	The index number of the statistical information type. For any of the information types listed above, the index number can be any number from 0 to (IX_CC_ETH_RX_NUM_PORTS - 1) representing the Ethernet port 0 to (IX_CC_ETH_RX_NUM_PORTS - 1).
pUserContext	The pointer to the calling application-provided context structure. This is used by the core component in the callback function so that the calling application can identify the instance of the request.

6.2.1.1.1 ix_e_cc_eth_rx_statistics_info Enumeration

The enumerated type that defines the supported types of statistics information for the Ethernet RX core component.

C Syntax

```
typedef enum ix_e_cc_eth_rx_statistics_info
{
    IX_CC_ETH_RX_OK_OCTETS, /* bytes received in legal frames*/
    IX_CC_ETH_RX_BAD_OCTETS, /*bytes received in all bad frames with
                               legal size */
    IX_CC_ETH_RX_UCAST_PACKETS, /*unicast packets received*/
    IX_CC_ETH_RX_MCAST_PACKETS, /*multicast packets received */
    IX_CC_ETH_RX_BCAST_PACKETS, /*broadcast packets received */
    IX_CC_ETH_RX_PACKETS_64, /*packets received that are 64 octets in length*/
    IX_CC_ETH_RX_PACKETS_65_127, /* packets received that are 65-127 octets
                                   in length */
    IX_CC_ETH_RX_PACKETS_128_255, /*packets received that are 128-255 octets
                                   in length */
    IX_CC_ETH_RX_PACKETS_256_511, /* packets received that are 256-511 octets
                                   in length */
    IX_CC_ETH_RX_PACKETS_512_1023, /* packets received that are 512-1023 octets
                                   in length */
    IX_CC_ETH_RX_PACKETS_1024_1518, /*packets received that are 1024-1518
                                   octets in length */
    IX_CC_ETH_RX_PACKETS_1519_MAX, /*packets received that are >1518 octets
                                   in length */
}
```

```

IX_CC_ETH_RX_FCS_ERR, /* frames received with legal size but with wrong
                        CRC field */
IX_CC_ETH_RX_VLAN_TAG, /*OK frames received with VLAN tag */
IX_CC_ETH_RX_DATA_ERR, /*frames received with legal length with
                        code violation */
IX_CC_ETH_RX_ALIGN_ERR, /*frames received with legal frame size but
                        containing less than 8 additional bits */
IX_CC_ETH_RX_LONG_ERR, /*frames received bigger than maximum allowed but
                        with ok CRC and integral number of octets */
IX_CC_ETH_RX_JABBER_ERR, /*frames received bigger than maximum allowed with
                        either bad CRC or a non-integral number of octets */
IX_CC_ETH_RX_PAUSE_MAC_CTL, /*number of Pause MAC control frames received*/
IX_CC_ETH_RX_UNKNOWN_CTL_FRAME, /*number of MAC control frames received
                        with an op code different from 0001 (Pause) */
IX_CC_ETH_RX_VLONG_ERR, /*frames received bigger than the larger of 2*max
                        frame size and 50000 bits */
IX_CC_ETH_RX_GET_RUNT_ERR, /*packets received less than 64 octets in length
                        but longer than or equal to 96 bit times */
IX_CC_ETH_RX_GET_SHORT_ERR, /*packets received that are less than 96-bit
                        times */
IX_CC_ETH_RX_GET_SEQ_ERR, /* number of sequencing errors that occur in Fiber
                        mode */
IX_CC_ETH_RX_GET_SYMBOL_ERR, /* number of symbol errors, encountered by
                        the PHY */
IX_CC_ETH_RX_ALL_DRIVER_COUNTERS, /* all above driver related counters of a
                        given port */
IX_CC_ETH_RX_COUNTER_LAST
} ix_cc_eth_rx_statistics_info;

```

6.2.1.2 ix_cc_eth_rx_cb_get_statistics_info

The function prototype of callback functions for calls to
[ix_cc_eth_rx_async_get_statistics_info\(\)](#).

C Syntax

```

ix_error (*ix_cc_eth_rx_cb_get_statistics_info)(
    ix_error arg_Result,
    void *arg_pContext,
    ix_uint64 *arg_pBuffer,
    ix_unit32_arg_MsgLen);

```

Input

arg_Result	The error code returned by the Ethernet RX core component for the result of the statistics request
arg_pContext	The pointer to the calling application-provided context
arg_pBuffer	The pointer to the buffer that stores the obtained statistical information
arg_MsgLen	The length, that is, number of bytes information. It specifies the length of <code>arg_pBuffer</code> , as it may be 8 bytes for single 64 bit counter requested or (8* total no. of counters) bytes for <code>ALL_COUNTERS</code> entity type.

6.2.2 ix_cc_eth_rx_async_get_interface_state()

This function gets the Ethernet interface port state information. It is an asynchronous function and the current state of the interface is returned through the calling application-provided callback function. The function may be used by an application to obtain the interface state information. The function generates a `IX_CC_ETH_RX_MSG_GET_INTERFACE_STATE` message and passes it to the Ethernet RX core component through calling the send message API—see [Chapter 26, “Message Helper and Support Library.”](#) The message handler function of the Ethernet RX core component—`ix_cc_eth_rx_msg_handler()`, calls the corresponding library API, `ix_cc_eth_rx_get_interface_state()`—for handling the message.

C Syntax

```
ix_error ix_cc_eth_rx_async_get_interface_state (
    ix_cc_eth_rx_if_state_context *arg_pStateContext,
    ix_cc_eth_rx_cb_get_interface_state arg_Callback);
```

Input

<code>arg_pStateContext</code>	The data structure defined in Section 6.2.2.1 , “ <code>ix_s_cc_eth_rx_if_state_context</code> .”
<code>arg_Callback</code>	Specifies the pointer to the callback routine. See Section 6.2.2.2 , “ <code>ix_cc_eth_rx_cb_get_interface_state</code> .”

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • A valid <code>ix_error</code>—the operation failed.
--------------	---

6.2.2.1 ix_s_cc_eth_rx_if_state_context

This data structure specifies the Ethernet interface port state information.

C Syntax

```
typedef struct ix_s_cc_eth_rx_if_state_context
{
    ix_uint8 port;
    void *pUserContext;
} ix_cc_eth_rx_if_state_context;
```

Input

<code>port</code>	The interface port number whose interface state is to be returned.
<code>pUserContext</code>	The pointer to a calling application-provided context structure. This is used by the core component in the callback functions to identify the target of the request to the calling application

6.2.2.2 **`ix_cc_eth_rx_cb_get_interface_state`**

The function prototype of the callback for call to `ix_cc_eth_rx_async_get_interface_state()`.

C Syntax

```
ix_error (*ix_cc_eth_rx_cb_get_interface_state)(
    ix_error arg_Result,
    void *arg_pContext,
    ix_cc_eth_rx_if_state *arg_pState);
```

Input

<code>arg_Result</code>	The error code returned by the Ethernet RX core component as a result of the interface state access request.
<code>arg_pContext</code>	The pointer to the calling application-provided context.
<code>arg_pState</code>	The pointer for storing the obtained interface state. See ix_e_cc_eth_rx_if_state Enumeration .

`ix_e_cc_eth_rx_if_state` Enumeration

The Ethernet RX state data type is defined as follows:

C Syntax

```
typedef enum ix_e_cc_eth_rx_if_state
{
    IX_CC_ETH_RX_IF_STATE_DOWN = 0,
    IX_CC_ETH_RX_IF_STATE_UP
} ix_cc_eth_rx_if_state;
```

6.2.3 **`ix_cc_eth_rx_async_add_mac_addr()`**

This function implements the messaging API for adding MAC address for filtering to MAC filter table. It is an asynchronous function and the result of the operation is returned through calling application-provided callback function. The function generates a `IX_CC_ETH_RX_MSG_ADD_MAC_ADDR` message and passes it to Ethernet RX core component through calling the send message API of Message support Library. The message handler function of Ethernet RX core component—`ix_cc_eth_rx_msg_handler()`, invokes the corresponding library API—`ix_cc_eth_rx_add_mac_addr()`, for handling the message.

C Syntax

```
ix_error ix_cc_eth_rx_async_add_mac_addr(
    ix_uint8 *arg_destMac,
    ix_uint32 arg_portNum,
    ix_cc_eth_rx_cb_mac_addr_op arg_Callback,
    void *arg_pUserContext);
```

Input

<code>arg_destMac</code>	A 6-bytes MAC address to be added in MAC filter table
<code>arg_portNum</code>	An outgoing port information to be added in MAC filter table
<code>arg_pUserContext</code>	A pointer to the calling application-provided context structure which is used by the core component in the callback function for the calling application to identify the instance of the request.
<code>arg_Callback</code>	the pointer to the callback routine. See ix_cc_eth_rx_cb_mac_addr_op

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—returned if the operation succeeds. A valid <code>ix_error</code>—returned if the operation fails.
--------------	---

6.2.3.1 [ix_cc_eth_rx_cb_mac_addr_op](#)

The function prototype for message-handler callback functions provided by the calling application.

C Syntax

```
typedef ix_error (*ix_cc_eth_rx_cb_mac_addr_op)(
    ix_error arg_Result,
    void *arg_pContext);
```

Input

<code>arg_Result</code>	An error code returned as a result of the add request: <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—returned if the operation succeeds. A valid <code>ix_error</code>—returned if the operation fails
<code>arg_pContext</code>	Pointer to a calling application-provided context.

6.2.4 `ix_cc_eth_rx_async_delete_mac_addr()`

This messaging API deletes MAC address from the MAC filter table. It is an asynchronous function and the result of the operation is returned through the calling application-provided callback function. The function generates a `IX_CC_ETH_RX_MSG_DELETE_MAC_ADDR` message and passes it to Ethernet RX core component through calling the send message API of Message support Library. The message handler function of Ethernet RX core component—`ix_cc_eth_rx_msg_handler()`, invokes the corresponding library API, that is, `ix_cc_eth_rx_del_mac_addr()`, for handling the message.

C Syntax

```
ix_error ix_cc_eth_rx_async_delete_mac_addr (
    ix_uint8 *arg_destMac,
    ix_cc_eth_rx_cb_mac_addr_op arg_Callback,
    void *arg_pUserContext);
```

Input

<code>arg_destMac</code>	A 6-bytes of MAC address to be deleted from MAC filter table
<code>arg_pUserContext</code>	A pointer to the calling application-provided context structure which is used by the core component in the callback function for the calling application to identify the instance of the request.
<code>arg_Callback</code>	A pointer to the callback routine. See ix_cc_eth_rx_cb_mac_addr_op .

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—returned if the operation succeeds. A valid <code>ix_error</code>—returned if the operation fails.
--------------	---

6.2.4.1 `ix_cc_eth_rx_cb_mac_addr_op`

The function prototype for message-handler callback functions provided by the calling application.

C Syntax

```
typedef ix_error (*ix_cc_eth_rx_cb_mac_addr_op)(
    ix_error arg_Result,
    void *arg_pContext);
```

Input

<code>arg_Result</code>	An error code returned as a result of the add request: <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—returned if the operation succeeds. A valid <code>ix_error</code>—returned if the operation fails
<code>arg_pContext</code>	Pointer to a calling application-provided context.

6.2.5 `ix_cc_eth_rx_async_lookup_port()`

This messaging API looks up port information from MAC filter table. It is an asynchronous function and the result of the operation is returned through the calling application-provided callback function. The function generates a `IX_CC_ETH_RX_MSG_LOOKUP_PORT` message and passes it to Ethernet RX core component through calling the send message API of Message support Library. The message handler function of Ethernet RX core component—`ix_cc_eth_rx_msg_handler()`, invokes the corresponding library API—`ix_cc_eth_rx_lookup_port()`, for handling the message.

C Syntax

```
ix_error ix_cc_eth_rx_async_lookup_port (
    ix_uint8 *arg_destMac,
    ix_cc_eth_rx_cb_lookup_port arg_Callback,
    void *arg_pUserContext);
```

Input

<code>arg_destMac</code>	Six bytes of MAC address to be looked up in MAC filter table
<code>arg_pUserContext</code>	A pointer to the calling application-provided context structure which is used by the core component in callback function for the calling application to identify the instance of the request.
<code>arg_Callback</code>	A pointer to the callback routine.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—returned if the operation succeeds. A valid <code>ix_error</code>—returned if the operation fails.
--------------	---

6.2.5.1 ix_cc_eth_rx_cb_lookup_port

The function prototype for message-handler callback functions provided by the calling application.

C Syntax

```
typedef ix_error (*ix_cc_eth_rx_cb_lookup_port)(
    ix_error arg_Result,
    void *arg_pContext,
    ix_uint32 *arg_pPortNum);
```

Input

arg_Result	An error code returned as a result of the add request: <ul style="list-style-type: none"> IX_SUCCESS—returned if the operation succeeds. A valid ix_error—returned if the operation fails
arg_pContext	Pointer to a calling application-provided context.

6.3 Library API

Table 6-3 shows the Library API provided by the Ethernet RX core component.

Table 6-3. Ethernet Interface RX Library Functions

API	Description
<code>ix_cc_eth_rx_get_interface_state()</code>	Gets the Ethernet interface state
<code>ix_cc_eth_rx_set_property()</code>	Sets or changes a dynamic property
<code>ix_cc_eth_rx_add_mac_addr()</code>	Adds MAC address to MAC filter table
<code>ix_cc_eth_rx_del_mac_addr()</code>	Deletes MAC address from MAC filter table
<code>ix_cc_eth_rx_lookup_port()</code>	Looks up port information in MAC filter table

6.3.1 ix_cc_eth_rx_get_interface_state()

The function gets the current state of the Ethernet RX interface.

C Syntax

```
ix_error ix_cc_eth_rx_get_interface_state (
    ix_uint32 arg_PortId,
    ix_cc_eth_rx_if_state *arg_pState,
    void *arg_pContext);
```

Input

<code>arg_PortId</code>	The interface port identifier whose state is to be returned.
<code>arg_pContext</code>	The pointer to the component control block memory allocated earlier in its init function.

Output/Returns

<code>arg_pState</code>	The pointer where the current state of the interface is stored when the function returns. The interface state is define in the Section 6.3.1.1 , “ enum ix_e_cc_eth_rx_if_state Enumeration .”
Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none">• <code>IX_SUCCESS</code>—the operation succeeded.• A valid <code>ix_error</code>—the operation failed.

6.3.1.1 `enum ix_e_cc_eth_rx_if_state` Enumeration

C Syntax

```
typedef enum ix_e_cc_eth_rx_if_state
{
    IX_CC_ETH_RX_IF_STATE_DOWN = 0;
    IX_CC_ETH_RX_IF_STATE_UP;
} ix_cc_eth_rx_if_state;
```

6.3.2 `ix_cc_eth_rx_set_property()`

The function implements the generic library API for setting dynamic property provided by Ethernet RX core component.

C Syntax

```
ix_error ix_cc_eth_rx_set_property (
    ix_uint32 arg_PropId,
    ix_cc_properties *arg_pProperty,
    void *arg_pContext);
```

Input

<code>arg_PropId</code>	property to be changed. It is <code>IX_CC_SET_PROPERTY_PHYSICAL_IF_STATUS</code> for Ethernet RX core component.
<code>arg_pProperty</code>	The pointer to the property structure.
<code>arg_pContext</code>	The pointer to the component control block memory allocated earlier in its init function.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. A valid <code>ix_error</code>—the operation failed.
--------------	---

6.3.3 `ix_cc_eth_rx_add_mac_addr()`

This is a generic library API for adding a MAC address into the MAC filtering table maintained by Ethernet RX core component. When a unit cast mode is enabled in the Ethernet RX microblock, the MAC filtering table is looked-up by the microblock. For MAC frames whose destination MAC address match with one of the MAC addresses stored in the table, the frame is forwarded by the Ethernet RX microblock to the next microblock. When the lookup fails to find a matching address from the table, the MAC frame is dropped by the microblock.

C Syntax

```
ix_error ix_cc_eth_rx_add_mac_addr (
    ix_uint8 *arg_pMacAddr,
    ix_uint32 arg_OutputPortId,
    void *arg_pContext);
```

Input

<code>arg_pMacAddr</code>	A pointer to a 6-byte array that contains the MAC address in network byte order.
<code>arg_OutputPortId</code>	An output port ID associated with the MAC address.
<code>arg_pContext</code>	A pointer to the component control block memory allocated earlier in its init function.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. A valid <code>ix_error</code>—the operation failed.
--------------	---

6.3.4 `ix_cc_eth_rx_del_mac_addr()`

This is a generic library API for deleting a MAC address from the MAC filtering table maintained by Ethernet RX core component.

C Syntax

```
ix_error ix_cc_eth_rx_del_mac_addr(
    ix_uint8 *arg_pMacAddr,
    void *arg_pContext);
```

Input

<code>arg_pMacAddr</code>	A pointer to a 6-byte array that contains the MAC address in network byte order to be removed from the table.
<code>arg_pContext</code>	A pointer to the component control block memory allocated earlier in its init function.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. A valid <code>ix_error</code>—the operation failed.
--------------	---

6.3.5 `ix_cc_eth_rx_lookup_port()`

This is a generic library API for getting output port id of a MAC address from the MAC filtering table maintained by Ethernet RX core component. The output port id is valid only if the return value is `IX_SUCCESS`.

C Syntax

```
ix_error ix_cc_eth_rx_lookup_port(
    ix_uint8 *arg_pMacAddr,
    ix_uint32 *arg_pOutputPortId,
    void *arg_pContext);
```

Input

<code>arg_pMacAddr</code>	A pointer to a 6-byte array that contains the MAC address in network byte order.
<code>arg_pContext</code>	A pointer to the component control block memory allocated earlier in its init function.

Output/Returns

<code>arg_pOutputPortId</code>	A pointer to an output port ID associated with the MAC address.
Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. A valid <code>ix_error</code>—the operation failed.

Transmit

The following chapters are included in this section:

- [Chapter 7, “CSIX TX”](#)

CSIX TX core component corresponds to CSIX TX microblock and performs the initialization and configuration for the CSIX TX microblock through patching of the imported symbols and memory block into the microblock.

- [Chapter 8, “ATM/POS TX”](#)

POS TX core component performs the initialization and configuration for POS Tx microblock through patching imported symbols and memory blocks into the microblocks.

- [Chapter 9, “Ethernet TX”](#)

Ethernet TX core component performs the following functions:

- Initialization and Configuration of Ethernet Tx microblock
- Configuration of multi-port Gigabit Ethernet media card
- Handling exception packets for ARP processing and resolution of layer2 address

- [Chapter 10, “Ethernet ARP Module”](#)

Ethernet ARP module - This module implements the basic ARP protocol outlined in RFC826 and additional mandatory rules specified in RFC1122:

- ARP cache
- Handling of ARP packet generation and reception
- ARP cache time-out
- Prevention of ARP flooding
- ARP packet queue
- Generation and handling of gratuitous ARP request.
- Detection of IP address duplication

The CSIX TX core component performs the following functions:

- Initializes and configures of the CSIX TX microblock
- Provides an interface to a system application for retrieving statistical parameters

For complete details, see [Chapter 44, “CSIX TX Core Component”](#) of the *Intel® Internet Exchange Architecture (IXA) Software Building Blocks Developer’s Manual*.

7.1 Core Component Infrastructure API

Table 7-1 summarizes the CSIX TX core component Infrastructure API.

Table 7-1. CSIX TX Core Component Infrastructure API

API	Description
<code>ix_cc_csix_tx_init()</code>	Initializes the core component.
<code>ix_cc_csix_tx_fini()</code>	Terminates the core component.
<code>ix_cc_csix_tx_msg_handler()</code>	Message handler for processing interface state and statistics.

7.1.1 `ix_cc_csix_tx_init()`

Initializes the CSIX TX core component. It must be called and returned successfully before any other function in the core component can be called. It performs static configuration for the microblock by allocating and patching required variables and a memory block into the microblock using the Resource Manager API. On return from this operation the interface of the CSIX TX core component is enabled.

C Syntax

```
ix_error ix_cc_csix_tx_init(  
    ix_cc_handle arg_CcHandle,  
    void **arg_ppContext);
```

Input

`arg_CcHandle` A handle to the core component.

Input/Output

<code>arg_ppContext</code>	<p>This is an INPUT and OUTPUT argument.</p> <ul style="list-style-type: none"> As an input, this is a pointer to a generic initialization context which is used to extract system configuration information. As an output, it is a pointer with the location of the pointer to the control block allocated by the core component. The control block is internal to the core component and contains transmit context memory and other variables and internal data structures. This pointer is later used by the <code>ix_cc_csix_tx_fini()</code> function to free memory when this core component is destroyed.
----------------------------	--

Output/Returns

Return Value	<p>Returns a valid <code>ix_error</code>.</p> <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. A valid <code>ix_error</code>—the operation failed.
--------------	---

7.1.2 `ix_cc_csix_tx_fini()`

This function terminates the CSIX TX core component. It is called when the execution engine running the core component is shut down. This function frees all allocated memory and resources obtained by the `ix_cc_csix_tx_init()` function.

C Syntax

```
ix_error ix_cc_csix_tx_fini(
    ix_cc_handle arg_CcHandle,
    void *arg_pContext);
```

Input

<code>arg_CcHandle</code>	A handle to the CSIX TX core component.
<code>arg_pContext</code>	A pointer to the control block memory allocated earlier in <code>ix_cc_csix_tx_init()</code> . This function uses the pointer to de-allocate the control block memory.

Output/Returns

Return Value	<p>Returns a valid <code>ix_error</code>.</p> <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. A valid <code>ix_error</code>—the operation failed.
--------------	---

7.1.3 `ix_cc_csix_tx_msg_handler()`

This is the message handler function for the CSIX TX core component.

C Syntax

```
ix_error ix_cc_csix_tx_msg_handler(
    ix_buffer_handle arg_Msg,
    ix_uint32 arg_UserData
    void *arg_pContext);
```

Input

<code>arg_Msg</code>	A buffer handle which has message-specific information embedded. Typically, the embedded information consists of the parameters for the invocation of associated library API functions.
<code>arg_UserData</code>	The message type. The following are defined message types: <ul style="list-style-type: none"> <code>IX_CC_CSIX_TX_MSG_GET_STATISTICS_INFO</code>—return statistics information.
<code>arg_pContext</code>	A handle to the internal control block structure created by the core component's initialization function, <code>ix_cc_csix_tx_init()</code> .

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. A valid <code>ix_error</code>—the operation failed.
--------------	---

7.2 Messaging API

Table 7-2 summarizes the messaging API provided by the CSIX TX core component.

Table 7-2. Messaging API for the CSIX TX Core Component

API	Description
ix_cc_csix_tx_async_get_statistics_info()	Returns statistics information.

7.2.1 [ix_cc_csix_tx_async_get_statistics_info\(\)](#)

The function returns statistics information. This is an asynchronous function and the state of the interface is returned through a calling application-provided callback function.

C Syntax

```
ix_error ix_cc_csix_tx_async_get_statistics_info(
    ix_cc_csix_tx_statistics_info_context *arg_pMgInfoContext,
    ix_cc_csix_tx_cb_get_statistics_info arg_Callback);
```

Input

<code>arg_pMgInfoContext</code>	A pointer to a data structure. See ix_s_cc_csix_tx_statistics_info_context .
<code>arg_Callback</code>	This argument specifies the pointer to the callback routine. See ix_cc_csix_tx_cb_get_statistics_info . The buffer size is: <ul style="list-style-type: none"> For entity type <code>IX_CC_CSIX_TX_ALL_COUNTERS</code>, the buffer length is eight times four or 32 bytes. For all other entity types—requiring a per-VOQ single 64-bit counter—the buffer length is 8 bytes.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. A valid <code>ix_error</code>—the operation failed.
--------------	---

7.2.1.1 **ix_s_cc_csix_tx_statistics_info_context**

A data structure that defines the type of statistics information desired. This is used with [ix_e_cc_csix_tx_statistics_info Enumeration](#).

C Syntax

```
typedef struct ix_s_cc_csix_tx_statistics_info_context {
    ix_cc_csix_tx_statistics_info entity;
    ix_uint32 index,
    void *pUserContext;
} ix_cc_csix_tx_statistics_info_context;
```

Input

entity	Specifies the type of statistics information. See ix_e_cc_csix_tx_statistics_info Enumeration for a list of valid types of statistics information.
index	Specifies the index number for the statistics information. For the CSIX TX core component, the index number can be any number from zero to 1023 corresponding to a virtual output queue (VOQ).
pUserContext	The pointer to a calling application-provided context structure used by the CSIX TX core component in the callback function to identify the originator of the request.

7.2.1.2 **ix_e_cc_csix_tx_statistics_info Enumeration**

An enumerated type specifying the supported kinds of CSIX TX statistics information.

C Syntax

```
typedef enum ix_e_cc_csix_tx_statistics_info {
    IX_CC_CSIX_TX_PACKETS = 0,
    IX_CC_CSIX_TX_BYTES,
    IX_CC_CSIX_TX_CFRAMES;
    IX_CC_CSIX_TX_RESERVED;
    IX_CC_CSIX_TX_ALL_COUNTERS;
} ix_cc_csix_tx_statistics_info;
```

Values

IX_CC_CSIX_TX_PACKETS	The per-VoQ TX packet count.
IX_CC_CSIX_TX_BYTES	The per-VoQ TX byte count.

Values (Continued)

IX_CC_CSIX_TX_CFRAMES	The per-VoQ TX C-Frame count.
IX_CC_CSIX_TX_RESERVED	The per-VoQ reserved counter.
IX_CC_CSIX_TX_ALL_COUNTERS	All supported counters of a given VoQ.

7.2.1.3 ix_cc_csix_tx_cb_get_statistics_info

This is the prototype of callback functions for calls to `ix_cc_csix_tx_async_get_statistics_info()`.

C Syntax

```
ix_error (*ix_cc_csix_tx_cb_get_statistics_info)(
    ix_error arg_Result,
    void *arg_pContext,
    ix_uint32 *arg_pBuffer);
```

Input

arg_Result	The error code returned by the CSIX TX core component for the result of the statistics request.
arg_pContext	The pointer to the user provided context.
arg_pBuffer	The pointer to the buffer that stores the obtained statistics information.

7.3 Library API

Table 7-3 summarizes the CSIX TX core component Library API.

Table 7-3. CSIX Library API

API	Description
<code>ix_cc_csix_tx_get_statistics_info()</code>	Returns statistics information.

7.3.1 ix_cc_csix_tx_get_statistics_info()

Returns the requested statistics information.

C Syntax

```
ix_error ix_cc_csix_tx_get_statistics_info(
    ix_cc_csix_tx_statistics_info arg_entity,
    ix_uint32 arg_index,
    ix_cc_csix_tx_statistics_info_data *arg_pBuffer,
```



```
void *arg_pContext);
```

Input

<code>arg_entity</code>	Type of statistics information. See also the messaging API, ix_e_cc_csix_tx_statistics_info Enumeration for the type of statistics information supported.
<code>arg_index</code>	Index number of the statistics information. The index number can be any number from zero to 1023 selecting the corresponding Virtual Output Queue (VOQ).
<code>arg_pBuffer</code>	A pointer to an <code>ix_cc_statistics_info_data</code> structure in which to return the requested statistics. See ix_s_cc_statistics_info_data . For 64-bit counters, the data length is eight bytes. It is the calling application's responsibility to allocate the required memory, eight bytes, and assign it to the <code>pDataBuffer</code> pointer.
<code>arg_pContext</code>	A pointer to the control block memory allocated earlier in the call to ix_cc_csix_tx_init() .

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • A valid <code>ix_error</code>—the operation failed.
--------------	---

7.3.1.1 [ix_s_cc_statistics_info_data](#)

The data structure defining the return storage for a statistics request from [ix_cc_csix_tx_get_statistics_info\(\)](#).

C Syntax

```
typedef struct ix_s_cc_statistics_info_data {
    ix_uint32 dataLength; /* length in bytes of the data buffer */
    void *pDataBuffer;    /* pointer to a buffer where */
                        /* the statistics data will be stored */
} ix_cc_statistics_info_data;
```

Input

<code>dataLength</code>	The length in bytes of the data buffer, <code>pDataBuffer</code> .
<code>pDataBuffer</code>	A pointer to a buffer that specifies where to store requested statistics data.

The ATM/POS core component combines the configurations of ATM TX and POS TX.

The ATM/POS core component performs the following functions:

- Initializes and configures the ATM TX microblock and the Packet TX microblock through patching symbols
- Provides interfaces for setting and retrieving the ATM/POS interface state and statistical parameters
- Initializes and configures the ATM/POS framer device

For complete details see [Chapter 45, “ATM/POS TX Core Component”](#) of the *Intel® Internet Exchange Architecture Software Building Blocks Developer’s Manual*.

Note: The support for the ATM TX part of the SoftSAR core component is described in [Chapter 28, “SoftSAR”](#).

8.1 Core Component Infrastructure API

Table 8-1 lists the functions in the Core Component Infrastructure API.

Table 8-1. ATM/POS Interface TX Core Component Infrastructure Functions

API	Description
<code>ix_cc_atm_pos_tx_init()</code>	Initialize the core component
<code>ix_cc_atm_pos_tx_fini()</code>	Terminate the core component
<code>ix_cc_atm_pos_tx_msg_handler()</code>	Message handler for supporting message helper API
<code>ix_cc_atm_pos_tx_property_msg_handler()</code>	Message handler for processing dynamic property update messages

8.1.1 `ix_cc_atm_pos_tx_init()`

The function initializes the core component. It should be called and returned before any other function in the core component is called. Based on the channel mode specified in the static data, the function performs static configuration for either the ATM TX microblock or POS TX microblock (or both) by allocating and patching the required variables and memory blocks into the microblock, by calling the Resource Manager function—`ix_rm_ueng_patch_symbols()`.

This function also registers message handlers for its defined message inputs. Finally, it initializes and configures the ATM/POS framer device by calling the framer device driver based on the static configuration data obtained.

C Syntax

```
ix_error ix_cc_atm_pos_tx_init(
    ix_cc_handle arg_CcHandle,
    void **arg_ppContext);
```

Input

`arg_CcHandle` The handle to the core component.

Output/Returns

`arg_ppContext` The location where the pointer to the control block allocated by the core component is stored. The control block is internal to the core component and contains Transmit Context memory and other variables and internal data structures. This pointer is passed into the `ix_cc_atm_pos_tx_fini()` function by the core component infrastructure to free memory when the core component is terminated.

Return Value Returns a valid `ix_error`.

- `IX_SUCCESS`—the operation succeeded.
- A valid `ix_error`—the operation failed.

8.1.2 `ix_cc_atm_pos_tx_fini()`

The function terminates the core component. It is executed when the execution engine running the core component is shut down. This function frees all the memory and resources allocated by the `ix_cc_atm_pos_tx_init()` function as well as other functions of the core component.

C Syntax

```
ix_error ix_cc_atm_pos_tx_fini (
    ix_cc_handle arg_CcHandle,
    void *arg_pContext);
```

Input

`arg_cc_handle` The handle to the core component.

`arg_pContext` The pointer to the control block memory allocated earlier in `cc_atm-tx_init()`. The termination routine uses it to deallocate the control block memory.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. A valid <code>ix_error</code>—the operation failed.
--------------	---

8.1.3 `ix_cc_atm_pos_tx_msg_handler()`

The ATM/POS TX core component defines an input for handling messages sent from the Messaging API. The input ID is defined in the system file, `bindings.h`, as `#define IX_CC_ATM_POS_TX_MSG_INPUT`.

This function is the message handler associated with the input. For each message received, the function calls an associated library API to process the message.

C Syntax

```
ix_error ix_cc_atm_pos_tx_msg_handler(
    ix_buffer_handle arg_Msg,
    ix_uint32 arg_UserData,
    void *arg_pContext);
```

Input

<code>arg_Msg</code>	The buffer handle which has message specific information embedded. The embedded information usually includes the parameters for calling the associated Library API functions.
<code>arg_UserData</code>	The message type. The following message types are supported: <ul style="list-style-type: none"> <code>IX_CC_ATM_POS_TX_MSG_GET_STATISTICS_INFO</code>—obtain the statistics information of either the ATM interface or the POS interface. <code>IX_CC_ATM_POS_TX_MSG_GET_INTERFACE_STATE</code>—obtain the interface state of an ATM or POS interface port.
<code>arg_pContext</code>	The handle to the core component created internal control block structure created by the core component's init function.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. A valid <code>ix_error</code>—the operation failed.
--------------	---

8.1.4 ix_cc_atm_pos_tx_property_msg_handler()

ATM/POS TX core component defines an input for handling the property update messages sent from the property master. The input id is defined in the system file, `bindings.h` as

```
#define IX_CC_ATM_POS_TX_PROPERTY_MSG_INPUT.
```

This function is the message handler associated with the input. The function calls an internal API to process each update message received.

C Syntax

```
ix_error ix_cc_atm_pos_tx_property_msg_handler (
    ix_buffer_handle arg_Msg,
    ix_uint32 arg_UserData
    void *arg_pContext);
```

Input

<code>arg_Msg</code>	The buffer handle that has message specific information embedded. The buffer embeds the <code>ix_cc_properties</code> structure, which has the required property set in one of its structure fields including the port id.
<code>arg_UserData</code>	The message type. The following is the defined update message type: <ul style="list-style-type: none"> <code>IX_CC_COMMON_MSG_ID_PROP_UPDATE</code>—interface state up/down
<code>arg_pContext</code>	The handle to the core component created internal control block structure created by the core component's init function.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. A valid <code>ix_error</code>—the operation failed.
--------------	---

8.2 Messaging API

Table 8-2 shows the Messaging API provided by the ATM/POS TX core component.

Table 8-2. ATM/POS Interface TX Messaging Functions

API	Description
<code>ix_cc_atm_pos_tx_async_get_statistics_info()</code>	Obtains the statistics information
<code>ix_cc_atm_pos_tx_async_get_interface_state()</code>	Gets the ATM/POS interface state

8.2.1 `ix_cc_atm_pos_tx_async_get_statistics_info()`

This function implements the messaging API for getting the statistics counter information from the ATM/POS TX core component. It is an asynchronous function and the requested statistics counter is returned through the calling application-provided callback function. The ATM/POS core component maintains various statistics counters. These counters are defined in the `ix_cc_atm_pos_tx_statistics_info` data structure.

This function generates an `IX_CC_ATM_POS_TX_MSG_GET_STATISTICS_INFO` message and passes it to ATM/POS TX core component through calling the send message API of the Message Support Library. The message handler function of ATM/POS Tx core component—`ix_cc_atm_pos_tx_msg_handler()`—invokes the corresponding Library API—`ix_cc_atm_pos_tx_get_statistics_info()`—for handling the message.

C Syntax

```
ix_error ix_cc_atm_pos_tx_async_get_statistics_info (
    ix_cc_atm_pos_tx_statistics_info_context *arg_pMgInfoContext,
    ix_cc_atm_pos_tx_cb_get_statistics_info arg_Callback);
```

Input

<code>arg_pMgInfoContext</code>	A pointer to a data structure of type <code>ix_s_cc_atm_pos_tx_statistics_info_context</code> specifying the type of statistics information requested, an index to a virtual circuit, and a pointer identifying the caller.
<code>arg_Callback</code>	The pointer to the <code>ix_cc_atm_pos_tx_cb_get_statistics_info</code> callback routine.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. A valid <code>ix_error</code>—the operation failed.
--------------	---

8.2.1.1 `ix_s_cc_atm_pos_tx_statistics_info_context`

This data structure specifies the type of statistics information requested, the index to a virtual circuit, and the requestor for the function call `ix_cc_atm_pos_tx_async_get_statistics_info()`

C Syntax

```
typedef struct ix_s_cc_atm_pos_tx_statistics_info_context
{
    ix_cc_atm_pos_tx_statistics_info entity;
    ix_uint32 index;
    void *pUserContext;
} ix_cc_atm_pos_tx_statistics_info_context;
```

Input

entity	Type of statistics information. Refer to ix_e_cc_atm_pos_tx_statistics_info Enumeration for the types of information supported. All ATM statistics counters are valid only when the ATM mode is selected. Similarly, all POS statistics counters are valid only when the POS mode is selected.
Index	The index number of the statistics information. For per-VCQ ATM statistics counters, the index number can be any number from 0 to 65535 representing ATM VCQ 0 to 65535. For per-port ATM statistics type, the index number is the ATM port number (total 2048 ports reserved for ATM [BLDBLK]). For any POS statistics type listed above, the index number represents the POS port number (For now total of 16 ports reserved for POS [BLDBLK]).
pUserContext	Pointer to the calling application-provided context structure. This is used by the core component in the callback function so that the calling application can identify the instance of the request.

8.2.1.2 [ix_e_cc_atm_pos_tx_statistics_info Enumeration](#)

The following types of statistics information are supported:

C Syntax

```
typedef enum ix_e_cc_atm_pos_tx_statistics_info
{
    IX_CC_ATM_TX_PACKETS_VCQ, /* per VCQ 64-bit ATM Tx packet count */
    IX_CC_ATM_TX_CELLS_VCQ, /* per VCQ 64-bit ATM Tx cell count */
    IX_CC_ATM_TX_ALL_COUNTERS_VCQ, /* all supported per-VcQ ATM counters
                                   of a VCQ */
    IX_CC_ATM_TX_PACKETS_PORT, /* per port 64-bit ATM Tx packet count */
    IX_CC_ATM_TX_CELLS_PORT, /* per port 64-bit ATM Tx cell count */
    IX_CC_ATM_TX_ALL_COUNTERS_PORT, /* all supported per-port ATM counters
                                   of a port */
    IX_CC_POS_TX_PACKETS, /* per port 64-bit POS Tx packet count */
    IX_CC_POS_TX_DROP_PACKETS, /* per port 64-bit POS drop packet count */
    IX_CC_POS_TX_BYTES, /* per port 64-bit POS Tx byte count */
    IX_CC_POS_TX_DROP_BYTES, /* per port 64-bit POS Tx drop byte count */
    IX_CC_POS_TX_ALL_COUNTERS /* all supported per-port POS counters of
                              a port */
} ix_cc_atm_pos_tx_statistics_info;
```

[ix_e_cc_atm_pos_tx_statistics_info Types](#)

IX_CC_ATM_TX_PACKETS_VCQ	per VCQ 64-bit ATM Tx packet count
IX_CC_ATM_TX_CELLS_VCQ	per VCQ 64-bit ATM Tx cell count

ix_e_cc_atm_pos_tx_statistics_info Types

<code>IX_CC_ATM_TX_ALL_COUNTERS_VCQ</code>	all supported per-VcQ ATM counters of a given VcQ
<code>IX_CC_ATM_TX_PACKETS_PORT</code>	per port 64-bit ATM Tx packet count
<code>IX_CC_ATM_TX_CELLS_PORT</code>	per port 64-bit ATM Tx cell count
<code>IX_CC_ATM_TX_ALL_COUNTERS_PORT</code>	all supported per-port ATM counters of a given port
<code>IX_CC_POS_TX_PACKETS</code>	per port 64-bit POS Tx packet count
<code>IX_CC_POS_TX_DROP_PACKETS</code>	per port 64-bit POS drop packet count
<code>IX_CC_POS_TX_BYTES</code>	per port 64-bit POS Tx byte count
<code>IX_CC_POS_TX_DROP_BYTES</code>	per port 64-bit POS Tx drop byte count
<code>IX_CC_POS_TX_ALL_COUNTERS</code>	all supported per-port POS counters of a given port

8.2.1.3 ix_cc_atm_pos_tx_cb_get_statistics_info

This is the prototype of callback functions for calls to `ix_cc_atm_pos_tx_async_get_statistics_info()`.

C Syntax

```
ix_error (*ix_cc_atm_pos_tx_cb_get_statistics_info)(
    ix_error arg_Result,
    void *arg_pContext,
    ix_uint64 *arg_pBuffer,
    ix_uint32 arg_MsgLen);
```

Input

<code>arg_Result</code>	The error code returned by the ATM/POS TX core component.
<code>arg_pContext</code>	The pointer to the calling application-provided context.
<code>arg_pBuffer</code>	the pointer to the buffer that stores the obtained statistics information.
<code>arg_MsgLen</code>	<p>The size of <code>arg_pBuffer</code> in byte. For entity of type <code>ALL_COUNTERS</code>, the size of the buffer depends on the number of 64-bit counters to be returned. For example, for:</p> <ul style="list-style-type: none"> <code>IX_CC_ATM_TX_ALL_COUNTERS_VCQ</code> and <code>IX_CC_ATM_TX_ALL_COUNTERS_PORT</code>, the size is $2 * 8$ bytes = 16 bytes. For <code>IX_CC_POS_TX_ALL_COUNTERS</code>, the buffer size is $4 * 8$ bytes = 32 bytes. For any other single counter, the buffer size is 8 bytes.

8.2.2 ix_cc_atm_pos_tx_async_get_interface_state()

This function gets the state information of an ATM or POS interface port. It is an asynchronous function and the current state of the interface is returned through the calling application-provided callback function. The function may be used by an application to obtain the interface state information.

The function generates a `IX_CC_ATM_POS_TX_MSG_GET_INTERFACE_STATE` message and passes it to ATM/POS TX core component by calling the send message API of the Message support Library.

The message handler function of ATM/POS TX core component—`ix_cc_atm_pos_tx_msg_handler()`, calls the corresponding library API—`ix_cc_atm_pos_tx_async_get_interface_state()`, for handling the message.

C Syntax

```
ix_error ix_cc_atm_pos_tx_async_get_interface_state (
    ix_cc_atm_pos_tx_if_state_context *arg_pStateContext,
    ix_cc_atm_pos_tx_cb_get_interface_state arg_Callback);
```

Input

`arg_pStateContext` The parameter is defined as follows:

Output/Returns

Return Value	<p>Returns a valid <code>ix_error</code>.</p> <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • A valid <code>ix_error</code>—the operation failed.
--------------	---

8.2.2.1 ix_s_cc_atm_pos_tx_if_state_context

This data structure specifies the type of statistics information requested, the index to a virtual circuit, and the requestor for the function call. It is used with `ix_cc_atm_pos_tx_cb_get_interface_state` Callback and `ix_e_cc_atm_pos_tx_if_state`.

C Syntax

```
typedef struct ix_s_cc_atm_pos_tx_if_state_context
{
    ix_uint8 port;
    void *pUserContext;
} ix_cc_atm_pos_tx_if_state_context;
```

Input

port	The interface port number whose interface state is to be returned.
pUserContext	The pointer to the calling application-provided context structure. This is used by the core component in the callback function so that the calling application can identify the instance of the request.
arg_Callback	The pointer to the callback routine. See ix_cc_atm_pos_tx_cb_get_interface_state Callback .

8.2.2.2 ix_cc_atm_pos_tx_cb_get_interface_state Callback

This is the prototype of callback functions for calls to `ix_cc_atm_pos_tx_async_get_interface_state()`. This callback is used with `ix_e_cc_atm_pos_tx_if_state`.

C Syntax

```
ix_error (*ix_cc_atm_pos_tx_cb_get_interface_state) (
    ix_error arg_Result,
    void *arg_pContext,
    ix_cc_atm_pos_tx_if_state *arg_pState);
```

Input

<code>arg_Result</code>	the error code returned by ATM/POS TX core component
<code>arg_pContext</code>	the pointer to the calling application-provided context.
<code>arg_pState</code>	The pointer for storing the obtained interface state. See <code>ix_e_cc_atm_pos_tx_if_state</code> .

8.2.2.3 ix_e_cc_atm_pos_tx_if_state

The pointer for storing the obtained interface state used with `ix_cc_atm_pos_tx_cb_get_interface_state` Callback.

C Syntax

```
typedef enum ix_e_cc_atm_pos_tx_if_state
{
    IX_CC_ATM_POS_TX_IF_STATE_DOWN = 0,
    IX_CC_ATM_POS_TX_IF_STATE_UP
} ix_cc_atm_pos_tx_if_state;
```

8.3 Library API

Table 8-3 shows the library API provided by ATM/POS TX core component.

Table 8-3. ATM/POS Interface Tx Library Functions

API	Description
<code>ix_cc_atm_pos_tx_get_statistics_info()</code>	Obtains the statistics info
<code>ix_cc_atm_pos_tx_get_interface_state()</code>	Gets the ATM or POS interface state
<code>ix_cc_atm_pos_tx_set_property()</code>	Sets or change a dynamic property

8.3.1 `ix_cc_atm_pos_tx_get_statistics_info()`

This function returns the requested statistical information. It is called internally by the ATM/POS TX core component in its message handler routine for processing the `IX_CC_ATM_POS_TX_MSG_GET_STATISTICS_INFO` message. It may also be called by an IXA Portability Framework application for getting the statistical information.

C Syntax

```
ix_error ix_cc_atm_pos_tx_get_statistics_info (
    ix_cc_atm_pos_tx_statistics_info arg_entity,
    ix_uint32 arg_index,
    ix_cc_atm_pos_tx_statistics_info_data *arg_pBuffer,
    void *arg_pContext);
```

Input

<code>arg_entity</code>	The type of statistical information. See also the messaging API, ix_cc_atm_pos_tx_get_statistics_info() for the type of statistical information supported.
<code>arg_index</code>	The index number to the statistics information. See also the messaging API, ix_cc_atm_pos_tx_async_get_statistics_info() for a detailed usage of the index number.
<code>arg_pBuffer</code>	The pointer to the <code>ix_cc_atm_pos_tx_statistics_info_data</code> structure where the requested entity is stored.
<code>arg_pContext</code>	The pointer to the component control block memory allocated earlier in its init function. See ix_s_cc_atm_pos_tx_statistics_info_data .

Note: For 64-bit counters, the data length is 8 bytes. It is the calling application's responsibility to allocate the required memory, 8 bytes, and assign it to the `pDataBuffer` pointer.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. A valid <code>ix_error</code>—the operation failed.
--------------	---

8.3.1.1 `ix_s_cc_atm_pos_tx_statistics_info_data`

The pointer to the component control block memory allocated earlier in its init function. This is used with `ix_cc_atm_pos_tx_get_statistics_info()`.

C Syntax

```
typedef struct ix_s_cc_atm_pos_tx_statistics_info_data
{
    ix_uint32 dataLength; /* length in bytes of the data buffer */
    void *pDataBuffer; /* pointer to a buffer where the */
                      /* statistics data will be stored */
} ix_cc_atm_pos_tx_statistics_info_data;
```

8.3.2 `ix_cc_atm_pos_tx_get_interface_state()`

This function returns the interface state information. It is called internally by ATM/POS TX core component in its message handler routine for processing the `IX_CC_ATM_POS_TX_MSG_GET_INTERFACE_STATE` message. It may also be called by an IXA Portability Framework-based application for getting the interface status.

C Syntax

```
ix_error ix_cc_atm_pos_tx_get_interface_state (
    ix_uint8 arg_Port,
    ix_cc_atm_pos_tx_if_state *arg_pIfState,
    void *arg_pContext);
```

Input

<code>arg_Port</code>	The interface port number whose state is to be returned.
<code>arg_pIfState</code>	The pointer to the memory location where the interface state is filled in (refer to the <code>ix_cc_atm_pos_tx_async_get_interface_state()</code> messaging API for the data type). This memory needs to be allocated by the calling application. The core component uses this pointer to pass the state back.
<code>arg_pContext</code>	The pointer to the component control block memory allocated earlier in its init function.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. A valid <code>ix_error</code>—the operation failed.
--------------	---

8.3.3 `ix_cc_atm_pos_tx_set_property()`

This function implements the generic library API for setting one or more dynamic properties provided by the ATM/POS TX core component. The function is called either by ATM/POS TX core component internally in its property message handler routine or by an IXA Portability Framework-based application.

C Syntax

```
ix_error ix_cc_atm_pos_tx_set_property (
    ix_uint32 arg_PropId,
    ix_cc_properties *arg_pProperty,
    void *arg_pContext);
```

Input

<code>arg_PropId</code>	<p>The property to be changed. For ATM/POS TX, these properties include:</p> <ul style="list-style-type: none"> <code>IX_CC_PROPERTY_UPDATE_IF_STATUS</code>—state of the interface (whether it is UP or DOWN). <p>The calling application can set multiple properties in a single function call by ORing—set up some or all the property IDs and corresponding fields in the <code>ix_cc_properties</code> structure.</p>
<code>arg_pProperty</code>	The pointer to the <code>ix_cc_properties</code> data structure that contains a valid port number to which the property is to be set and the associated properties.
<code>arg_pContext</code>	the pointer to the component control block memory allocated earlier in its init function.

Output/Returns

Return Value	<p>Returns a valid <code>ix_error</code>.</p> <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded, if the desired state is the same as the existing state. <code>IX_CC_ATM_POS_TX_ERROR_INTERNAL_ERROR</code>—the operation failed, due an error countered from underlying component. The property is not changed.
--------------	---

Ethernet TX provides the following services:

- Setting static configuration of Ethernet TX microblock through patching symbols.
- Exposing interface to a system application for setting and retrieving configuration and statistical parameters.
- Exposing interface to the calling application for accessing the Ethernet ARP library for ARP cache entry update, creation, and deletion, etc.
- Exposing interface to port property master for setting dynamic configuration data such as port IP address and port layer-2 address in its internal Local Interface Table.
- Providing packet handler routine for exception packets from Ethernet TX microblock and other core component.
- Configuring the multi-port Gigabit Ethernet media card by utilizing the device driver of the media card.

For complete details, see [Chapter 46, “Ethernet TX Core Component”](#) of the *Intel® Internet Exchange Architecture Software Building Blocks Developer's Manual*.

9.1 Data Structures

[Table 9-1](#) summarizes the data structures defined by the core component.

Table 9-1. Ethernet TX Core Component Data Structures

Name	Description
<code>ix_cc_eth_tx_next_hop_info</code>	This data structure defines the parameters required when using the external API for accessing the ARP cache.
<code>ix_ether_addr</code>	This data structure specifies a layer-2 address.
<code>ix_cc_eth_tx_if_state</code>	This enumeration represents the interface state of an Ethernet port.

9.1.1 `ix_cc_eth_tx_next_hop_info`

This data structure defines the parameters required when using the external API for accessing the ARP cache.

C Syntax

```
typedef struct ix_s_cc_eth_tx_next_hop_info {
    ix_int32 l2Index;
    ix_int32 nextHopIp; /*next hop IP in network-byte order */
    ix_int32 outputPort; /*egress output port in network-byte order */
    ix_ether_addr l2Addr; /* layer-2 address */
} ix_cc_eth_tx_next_hop_info;
```

Data Members

<code>l2Index</code>	A 16-bit layer-2 index into the L2 table.
<code>nextHopIp</code>	The next hop IP address associated with the L2 index—in network-byte order.
<code>outputPort</code>	A 16-bit egress output port with the bit arrangement specified in <i>Intel® Internet Exchange Architecture Software Building Blocks Developer's Manual</i> .
<code>l2Addr</code>	A layer-2 address as described in <code>ix_ether_addr</code> .

9.1.2 `ix_ether_addr`

This data structure specifies a layer-2 address.

C Syntax

```
typedef struct ix_s_ether_addr {
    ix_uint8 etherDstAddr[6];
} ix_ether_addr;
```

Data Members

<code>etherDstAddr</code>	An six-byte Ethernet address—in network-byte order.
---------------------------	---

9.1.3 ix_cc_eth_tx_if_state

This enumeration represents the interface state of an Ethernet port.

C Syntax

```
typedef enum ix_e_cc_eth_tx_if_state {
    IX_CC_ETH_TX_IF_STATE_DOWN = 0,
    IX_CC_ETH_TX_IF_STATE_UP
} ix_cc_eth_tx_if_state;
```

Defined Values

IX_CC_ETH_TX_IF_STATE_DOWN The Ethernet port is down.

IX_CC_ETH_TX_IF_STATE_UP The Ethernet port is up.

9.2 Core Component Infrastructure API

This section describes the Core Component Infrastructure functions provided by the Ethernet TX core component. These functions are summarized in [Table 9-2](#).

Table 9-2. Ethernet TX Core Component Infrastructure API

API	Description
<code>ix_cc_eth_tx_init()</code>	Initializes the core component.
<code>ix_cc_eth_tx_fini()</code>	Terminates the core component.
<code>ix_cc_eth_tx_msg_handler()</code>	Message handler for processing ARP and statistics requests.
<code>ix_cc_eth_tx_property_msg_handler()</code>	Message handler for processing port-related properties.
<code>ix_cc_eth_tx_pkt_handler()</code>	Packet handler for processing exception packets.

9.2.1 ix_cc_eth_tx_init()

Initializes the core component. It must be called and must return successfully before any other function in the core component is called. It performs static configuration for the Ethernet TX microblock by allocating and patching required variables and memory blocks on the microblock.

Patching symbols is performed through a call to the Resource Manager's function. This call initializes the L2 library to create the L2 table and obtain the base address of the table so that it can be patched into the microblock. It initializes the ARP module for the creation of the ARP cache. It also creates the Local Interface table for storing the port IP address and layer-2 address for the ARP module to access. Finally, it registers the packet handler and message handlers for the input IDs it defines using the handler registration API.

C Syntax

```
ix_error ix_cc_eth_tx_init(
    ix_cc_handle arg_hCcHandle,
    void **arg_ppContext);
```

Input

`arg_hCcHandle` The handle to the core component.

Output/Return

`arg_ppContext` The location where the pointer to the control block allocated by the core component is to be stored. The control block is internal to the core component and contains Transmit Context memory and other variables and internal data structures. This pointer is used later by the [ix_cc_eth_tx_fini\(\)](#) function to free memory when this core component is destroyed.

Return Value Returns a valid `ix_error`.

- `IX_SUCCESS`—the operation succeeded.
- A valid `ix_error`—the operation failed.

9.2.2 [ix_cc_eth_tx_fini\(\)](#)

Terminates the core component. It is executed when the execution engine running the core component is shut down. This function frees all memory and resources allocated by this component from the time of the call to the [ix_cc_eth_tx_init\(\)](#) function until the call to this function.

C Syntax

```
ix_error ix_cc_eth_tx_fini(
    ix_cc_handle arg_hCcHandle,
    void *arg_pContext);
```

Input

`arg_hCcHandle` A handle to the core component.

`arg_pContext` A pointer to the control block memory allocated earlier in [ix_cc_eth_tx_init\(\)](#). The termination routine uses this pointer to de-allocate the control block memory.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. A valid <code>ix_error</code>—the operation failed.
--------------	---

9.2.3 `ix_cc_eth_tx_msg_handler()`

The message handler associated with the messaging input to the Ethernet TX core component. This function calls an associated library API to process each message it defines.

The Ethernet TX core component has an input for handling messages sent from the Messaging API. The input ID for this component's messaging input is defined in the System Application file `bindings.h` as `IX_ETH_TX_MSG_INPUT`.

C Syntax

```
ix_error ix_cc_eth_tx_msg_handler(
    ix_buffer_handle arg_hMsg,
    ix_uint32 arg_UserData
    void *arg_pContext);
```

Input

<code>arg_hMsg</code>	The buffer handle that has message-specific information embedded. Typically, the embedded information consists of the parameters for the invocation of the associated messaging API function.
<code>arg_UserData</code>	<p>The message type.</p> <p>The following defined message types are currently supported:</p> <ul style="list-style-type: none"> <code>IX_CC_ETH_TX_MSG_GET_STATISTICS_INFO</code>—obtain the statistics info. <code>IX_CC_ETH_TX_MSG_GET_INTERFACE_STATE</code>—obtain the state of an interface port <code>IX_CC_ETH_TX_MSG_CREATE_ARP_ENTRY</code>—create or update a dynamic ARP cache entry when a new L2 index is created. <code>IX_CC_ETH_TX_MSG_ADD_ARP_ENTRY</code>—add a static ARP entry. <code>IX_CC_ETH_TX_MSG_DEL_ARP_ENTRY</code>—delete an ARP entry. <code>IX_CC_ETH_TX_MSG_PURGE_ARP_CACHE</code>—clear the entire ARP cache. <code>IX_CC_ETH_TX_MSG_DUMP_ARP_CACHE</code>—print out the content of the ARP cache.
<code>arg_pContext</code>	A handle to the core component created internal control block structure created by the core component initialization function, <code>ix_cc_eth_tx_init()</code> .

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. A valid <code>ix_error</code>—the operation failed.
--------------	---

IX_ETH_TX_MSG_INPUT Symbolic Constant

```
#define IX_ETH_TX_MSG_INPUT
```

9.2.4 ix_cc_eth_tx_property_msg_handler()

The message handler associated with port-messaging input to the Ethernet TX. The function updates the Local Interface table with the new property by calling the Library API, `ix_cc_eth_tx_set_property()`.

The Ethernet TX core component defines an input for the calling application to send port-related property messages. This includes the port status, port IP address and port Ethernet address. The input ID for port messaging input is defined in the System Application file `bindings.h` as `IX_ETH_TX_PROPERTY_MSG_INPUT`.

C Syntax

```
ix_error ix_cc_eth_tx_property_msg_handler(
    ix_buffer_handle arg_hMsg,
    ix_uint32 arg_UserData
    void *arg_pContext);
```

Input

<code>arg_hMsg</code>	The buffer handle that has embedded message-specific information. The buffer embeds the <code>ix_cc_properties</code> structure, which has the required property set in one of its structure fields and which also specifies the port.
<code>arg_UserData</code>	The message type. The following message types are defined: <ul style="list-style-type: none"> <code>IX_CC_COMMON_MSG_ID_PROP_UPDATE</code>—adds or deletes an interface IP address for a port or sets or changes the MAC address of a port or sets the interface state to up or down.
<code>arg_pContext</code>	A handle to the core component created internal control block structure created by the core component initialization function, <code>ix_cc_eth_tx_init()</code> .

Output/Returns

Return Value	Returns a valid <code>ix_error</code> .
	<ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • A valid <code>ix_error</code>—the operation failed.

`IX_ETH_TX_PORT_MSG_INPUT` Symbolic Constant

```
#define IX_ETH_TX_PROPERTY_MSG_INPUT
```

9.2.5 `ix_cc_eth_tx_pkt_handler()`

The Ethernet TX core component can receive packets from other components. Currently, it defines two inputs for handling packets received from the Ethernet TX microblock and the Ethernet RX core component. This handler function processes the following two types of exception packets:

- ARP packets from `IX_CC_ETH_TX_ARP_PKT_INPUT`
- Exception IP packets from `IX_CC_ETH_TX_COMMON_PKT_INPUT`

When the Ethernet RX microblock receives an ARP packet from the network, it sends it to the Ethernet RX core component as an exception packet. The Ethernet RX core component sends the ARP packet to the Ethernet TX core component through the PCI bus on the egress side. The Ethernet TX core component passes the ARP packet to its ARP module for processing. The ARP module processes the packet based on that packet's type. This may include operations like ARP cache look-up and update, sending an ARP reply, sending any previously held IP packet, or any other valid ARP operation. See also the ARP function documented in [Section 10.2.10](#), "`ix_cc_arp_process_arp_pkts()`" for details.

The exception packet can also be an IP packet from the Ethernet TX microblock which is treated as an exception due to un-resolved L2 information in the L2 table. For this type of packets, the Ethernet TX core component first checks the L2 index extracted from the packet meta data. If the L2 index is 0, it indicates the packet is destined to a directly connected local host and a new L2 index needs to be created. The Ethernet TX core component retrieves the destination IP address from the IP packet and then sends a message—a 32-bit destination IP address, to its message output, `IX_CC_ETH_TX_FP_MODULE_MSG_OUTPUT`, with message type `IX_NEW_DIRECT_HOST_MSG_ID`. The output can be mapped to CP-PDK for the creation of a new L2 index.

If the L2 index is not 0, the Ethernet TX core component accesses its ARP module for the resolution of the L2 information. The ARP module either performs an ARP cache lookup and synchronization between the ARP cache and the L2 table or it generates an ARP request packet for soliciting the L2 information. In the case of an ARP request packet, the IP packet is sent back to the microblock for transmission. In the case of ARP cache lookup, the IP packet is held by the ARP module and sent out later when corresponding the ARP reply is received. See also the ARP function documented in [Section 10.2.9](#), "`ix_cc_arp_resolve_l2_addr()`" for details.

C Syntax

```
ix_error ix_cc_eth_tx_pkt_handler(
    ix_buffer_handle arg_Pkt,
    ix_uint32 arg_ExceptionCode,
```

```
void *arg_pContext);
```

Input

arg_Pkt	A buffer handle to the exception packet. For ARP packets, the Ethernet TX core component assumes the buffer contains a full Ethernet packet including the 14-byte Ethernet header. For IP packets, the Ethernet TX core component assumes the metadata is prepended at the beginning of the packet.
arg_ExceptionCode	<p>The packet types.</p> <p>The following defined packet types are currently supported:</p> <ul style="list-style-type: none"> • ARP packets from the Ethernet RX core component. • Exception packets from the Ethernet TX microblock.
arg_pContext	The pointer to a component context structure—that is passed to the core component when a packet arrives.

Output/Returns

Return Value	<p>Returns a valid <code>ix_error</code>.</p> <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • A valid <code>ix_error</code>—the operation failed. • <code>IX_CC_ERROR_UNDEFINED_EXCP</code>—the packet cannot be handled because of an undefined exception code. <p>For any of these error conditions, the packet is dropped and the buffer is returned to system buffer pool.</p>
--------------	---

9.3 Messaging API

This section describes the functions comprising the Ethernet TX core component Messaging API. Table 9-3 summarizes these functions.

Table 9-3. Ethernet TX Messaging Functions

API	Description
<code>ix_cc_eth_tx_async_get_statistics_info()</code>	Returns the requested statistics information.
<code>ix_cc_eth_tx_async_get_interface_state()</code>	Returns the state information of an Ethernet interface port.
<code>ix_cc_eth_tx_async_create_arp_entry()</code>	Create or update a dynamic ARP entry
<code>ix_cc_eth_tx_async_add_arp_entry()</code>	Add a static ARP entry
<code>ix_cc_eth_tx_async_del_arp_entry()</code>	Delete an ARP entry
<code>ix_cc_eth_tx_async_purge_arp_cache()</code>	Clear the ARP cache
<code>ix_cc_eth_tx_async_dump_arp_cache()</code>	Dump the ARP cache to standard output device

9.3.1 `ix_cc_eth_tx_async_get_statistics_info()`

Returns the requested statistics information. This is an asynchronous function—the state of the interface is returned through a user-provided callback function.

C Syntax

```
ix_error ix_cc_eth_tx_async_get_statistics_info(
    ix_cc_eth_tx_statistics_info_context *arg_pMgInfoContext,
    ix_cc_eth_tx_cb_get_statistics_info arg_Callback);
```

Input

<code>arg_pMgInfoContext</code>	Defines the statistics request. See ix_cc_eth_tx_statistics_info_context in Section 9.3.1.1.
<code>arg_Callback</code>	A pointer to the callback routine used to return the result of the statistics request. This callback has the function prototype, ix_cc_eth_tx_cb_get_statistics_info() . See ix_cc_eth_tx_statistics_info in Section 9.3.1.2.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. A valid <code>ix_error</code>—the operation failed.
--------------	---

9.3.1.1 `ix_cc_eth_tx_statistics_info_context`

This is the function prototype of a callback routine for calls to [ix_cc_eth_tx_async_get_statistics_info\(\)](#).

C Syntax

```
typedef struct ix_s_cc_eth_tx_statistics_info_context {
    ix_cc_eth_tx_statistics_info entity;
    ix_uint32 index,
    void *pUserContext;
} ix_cc_eth_tx_statistics_info_context;
```

Input

entity	Specifies the type of statistics information requested. The supported types of information are specified by the enumerated type, ix_cc_eth_tx_statistics_info . See Section 9.3.1.3 .
index	Specifies the index number representing the port number for which statistics information is requested. For all valid values of ix_cc_eth_tx_cb_get_statistics_info() the index number can be any number from zero to <code>IX_CC_ETH_TX_MAX_NUM_PORTS</code> minus one representing Ethernet port zero to port <code>IX_CC_ETH_TX_MAX_NUM_PORTS</code> minus one.
pUserContext	The pointer to a user-provided context structure, used by the core component in the callback function to identify the originator of the request.

9.3.1.2 [ix_cc_eth_tx_statistics_info](#)

The enumerated type listing the types of statistics information maintained by the Ethernet TX core component.

C Syntax

```
typedef enum ix_e_cc_eth_tx_statistics_info {
    IX_CC_ETH_TX_OCTETS_OK,
    IX_CC_ETH_TX_OCTETS_BAD,
    IX_CC_ETH_TX_UC_PKTS,
    IX_CC_ETH_TX_MC_PKTS,
    IX_CC_ETH_TX_BC_PKTS,
    IX_CC_ETH_TX_PKTS_64,
    IX_CC_ETH_TX_PKTS_65_127,
    IX_CC_ETH_TX_PKTS_128_255,
    IX_CC_ETH_TX_PKTS_256_511,
    IX_CC_ETH_TX_PKTS_512_1023,
    IX_CC_ETH_TX_PKTS_1024_1518,
    IX_CC_ETH_TX_PKTS_1519_MAX,
    IX_CC_ETH_TX_DEFERRED_ERR,
    IX_CC_ETH_TX_TOTAL_COLLISION,
    IX_CC_ETH_TX_SINGLE_COLLISION,
    IX_CC_ETH_TX_MUL_COLLISION,
    IX_CC_ETH_TX_LATE_COLLISION,
    IX_CC_ETH_TX_EXCV_COLLISION,
    IX_CC_ETH_TX_EXCV_DEFERRED_ERR,
    IX_CC_ETH_TX_EXCV_LEN_DROP,
    IX_CC_ETH_TX_UNDERRUN,
    IX_CC_ETH_TX_VLAN_TAG,
    IX_CC_ETH_TX_CRC_ERR,
    IX_CC_ETH_TX_PAUSE_FRAME,
    IX_CC_ETH_TX_FC_COLLISION_SEND,
    IX_CC_ETH_TX_ALL_DRIVER_COUNTERS;
}
```

```
} ix_cc_eth_tx_statistics_info;
```

Defined Types

<code>IX_CC_ETH_TX_OCTETS_OK</code>	The number of bytes transmitted in all legal frames.
<code>IX_CC_ETH_TX_OCTETS_BAD</code>	The number of bytes transmitted in all bad frames.
<code>IX_CC_ETH_TX_UC_PKTS</code>	The number of unicast packets transmitted.
<code>IX_CC_ETH_TX_MC_PKTS</code>	The number of multicast packets transmitted.
<code>IX_CC_ETH_TX_BC_PKTS</code>	The number of broadcast packets transmitted.
<code>IX_CC_ETH_TX_PKTS_64</code>	The number of packets transmitted that are 64 bytes in length.
<code>IX_CC_ETH_TX_PKTS_65_127</code>	The number of packets transmitted that are 65 to 127 bytes in length.
<code>IX_CC_ETH_TX_PKTS_128_255</code>	The number of packets transmitted that are 128 to 255 bytes in length.
<code>IX_CC_ETH_TX_PKTS_256_511</code>	The number of packets transmitted that are 256 to 511 bytes in length.
<code>IX_CC_ETH_TX_PKTS_512_1023</code>	The number of packets transmitted that are 512 to 1023 bytes in length.
<code>IX_CC_ETH_TX_PKTS_1024_1518</code>	The number of packets transmitted that are 1024 to 1518 bytes in length.
<code>IX_CC_ETH_TX_PKTS_1519_MAX</code>	The number of packets transmitted that are greater than 1518 bytes in length.
<code>IX_CC_ETH_TX_DEFERRED_ERR</code>	The number of times an initial frame transmission attempt was postponed due to another frame already being transmitted on the Ethernet network.
<code>IX_CC_ETH_TX_TOTAL_COLLISION</code>	The sum of all collision events.
<code>IX_CC_ETH_TX_SINGLE_COLLISION</code>	The number of successfully transmitted frames on a particular interface where the transmission is inhibited by exactly one collision.
<code>IX_CC_ETH_TX_MUL_COLLISION</code>	The number of successfully transmitted frames on a particular interface for which transmission is inhibited by more than one collision.
<code>IX_CC_ETH_TX_LATE_COLLISION</code>	The number of times a collision is detected on a particular interface later than 512 bit-times into the transmission of a packet. These frames were terminated and discarded.

Defined Types (Continued)

<code>IX_CC_ETH_TX_EXCV_COLLISION</code>	The number of frames which collided 16 times and were then discarded by the MAC. Not effected by multiple collisions.
<code>IX_CC_ETH_TX_EXCV_DEFERRED_ERR</code>	The number of frames for which transmission is postponed more than twice the maximum frame size due to another frame already being transmitted on the Ethernet network. The MAC discards these frames.
<code>IX_CC_ETH_TX_EXCV_LEN_DROP</code>	The number of frames for which transmissions are aborted by the MAC because the frame is longer than the maximum frame size.
<code>IX_CC_ETH_TX_UNDERRUN</code>	The number of internal transmission errors—these errors result in the MAC ending the transmission before the end of the frame because the MAC did not get the needed data in time for transmission. The frames are lost and a fragment or a CRC error is transmitted.
<code>IX_CC_ETH_TX_VLAN_TAG</code>	The number of good frames with VLAN tags.
<code>IX_CC_ETH_TX_CRC_ERR</code>	The number of frames which were transmitted with a legal size but with the wrong CRC field—also called the FCS field.
<code>IX_CC_ETH_TX_PAUSE_FRAME</code>	The number of pause frames transmitted.
<code>IX_CC_ETH_TX_FC_COLLISION_SEND</code>	The number of times a collision is purposely generated on incoming frames to avoid reception of traffic while the port is in half-duplex mode, has flow control enabled, and does not have sufficient memory to receive more frames.
<code>IX_CC_ETH_TX_ALL_DRIVER_COUNTERS</code>	All of the above counters for a specified port.

9.3.1.3 `ix_cc_eth_tx_cb_get_statistics_info()`

The function prototype used to return statistics information.

C Syntax

```
ix_error (*ix_cc_eth_tx_cb_get_statistics_info)(
    ix_error   arg_Result,
    void       *arg_pContext,
    ix_uint64  *arg_pBuffer,
    ix_uint32  arg_MsgLen);
```

Input

<code>arg_pContext</code>	A pointer to the user-provided context.
<code>arg_pBuffer</code>	A pointer to the buffer used to return statistics information. <ul style="list-style-type: none"> For individual statistics attributes, the buffer length is eight bytes (64-bits). For <code>IX_CC_ETH_TX_ALL_DRIVER_COUNTERS</code> type, the buffer length is $(8 * 25) = 200$ bytes.
<code>arg_MsgLen</code>	The length—in number of bytes—of the buffer. <ul style="list-style-type: none"> For individual statistics attributes, the buffer length is eight bytes—64-bits. For <code>IX_CC_ETH_TX_ALL_DRIVER_COUNTERS</code> type, the buffer length is $(8 * 25) = 200$ bytes.

Output/Returns

<code>arg_Result</code>	The error code returned by the Ethernet TX core component specifying the result of the statistics request. It is either <code>IX_SUCCESS</code> for success or a valid <code>ix_error</code> code for failure.
Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. A valid <code>ix_error</code>—the operation failed.

9.3.2 `ix_cc_eth_tx_async_get_interface_state()`

This function implements the messaging API that returns the state information of an Ethernet interface port. It is an asynchronous function and the current state of the interface will be returned through user-provided callback function. The function may be used by an application to obtain the interface state information. The function generates an

`IX_CC_ETH_TX_MSG_GET_INTERFACE_STATE` message and passes it to Ethernet TX core component through calling the send-message API of the Message Support library. The message handler function of the Ethernet TX core component—`ix_cc_eth_tx_msg_handler()`—invokes the corresponding library API—`ix_cc_eth_tx_get_interface_state()`—to process the message.

C Syntax

```
ix_error ix_cc_eth_tx_async_get_interface_state(
    ix_cc_eth_tx_if_state_context *arg_pStateContext,
    ix_cc_eth_tx_cb_get_interface_state arg_Callback);
```

Input

<code>arg_pStateContext</code>	The interface state context as defined by ix_cc_eth_tx_if_state_context .
<code>arg_Callback</code>	A pointer to the callback routine. The function prototype of the callback is defined by ix_cc_eth_tx_cb_get_interface_state() .

Output/Returns

Return Value	Returns <code>IX_SUCCESS</code> if successful or a valid <code>ix_error</code> for failure.
--------------	---

9.3.2.1 [ix_cc_eth_tx_if_state_context](#)

The interface state context as defined by data structure.

C Syntax

```
typedef struct ix_s_cc_eth_tx_if_state_context {
    ix_uint8 port;
    void *pUserContext;
} ix_cc_eth_tx_if_state_context;
```

Data Members

<code>port</code>	Specifies the interface port number whose interface state is to be returned.
<code>pUserContext</code>	A pointer to the user-provided context structure used by the core component in the callback function to identify the instance of the request.

9.3.2.2 [ix_cc_eth_tx_cb_get_interface_state\(\)](#)

The function prototype defining the callback function used to return the results of the ARP entry creation request.

C Syntax

```
ix_error (*ix_cc_eth_tx_cb_get_interface_state)(
    ix_error arg_Result,
    void *arg_pContext,
    ix_cc_eth_tx_if_state *arg_pState);
```

Input

<code>arg_Result</code>	The error code returned by the Ethernet TX core component for the result of the interface state request. Returns one of: <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. A valid <code>ix_error</code>—the operation failed.
<code>arg_pContext</code>	A pointer to the user-provided context.
<code>arg_pState</code>	A pointer for storing the obtained interface state. The data type is defined in Section 9.1.3 , “ <code>ix_cc_eth_tx_if_state</code> .”

9.3.3 `ix_cc_eth_tx_async_create_arp_entry()`

This function implements the Messaging API for creating or updating a dynamic ARP entry. It is used by the Forwarding Plane module when a new L2 index is created by the Next Hop ID Manager of the Control Plane PDK. The function generates an `IX_CC_ETH_TX_MSG_CREATE_ARP_ENTRY` message and passes it to Ethernet TX core component. The message handler function of Ethernet TX core component invokes the corresponding library API, that is, `ix_cc_eth_tx_create_arp_entry()`, to process the message.

C Syntax

```
ix_error ix_cc_eth_tx_async_create_arp_entry(
    ix_cc_eth_tx_next_hop_info *arg_pArpInfo,
    ix_cc_eth_tx_cb_create_arp_entry arg_Callback,
    void *arg_pContext);
```

Input

<code>arg_pArpInfo</code>	The ARP information structure containing the information used to create or update the ARP entry. The function assumes the L2 index, next hop IP, and port ID are valid. The <code>l2Addr</code> field is ignored by the function.
<code>arg_Callback</code>	A pointer to the callback routine with the function prototype, <code>ix_cc_eth_tx_cb_create_arp_entry()</code> . See Section 9.3.3.1 .
<code>arg_pContext</code>	The pointer to a user-provided context structure, which is used by the core component in the callback function for the caller to identify the origin of the request.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. A valid <code>ix_error</code>—the operation failed.
--------------	---

9.3.3.1 `ix_cc_eth_tx_cb_create_arp_entry()`

The function prototype defining the callback function used to return the results of the ARP entry creation request.

C Syntax

```
ix_error (*ix_cc_eth_tx_cb_create_arp_entry)(
    ix_error arg_Result,
    void *arg_pContext);
```

Input

<code>arg_Result</code>	The <code>ix_error</code> returned by the Ethernet TX core component for the result of the function call. It is either <code>IX_SUCCESS</code> if the operation succeeds or a valid <code>ix_error</code> if the operation fails. See also <code>ix_cc_eth_tx_create_arp_entry()</code> for the defined error codes.
<code>arg_pContext</code>	A pointer to the user-provided context.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. A valid <code>ix_error</code>—the operation failed.
--------------	---

9.3.4 `ix_cc_eth_tx_async_add_arp_entry()`

Implements the messaging API for adding a static ARP entry. The function generates an `IX_CC_ETH_TX_MSG_ADD_ARP_ENTRY` message and passes it to the Ethernet TX core component. The message handler function of the Ethernet TX core component calls the corresponding library API—that is, `ix_cc_eth_tx_add_arp_entry()`—to process the message.

C Syntax

```
ix_error ix_cc_eth_tx_async_add_arp_entry(
    ix_cc_eth_tx_next_hop_info *arg_pArpInfo,
    ix_cc_eth_tx_cb_add_arp_entry arg_Callback,
    void *arg_pContext);
```


Input

<code>arg_pArpInfo</code>	A pointer to the ARP data structure that contains the ARP information to add. The function assumes all fields in the structure are valid.
<code>arg_Callback</code>	The pointer to the callback routine with the function prototype, <code>ix_cc_eth_tx_cb_add_arp_entry()</code> .
<code>arg_pContext</code>	The pointer to user-provided context structure used by the core component in the callback function to identify the origin of the request.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • A valid <code>ix_error</code>—the operation failed.
--------------	---

9.3.4.1 [`ix_cc_eth_tx_cb_add_arp_entry\(\)`](#)

The function prototype for the callback used to notify the calling application of the results of a call to [`ix_cc_eth_tx_async_add_arp_entry\(\)`](#).

C Syntax

```
ix_error (*ix_cc_eth_tx_cb_add_arp_entry)(
    ix_error arg_Result,
    void *arg_pContext);
```

Input

<code>arg_Result</code>	The error code returned by the Ethernet TX core component representing the result of the function call. It is either <code>IX_SUCCESS</code> if the operation succeeds or a valid <code>ix_error</code> if the operation fails. See <code>ix_cc_eth_tx_add_arp_entry()</code> for the defined error codes.
<code>arg_pContext</code>	The second argument is the pointer to the user provided context.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • A valid <code>ix_error</code>—the operation failed.
--------------	---

9.3.5 `ix_cc_eth_tx_async_del_arp_entry()`

The messaging API function for deleting an ARP entry. The entry can be a dynamic entry or a static entry. The function generates a `IX_CC_ETH_TX_MSG_DEL_ARP_ENTRY` message and passes it to Ethernet TX core component. The message handler function of the Ethernet TX core component invokes the corresponding library API—that is, `ix_cc_eth_tx_del_arp_entry()`—to process the message.

C Syntax

```
ix_error ix_cc_eth_tx_async_del_arp_entry(
    ix_uint32 arg_L2Index,
    ix_cc_eth_tx_cb_del_arp_entry arg_Callback,
    void *arg_pContext);
```

Input

<code>arg_L2Index</code>	The L2 index used to search for the entry to be removed.
<code>arg_Callback</code>	A pointer to the callback routine with the function prototype, <code>ix_cc_eth_tx_cb_del_arp_entry()</code> .
<code>arg_pContext</code>	The pointer to a user-provided context structure used by the core component in the callback function identifying the origin of the request.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. A valid <code>ix_error</code>—the operation failed.
--------------	---

9.3.5.1 `ix_cc_eth_tx_cb_del_arp_entry()`

The function prototype for the callback function used to notify the calling application of the results of a call to `ix_cc_eth_tx_async_del_arp_entry()`.

C Syntax

```
ix_error (*ix_cc_eth_tx_cb_del_arp_entry)(
    ix_error arg_Result,
    void *arg_pContext);
```

Input

<code>arg_Result</code>	The <code>ix_error</code> returned by the Ethernet TX core component for the result of the function call. It is either <code>IX_SUCCESS</code> for success or a valid <code>ix_error</code> type for failure. See also <code>ix_cc_eth_tx_del_arp_entry()</code> for the defined error codes.
<code>arg_pContext</code>	A pointer to the user-provided context.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. A valid <code>ix_error</code>—the operation failed.
--------------	---

9.3.6 `ix_cc_eth_tx_async_purge_arp_cache()`

This is the messaging API function for purging the ARP cache. The function generates an `IX_CC_ETH_TX_MSG_PURGE_ARP_CACHE` message and passes it to Ethernet TX core component. The message handler function of the Ethernet TX core component invokes the corresponding library API—that is, `ix_cc_eth_tx_purge_arp_cache()`—to process the message.

C Syntax

```
ix_error ix_cc_eth_tx_async_purge_arp_cache();
```

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. A valid <code>ix_error</code>—the operation failed.
--------------	---

9.3.7 `ix_cc_eth_tx_async_dump_arp_cache()`

Implements the messaging function for dumping the ARP cache. The function generates a `IX_CC_ETH_TX_MSG_DUMP_ARP_CACHE` message and passes it to Ethernet TX core component. The message handler function of Ethernet TX core component invokes the corresponding library API—`ix_cc_eth_tx_dump_arp_cache()`—to process the request.

C Syntax

```
ix_error ix_cc_eth_tx_async_dump_arp_cache();
```

Output/Returns

Return Value	Returns a valid <code>ix_error</code> .
	<ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. A valid <code>ix_error</code>—the operation failed.

9.4 Library API

Table 9-4 lists the Library API functions provided by the Ethernet TX core component.

Table 9-4. Ethernet TX Library API

API	Description
<code>ix_cc_eth_tx_get_statistics_info()</code>	Returns statistics information.
<code>ix_cc_eth_tx_get_interface_state()</code>	Returns interface state information.
<code>ix_cc_eth_tx_create_arp_entry()</code>	Creates or updates a dynamic ARP entry.
<code>ix_cc_eth_tx_add_arp_entry()</code>	Adds a static ARP entry.
<code>ix_cc_eth_tx_del_arp_entry()</code>	Deletes an ARP entry.
<code>ix_cc_eth_tx_purge_arp_cache()</code>	Clears the ARP cache.
<code>ix_cc_eth_tx_dump_arp_cache()</code>	Dumps the ARP cache to standard output device.
<code>ix_cc_eth_tx_set_property()</code>	Sets or changes a dynamic property.

9.4.1 `ix_cc_eth_tx_get_statistics_info()`

Returns the requested statistics information.

C Syntax

```
ix_error ix_cc_eth_tx_get_statistics_info(
    ix_cc_eth_tx_statistics_info arg_entity,
    ix_uint32 arg_index,
    ix_cc_statistics_info_data *arg_pBuffer,
    void *arg_pContext
);
```

Input

<code>arg_entity</code>	The type of statistics information requested. See also the messaging API, ix_cc_eth_tx_async_get_statistics_info() in Section 9.3.1 for the types of statistics information supported.
<code>arg_index</code>	An index number representing the port for which statistics information is requested. See the messaging API, ix_cc_eth_tx_async_get_statistics_info() in Section 9.3.1 for details.
<code>arg_pBuffer</code>	A pointer to an <code>ix_cc_statistics_info_data</code> structure which stores the requested entity. See ix_s_cc_statistics_info_data in Section 9.4.1.1.
<code>arg_pContext</code>	A pointer to the component control block memory allocated earlier in the core component initialization function.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. A valid <code>ix_error</code>—the operation failed.
--------------	---

9.4.1.1 [ix_s_cc_statistics_info_data](#)

A pointer to an `ix_cc_statistics_info_data` structure which stores the requested entity.

C Syntax

```
typedef struct ix_s_cc_statistics_info_data {
    ix_uint32 dataLength;
    void *pDataBuffer;
} ix_cc_statistics_info_data;
```

Data Elements

<code>dataLength</code>	The length in bytes of the data buffer. <ul style="list-style-type: none"> For individual statistics attribute, the data length needs to be eight bytes—64-bits. For <code>IX_CC_ETH_TX_ALL_COUNTERS</code> type, the data length needs to be $(8 * 25) = 200$ bytes.
<code>pDataBuffer</code>	A pointer to a buffer where the statistics data are to be stored. It is the calling application's responsibility to allocate the required memory and assign it to the <code>pDataBuffer</code> pointer.

9.4.2 ix_cc_eth_tx_get_interface_state()

This function implements the library API for returning the interface state information. It is called internally by Ethernet Tx CC in its message handler routine for processing IX_CC_ETH_TX_MSG_GET_INTERFACE_STATE message. It may also be called by a level-0 based application for getting the interface status.

C Syntax

```
ix_error ix_cc_eth_tx_get_interface_state(
    ix_uint8 arg_Port,
    ix_cc_eth_tx_if_state *arg_pIfState,
    void *arg_pContext);
```

Input

arg_Port	The interface port number whose state is to be returned.
arg_pIfState	A pointer to the memory location used to return the interface state—see also the messaging API, ix_cc_eth_tx_if_state_context for the data type description. The calling application must allocate this memory. The core component passes the state back using the memory referenced by this pointer.
arg_pContext	A pointer to the component control block memory allocated earlier in the core component initialization function.

Output/Returns

Return Value	Returns a valid ix_error. <ul style="list-style-type: none"> IX_SUCCESS—the operation succeeded. A valid ix_error—the operation failed.
--------------	---

9.4.3 ix_cc_eth_tx_create_arp_entry()

This function implements the library API for creating or updating a dynamic ARP entry. It is called internally by the Ethernet TX core component in its message handler routine for processing the IX_CC_ETH_TX_MSG_CREATE_ARP_ENTRY message.

When a new L2 index is created by Next Hop ID Manager of CP PDK, the Ethernet Tx core component is informed by the Forwarding Plane module to create or update an entry in the ARP cache—performed by calling this operation. This function calls its ARP module's [ix_cc_arp_create_entry\(\)](#) to accomplish the operation. If an entry already exists with the same next hop IP, the ARP entry and corresponding entry in the L2 table are updated with the L2 index and L2 header information. If the entry does not exist, a new entry is created in the ARP cache using the specified L2 index, next hop IP, output port ID and an ARP request packet for the next hop IP is sent to the Queue Manager for transmitting out to the Ethernet media.

C Syntax

```
ix_error ix_cc_eth_tx_create_arp_entry (
    ix_cc_eth_tx_next_hop_info *arg_pArpInfo,
    void *arg_pContext);
```

Input

<code>arg_pArpInfo</code>	The ARP information data structure containing the ARP information used to create or update the entry. The function assumes the L2 index, next hop IP, and port ID are valid. The <code>l2Addr</code> field is not used and will be ignored by the function.
<code>arg_pContext</code>	A pointer to the component control block memory allocated earlier in the core component initialization function.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. <code>IX_ERROR_CC_ARP_ERROR</code>—an error was countered in an underlying component.
--------------	---

9.4.4 `ix_cc_eth_tx_add_arp_entry()`

This function implements the Library API for adding a static ARP entry. It is called internally by the Ethernet TX core component in its message handler routine to process `IX_CC_ETH_TX_MSG_ADD_ARP_ENTRY` messages.

The function invokes the ARP module's `ix_cc_arp_add_entry()` to accomplish the operation. Based on the L2 index, the corresponding entry in the L2 table is also synchronized with the next hop IP and L2 header information.

C Syntax

```
ix_error ix_cc_eth_tx_add_arp_entry(
    ix_cc_eth_tx_next_hop_info *arg_pArpInfo,
    void *arg_pContext);
```

Input

<code>arg_pArpInfo</code>	The ARP data structure containing the ARP information used to create the entry. The function assumes all fields in the structure are valid.
<code>arg_pContext</code>	A pointer to the component control block memory allocated earlier in the core component initialization function.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • A valid <code>ix_error</code>—the operation failed.
--------------	---

9.4.5 `ix_cc_eth_tx_del_arp_entry()`

Implements the library API for deleting an ARP entry. The entry can be a dynamic entry or a static entry. It is called internally by the Ethernet TX core component in its message handler routine for processing the `IX_CC_ETH_TX_MSG_DEL_ARP_ENTRY` message.

This function invokes the ARP module's `ix_cc_arp_del_entry()` to accomplish the operation. Based on the L2 index, the corresponding entry in the L2 table will also be marked as an invalid entry.

C Syntax

```
ix_error ix_cc_eth_tx_del_arp_entry(
    ix_uint32 arg_L2Index,
    void *arg_pContext);
```

Input

<code>arg_L2Index</code>	The L2 index to be used to search for the entry to be removed.
<code>arg_pContext</code>	A pointer to the component control block memory allocated earlier in the core component initialization function.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • A valid <code>ix_error</code>—the operation failed.
--------------	---

9.4.6 `ix_cc_eth_tx_purge_arp_cache()`

Implements the Library API for purging the ARP cache. It is called internally by the Ethernet TX core component in its message handler routine for processing the `IX_CC_ETH_TX_MSG_PURGE_ARP_CACHE` message.

This function invokes the ARP module's `ix_cc_arp_purge()` function to accomplish the operation. Once cleared, all ARP entries including dynamic and static entries are removed. All corresponding entries in the L2 table are also marked as invalid entries.

C Syntax

```
ix_error ix_cc_eth_tx_purge_arp_cache(void *arg_pContext);
```

Input

<code>arg_pContext</code>	A pointer to the component control block memory allocated earlier in the core component initialization function.
---------------------------	--

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • A valid <code>ix_error</code>—the operation failed.
--------------	---

9.4.7 `ix_cc_eth_tx_dump_arp_cache()`

Implements the Library API for printing out the content of the ARP cache to the VxWorks^{*} or Linux^{*} standard output. The function is called internally by the Ethernet TX core component in its message handler routine for processing the `IX_CC_ETH_TX_MSG_DUMP_ARP_CACHE` message.

This function invokes the ARP module's `ix_cc_arp_dump()` function to implement the operation. This function is used primarily for debugging purposes.

C Syntax

```
ix_error ix_cc_eth_tx_dump_arp_cache(void *arg_pContext);
```

Input

<code>arg_pContext</code>	A pointer to the component control block memory allocated earlier in the core component initialization function.
---------------------------	--

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • A valid <code>ix_error</code>—the operation failed.
--------------	---

9.4.8 `ix_cc_eth_tx_set_property()`

Implements the Library API for setting one or more dynamic properties provided by the Ethernet TX core component. The function is called either by the Ethernet TX core component internally in its port message handler routine or by a portability framework based application.

For the IP address property, more than one IP address can be added to a port. When adding an IP address that already exists in the Local Interface table, no change is made and the function returns `IX_SUCCESS`.

For the MAC address property, an interface port can have only one unique MAC address assigned to it and is not allowed to have more than one layer-2 address. Once set, the user can still call this function to change the MAC address. A gratuitous ARP packet is generated and sent out to the Ethernet media automatically by the Ethernet TX core component the first time the address is set and any time the address is subsequently changed. Internally, the Ethernet TX core component uses the ARP module to generate the gratuitous ARP packet.

C Syntax

```
ix_error ix_cc_eth_tx_set_property(
    ix_uint32 arg_PropId,
    ix_cc_properties *arg_pProperty,
    void *arg_pContext);
```

Input

<code>arg_PropId</code>	<p>The property to be changed.</p> <p>For Ethernet TX, these properties include:</p> <ul style="list-style-type: none"> <code>IX_CC_SET_PROPERTY_ADD_INTERFACE_IPV4</code>—adds a port IPv4 address to a port. <code>IX_CC_SET_PROPERTY_DEL_INTERFACE_IPV4</code>—deletes an IPv4 address from a port. <code>IX_CC_SET_PROPERTY_MAC_ADDR</code>—sets the MAC address of a port. <code>IX_CC_SET_PROPERTY_PHYSICAL_IF_STATUS</code>—the state of the interface port; either up or down. <p>The user can set multiple properties in a single function call by taking the logical <code>or</code> of some or all these property IDs and setting up corresponding fields in the <code>ix_cc_properties</code> structure.</p>
<code>arg_pProperty</code>	A pointer to the <code>ix_cc_properties</code> structure which contains the valid port number of the port whose property is to be set as well as the associated properties—either an IP address, a MAC address, or both.
<code>arg_pContext</code>	A pointer to the component control block memory allocated earlier in the core component initialization function.

Output/Returns

Return Value	<p>Returns a valid <code>ix_error</code>.</p> <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. A valid <code>ix_error</code>—the operation failed.
--------------	---

The Ethernet ARP module is a library within the Ethernet TX core component. It provides ARP functionality to the Ethernet TX and other core components.

The Ethernet ARP module provides the following services specified in RFC 826, 1122, and 3220:

- Processing of ARP packets, see [Section 10.2.10, “ix_cc_arp_process_arp_pkts\(\).”](#)
- Layer 2 address resolution, see [Section 10.2.9, “ix_cc_arp_resolve_l2_addr\(\).”](#)
- Creation of dynamic ARP entries, see [Section 10.2.3, “ix_cc_arp_create_entry\(\).”](#)
- ARP entry aging
- ARP flooding prevention
- ARP packet queue

A set of additional rules is also implemented on top of these basic rules for the purpose of compatibility and enhancement. This includes auto-detection of duplication of IP address and layer-2 address update for gratuitous ARP, as specified in RFC3220.

For fulfilling requirements from RFC 1122, the Ethernet ARP module has:

- an ARP entry aging timer
- ARP flooding prevention
- an ARP packet queue.

The ARP aging timer flushes out-of-date cache entries. All dynamic entries of the ARP cache are aged automatically using the same fixed period timer (default 20 minutes). Any dynamic entry that is inactive between two consecutive timer events is deleted from the ARP cache.

The Ethernet ARP module also has a mechanism to prevent sending ARP requests for the same IP address at a high rate. The maximum rate is 1 ARP request per second per destination.

The ARP module holds the latest packet of each set of packets destined to the same unresolved IP address, and transmits the held packet when the address has been resolved.

In addition to the above features, the Ethernet ARP module also provides the following services for clients to use:

- Creation of static ARP entries
- Deletion of ARP entries
- ARP cache purging
- Printing ARP cache

For complete details see [Chapter 47, “Ethernet ARP Module”](#) of the *Intel® Internet Exchange Architecture Software Building Blocks Developer's Manual*.

10.1 Error Types

The Ethernet ARP module defines the following error types for its external functions.

C Syntax

```
enum ix_cc_arp_error {
    IX_CC_ARP_ERROR_FIRST = IX_ERROR_MAKE_GROUP(CC_ARP),
        IX_CC_ARP_ERROR_NEW_ENTRY_CREATED,
        IX_CC_ARP_ERROR_INVALID_L2_INDEX,
        IX_CC_ARP_ERROR_INCORRECT_HEADER,
        IX_CC_ARP_ERROR_INVALID_PARAMETER,
        IX_CC_ARP_INVALID_HANDLE,
        IX_CC_ARP_ERROR_NHIP_NOT_FOUND,
        IX_CC_ARP_ERROR_NO_CACHE_ENTRY,
        IX_CC_ARP_ERROR_SIP_CONFLICT,
        IX_CC_ARP_ERROR_NO_LOCAL_MAC_ADDRESS,
        IX_CC_ARP_ERROR_ENTRY_EXIST,
        IX_CC_ARP_FAIL_UPDATE_L2TM,
        IX_CC_ARP_ERROR_EVENT_HANDLER_GET_TIME,
        IX_CC_ARP_ERROR_CCI,
        IX_CC_ARP_ERROR_LAST,
};
```

Supported ARP Error Codes

```
IX_CC_ARP_ERROR_INVALID_L2_INDEX

IX_CC_ARP_ERROR_INCORRECT_HEADER

IX_CC_ARP_ERROR_INVALID_PARAMETER

IX_CC_ARP_INVALID_HANDLE

IX_CC_ARP_ERROR_NHIP_NOT_FOUND

IX_CC_ARP_ERROR_NO_CACHE_ENTRY

IX_CC_ARP_ERROR_SIP_CONFLICT

IX_CC_ARP_ERROR_NO_LOCAL_MAC_ADDRESS

IX_CC_ARP_ERROR_ENTRY_EXIST

IX_CC_ARP_FAIL_UPDATE_L2TM

IX_CC_ARP_ERROR_EVENT_HANDLER_GET_TIME

IX_CC_ARP_ERROR_CCI

IX_CC_ARP_ERROR_LAST
```

10.2 Library API

Table 10-1 shows the Ethernet ARP library external functions for the Ethernet TX core component.

Table 10-1. Ethernet ARP Library API

Function Name	Description
§ ix_cc_arp_init()	Initializes the ARP library.
§ ix_cc_arp_fini()	Terminates the ARP library.
§ ix_cc_arp_create_entry()	Creates or updates a dynamic ARP entry.
§ ix_cc_arp_add_entry()	Adds a static ARP entry.
ix_cc_arp_update_entry()	Updates existing entry in the ARP based on the Next Hop IP address
§ ix_cc_arp_del_entry()	Deletes an ARP entry.
§ ix_cc_arp_purge()	Clears the ARP cache.
§ ix_cc_arp_dump()	Dumps the ARP cache to standard output device.
§ ix_cc_arp_resolve_l2_addr()	Resolves a layer-2 address for outgoing exception IP packets.
§ ix_cc_arp_process_arp_pkts()	Handles incoming ARP packets.
§ ix_cc_arp_create_gratuitous_arp()	Generates a gratuitous ARP packet.

10.2.1 ix_cc_arp_init()

This function initializes the ARP library. It should be called and returned before any other function in this library can be called. The function performs the following:

- Creates and initializes the ARP cache.
- Registers the ARP timer for ARP cache entry time-out.

Entries in the ARP cache are both dynamically and statically added and deleted at run time. The size of the ARP cache depends on the number of hosts connected to each port of the output blade.

The function is called by the Ethernet TX core component in its init function.

C Syntax

```
ix_error ix_cc_arp_init(ix_arp_init_context* arg_pInitContext);
```

Input

```
ix_arp_init_context*  
    arg_pInitContext
```

pointer to init context structure. Contains free list handle for allocating hardware buffers, function pointers for looking up information in local interface table and function pointer for sending out ARP packets.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> .
	<ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • A valid <code>ix_error</code>—the operation failed.

10.2.2 `ix_cc_arp_fini()`

The function terminates the ARP library. It frees the ARP cache, related memory, resources allocated by `ix_cc_arp_init()` function, and other functions of this library. The function is called by the Ethernet TX core component in its fini function.

C Syntax

```
ix_error ix_cc_arp_fini (void);
```

Input

none

Output/Returns

Return Value	Returns a valid <code>ix_error</code> .
	<ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • A valid <code>ix_error</code>—the operation failed.

10.2.3 `ix_cc_arp_create_entry()`

This function creates or updates an entry in the ARP cache. When a new next hop ID is created by Next Hop ID Manager of CP PDK, the ARP module is informed by the Forwarding Plane module through the Ethernet TX core component's exposed API to create or update entry in the ARP cache

The function first performs a look-up in the ARP cache based on the next hop IP. If the entry is found, it updates the ARP cache with L2 index. based on the L2 index, it updates entry in L2 table with L2 header info from the ARP cache and set the entry to valid. If the entry is not found in ARP cache the function creates a new entry in the ARP cache using the L2 index, next hop IP, and output port ID. In addition, it generates an ARP request packet using the next hop IP and returns the packet to the calling application for transmission.

C Syntax

```
ix_error ix_cc_arp_create_entry (
    ix_cc_arp_next_hop_info* arg_pArpInfo
    ix_buffer_handle* arg_pRtnPkt);
```

Input

<code>arg_pArpInfo</code>	The ARP information structure containing the ARP information to be used to create or update the entry. The L2 index must be a valid number—not -1—and the function assumes the next hop IP and output port are valid. The <code>l2Addr</code> field will be ignored by the function.
---------------------------	--

Output/Returns

<code>arg_pRtnPkt</code>	If the function returns <code>IX_ERROR_CC_ARP_NEW_ENTRY_CREATED</code> , the argument contains a buffer handle—pointing to a complete ARP request packet including meta-data and a 14-byte Ethernet header—generated by the ARP module. The next hop ID field in the meta-data of the packet is set to -1 to indicate it is a non-IP packet for the microblock to identify. It is the calling application's responsibility to transmit this ARP packet out to the Ethernet media.
--------------------------	---

Return Value	<p>Returns a valid <code>ix_error</code>.</p> <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded, if entry found and updated. • A valid <code>ix_error</code>—the operation failed. • <code>IX_CC_ARP_ERROR_NEW_ENTRY_CREATED</code>—the operation failed, the entry is not found and a new entry is created. An ARP request packet is returned. • <code>IX_CC_ARP_ERROR_INVALID_L2_INDEX</code>—the operation failed due to an invalid L2 index • <code>IX_CC_ERROR_INTERNAL</code>—the operation failed, due to an error encountered from an underlying component.
--------------	---

10.2.4 `ix_cc_arp_add_entry()`

The ARP entries usually are created and timed out dynamically during run time as the result of the network traffic exchange. However, sometimes it is desirable to create permanent entries. This function can add a static entry to the ARP table. Once added, the static entry is permanent and is not timed out by the ARP module. Based on the next hop ID, the corresponding entry in the L2 table is also synchronized with the next hop IP and L2 header information.

The calling application must not use this function to create a dynamic ARP entry. For that, the calling application should call `ix_cc_arp_create_entry()`.

C Syntax

```
ix_error ix_cc_arp_add_entry (
    ix_cc_arp_next_hop_info *arg_pArpInfo);
```

Input

`arg_pArpInfo` The pointer to the ARP information structure containing the ARP information to be added. The next hop identifier must be a valid value—not -1—and the function assumes that all other fields in the structure are valid.

:

Output/Returns

Return Value Returns a valid `ix_error`.

- `IX_SUCCESS`—the operation succeeded, entry has been added
- `IX_CC_ARP_ERROR_INVALID_L2_INDEX`—Invalid L2 index
- `IX_CC_ERROR_ARP_ENTRY_EXIST`—the operation failed, an entry with the same next hop ID already exists in the ARP cache. Information in that entry was not changed.
- `IX_CC_ERROR_INTERNAL`—the operation failed, due to an error encountered in an underlying component.

10.2.5 `ix_cc_arp_update_entry()`

This function updates the existing entry in the ARP based on the Next Hop IP address. The entry can be a dynamic entry or a static entry created by either `ix_cc_arp_create_entry()` or `ix_cc_arp_add_entry()`. It also updates the corresponding L2 Table entry and sets it to a valid state. The function returns an error if ARP cache entry does not exist—it does not add new entry.

C Syntax

```
ix_error ix_cc_arp_update_entry (ix_cc_arp_next_hop_info* arg_pArpInfo);
```

Input

`arg_pArpInfo` A pointer to the ARP information structure which contains the ARP info to be added. The next hop identifier must be a valid number (not -1) and the function will assume all other fields in the structure are valid.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—operation succeeded. an entry has been added successfully <code>IX_CC_ARP_ERROR_INVALID_L2_INDEX</code>—operation failed, Invalid L2 Index <code>IX_CC_ERROR_ENTRY_NOT_FOUND</code>—operation failed, an entry is not found in the ARP cache, client need to use <code>ix_cc_arp_add_entry()</code> to add new ARP cache entry. <code>IX_CC_ERROR_INTERNAL</code>—operation failed, an error is encountered from underlying component.
--------------	---

10.2.6 `ix_cc_arp_del_entry()`

This function removes an entry from the ARP cache based on the L2 Index. The entry can be a dynamic entry or a static entry created by either `ix_cc_arp_create_entry()` or `ix_cc_arp_add_entry()` The corresponding entry in the L2 table is marked as an invalid entry.

C Syntax

```
ix_error ix_cc_arp_del_entry (ix_uint32 arg_L2Index);
```

Input

<code>arg_L2Index</code>	The L2 Index—used to search for the entry to be removed
:	

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded, entry has been removed <code>IX_CC_ARP_ERROR_ENTRY_NOT_FOUND</code>—the operation failed, the entry was not found. <code>IX_CC_ARP_ERROR_INVALID_L2_INDEX</code>—the operation failed due to an invalid L2 index
--------------	---

10.2.7 `ix_cc_arp_purge()`

The function clears the ARP cache. Once cleared, all ARP entries including dynamic and static entries are removed.

C Syntax

```
ix_error ix_cc_arp_purge (void);
```

:

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeed, the ARP cache has been purged <code>IX_CC_ERROR_INTERNAL</code>—the operation failed, the operation cannot be handled because of some underlying component errors
--------------	---

10.2.8 `ix_cc_arp_dump()`

This function prints the content of the ARP cache to VxWorks or Linux standard output. This includes information of all valid static and dynamic entries in the ARP cache. This function is used mainly for debugging purposes.

C Syntax

```
ix_error ix_cc_arp_dump (void);
```

Input

none

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. A valid <code>ix_error</code>—the operation failed.
--------------	---

10.2.9 `ix_cc_arp_resolve_l2_addr()`

The function is used solely by Ethernet TX core component internally for the resolution of L2 address when receiving an exception IP packet from Ethernet TX Microblock due to unsolved L2 information in the L2 table. This event (unsolved L2 information in the L2 table) prompts the Ethernet TX core component to call the function `ix_cc_arp_resolve_l2_addr()`. Therefore, there is no need and not an associated message defined by Ethernet TX core component for other outside components or system application to access this function.

The function resolves the L2 information by either synchronizing the L2 table with L2 information from ARP cache or generates an ARP request packet and returns it to the calling application for solicitation of L2 information.

The function first looks up the next hop IP in the ARP cache, based on the next hop identifier embedded in the meta data of the packet. If the L2 header information of the matched ARP entry is valid, it is returned to the calling application through the output parameter and ARP information structure. In addition, the corresponding entry in L2 table is synchronized by ARP module with the L2 header information. The ownership of the input IP packet is then given back to the calling application for transmission.

If no entry is found, an ARP request packet is generated using the next hop IP address and returned to the calling application for transmission. The input IP packet is held by the ARP module and is sent to the Ethernet media later by the ARP module when the corresponding ARP reply is received.

C Syntax

```
ix_error ix_cc_arp_resolve_l2_addr (
    ix_buffer_handle arg_Pkt,
    ix_cc_arp_next_hop_info* arg_pArpInfo,
    ix_buffer_handle* arg_pRtnPkt);
```

Input

<code>arg_Pkt</code>	The buffer handle pointing to the IP packet with an un-resolved layer-2 address. The meta-data is assumed to be present at the beginning of the packet.
----------------------	---

Output/Returns

<code>arg_pArpInfo</code>	The pointer to the ARP info structure to be written with the L2 header info. This argument must be ignored if any error code is returned.
<code>arg_pRtnPkt</code>	<p>The pointer to a buffer handle which points to a complete ARP request packet—including meta-data and 14-byte Ethernet header—generated by the ARP module. The next hop ID field in the meta-data of the packet is set to -1—as a non-IP packet—for the microblock to identify. The protocol type field in the Ethernet header is set to 0x0806.</p> <p>It is the calling applications's responsibility to transmit this ARP packet out to Ethernet media.</p> <p>This argument is ignored if the function returns anything other than <code>IX_CC_ARP_ENTRY_NOT_FOUND</code>.</p>
Return Value	<p>Returns a valid <code>ix_error</code>.</p> <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeed, the look-up is successful. <code>arg_pArpInfo</code> contains the L2 header info and the input packet is given back to the calling application. • <code>IX_CC_ARP_ERROR_ENTRY_NOT_FOUND</code>—the operation failed, due to the look-up failure. An ARP request packet is created and returned and the input packet is held by the ARP module. • <code>IX_CC_ARP_ERROR_INVALID_L2_INDEX</code>—the operation failed, due to an invalid next hop identifier. The input packet was dropped.

10.2.10 `ix_cc_arp_process_arp_pkts()`

The function is used solely by Ethernet TX core component internally for handling ARP packets received from Ethernet RX core component as exception packets. This event prompts the Ethernet TX core component to call the function `ix_cc_arp_process_arp_pkts()`. Therefore, there is no need and not an associated message defined by Ethernet TX core component for other outside components or system application to access this function.

The function receives ARP packet from the calling application and performs appropriate operations conforming to RFC826, 1122, and 3220 based on the ARP packet type, target IP address, and sender's IP address. This includes the following operations:

- For ARP request packet destined to one of the blade interfaces, an ARP reply packet is generated and returned to the calling application for transmission to the Ethernet media. In addition, the sender's IP address and L2 information is learned and a new entry is added into the ARP cache (if one doesn't already exist).
- For ARP reply packet destined to one of the blade interfaces, the ARP cache and L2 table is updated with the L2 information. The ARP module returns any previously held IP packet to the calling application for transmission.
- For any ARP packet, if the sender's IP address equals one of the blade interface addresses, a message is logged by the ARP module for any configuration error.
- For any ARP packet, if an entry corresponding with sender's IP already exists in the ARP cache, the L2 information of that entry as well as the corresponding entry in the L2 table is updated accordingly to deal with gratuitous ARP packet [RFC 3220].

C Syntax

```
ix_error ix_cc_arp_process_arp_pkts(  
    ix_buffer_handle arg_ArpPkt  
    ix_buffer_handle *arg_pRtnPkt,  
    ix_uint32* arg_pPktType);
```

Input

<code>arg_ArpPkt</code>	The buffer handle pointing to the ARP packet to be handled. The meta-data is assumed to be present for the packet—assuming output port is valid. Once processed, the buffer may be re-used or may be freed back to system buffer pool by the ARP module. The calling application must not make any other assumption concerning the ownership of the buffer.
-------------------------	---

Output

<code>arg_pRtnPkt</code>	<p>The pointer to a buffer handle which points to the packet returned by the ARP module. The packet may be either an ARP reply packet or a previously held IP packet.</p> <p>For ARP reply packet, the ARP module includes the 14-byte Ethernet header at the beginning of the packet. The next hop ID field in the meta-data of the packet is set to -1 as a non-IP packet and the protocol type field in the Ethernet header is set to 0x0806.</p> <p>For a previously held IP packet, the ARP module updates only the meta-data.</p> <p>It is the calling application's responsibility to transmit this packet out to the network.</p> <p>This argument is ignored if the function returns an error code.</p>
<code>arg_pPktType</code>	<p>The type of packet returned to the calling application. The types could be <code>ARP_REQUEST</code>, <code>ARP_REPLY</code>, or <code>IP_PACKET</code>.</p>
Return Value	<p>Returns a valid <code>ix_error</code>.</p> <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeed, the ARP packet has been handled successfully. <code>IX_CC_ARP_ERROR_INCORRECT_HEADER</code>—the operation failed, the packet could not be handled because of missing or incorrect header information. <code>IX_CC_ERROR_INTERNAL</code>—the operation failed, the ARP packet could not be handled because of some underlying component errors.

Note: For any error case, the packet is dropped by the ARP module.

10.2.11 `ix_cc_arp_create_gratuitous_arp()`

The function is to be used to generate a gratuitous ARP packet. It is called solely by Ethernet TX core component. Any change of layer-2 address of an interface port prompts the Ethernet TX core component to call the function `ix_cc_arp_create_gratuitous_arp()`. Generally, the gratuitous ARP packet is sent out to the network media when the system reboots or layer-2 address of the network port is changed.

C Syntax

```
ix_error ix_cc_arp_create_gratuitous_arp (
    ix_uint16 arg_OutputPort,
    ix_buffer_handle* arg_pRtnPkt);
```

Input

<code>arg_OutputPort</code>	The port number that has changes of properties. Gratuitous ARP packet is sent through this port.
-----------------------------	--

Output/Returns

`arg_pRtnPkt`

The pointer to a buffer handle which points to a complete gratuitous ARP packet—including meta-data and 14-byte Ethernet header—generated by the ARP module. The next hop ID field in the meta-data of the packet will be set to -1—as a non-IP packet—for the microblock to identify. The protocol type field in the Ethernet header will be set to 0x0806.

It is the calling application's responsibility to transmit this packet out to the Ethernet media.

This argument is ignored when the function returns anything other than `IX_SUCCESS`.

Return Value

Returns a valid `ix_error`.

- `IX_SUCCESS`—the operation succeed, the gratuitous ARP packet has been created successfully.
- `IX_CC_ARP_ERROR_INCORRECT_HEADER`—the operation failed, the packet could not be handled because of missing or incorrect header information.
- `IX_CC_ERROR_INTERNAL`—the operation failed, the request could not be handled because of underlying component errors.

Queue Manager

The following chapters are included in this section:

- [Chapter 11, “Queue Manager”](#)

Queue Manager Core Component – There are two Queue Manager Core Components that exist in the system – Ingress Queue Manager Core Component and Egress Queue Manager Core Component. Queue Manager Core Component provides the following functionality in IXA SDK 3.1, common to both Ingress and Egress Queue Manager:

- configuration module for Queue Manager microblock
- centralized packet en-queuing from core to the microblocks
- handling of packets en-queued for local output ports in case the switch fabric does not support loop back

- [Chapter 12, “Egress Queue Manager \(DiffServ\)”](#)

The Egress Queue Manager core component for DiffServ is identical to the Queue Manager (QM) core component detailed in [Chapter 11, “Queue Manager”](#) of this document.

The difference in the Egress Queue manager (Diffserv) from the QM core component is the following:

- The queue descriptor size is 8 LW. The core component must take this into account while allocating SRAM memory for the queue descriptor table.

During initialization, the core component inserts into the system repository

The Queue Manager core component performs the following functions:

- configuration for Queue Manager microblock
- centralized packet enqueueing from all core components to the microblocks
- handling of packets enqueued for local output ports in case the switch fabric does not support loopback.

There are generally two Queue Manager core components in an application -- one for ingress and one for egress. In most of the cases, the Queue Manager core component on ingress side receives packets and enqueues them to be sent to the CSIX switch fabric media. The Queue Manager on the egress side enqueues packets to be sent to the ATM, POS, or Ethernet media. The differences between these two are mentioned explicitly in the remainder of this chapter.

For complete details, see [Chapter 48, “Queue Manager Core Component”](#) of the *Intel® Internet Exchange Architecture (IXA) Software Building Blocks Developer's Manual*.

11.1 Core Component Infrastructure API

This section describes the Queue Manager core component Infrastructure interface and the message identifiers used by the Queue Manager. [Table 11-1](#) summarizes the Queue Manager core component Infrastructure API.

Table 11-1. Queue Manager Core Component Infrastructure API

Name	Description
<code>ix_cc_qm_init()</code>	Initializes a Queue Manager core component.
<code>ix_cc_qm_fini()</code>	Terminates a Queue Manager core component.
<code>ix_cc_qm_pkt_handler()</code>	Receives the packets sent to the Queue Manager by other core components.
<code>ix_cc_qm_msg_handler()</code>	Processes messages of type <code>IX_CC_QM_MSG_GETPKTCOUNT</code> .

11.1.1 `ix_cc_qm_init()`

Initializes the queue manager core component. It is called as result of the `ix_cci_cc_create()` Core Component Infrastructure call from the execution engine and must return successfully before any other function in the core component is called. This function performs static configuration for the microblock by allocating and patching required microblock variables and memory blocks using the Resource Manager API.

C Syntax

```
ix_error ix_cc_qm_init (
    ix_cc_handle arg_hCcHandle,
    void **arg_ppContext);
```

Input

`arg_cc_handle` A handle to the core component.

Output/Returns

`arg_ppContext` A pointer to the location where the pointer to the Queue Manager core component internal context is stored. This internal context pointer is allocated by the core component. It contains Queue Manager core component configuration parameters and internal data structures. This pointer is later passed into the `ix_cc_qm_fini()` function by the Core Component Infrastructure and is used to free memory when the core component is destroyed.

Return Value Returns a valid `ix_error`.

- `IX_SUCCESS`—the operation succeeded.
- A valid `ix_error`—the operation failed

11.1.2 `ix_cc_qm_fini()`

Terminates the Queue Manager core component. It is invoked when the execution engine running the core component is shut down. This function frees all allocated memory and resources obtained in the `ix_cc_qm_init()` function.

C Syntax

```
ix_error ix_cc_qm_fini(
    ix_cc_handle arg_hCcHandle,
    void *arg_pContext);
```

Input

`arg_cc_handle` A handle to the core component.

`arg_pContext` A pointer to the control block memory allocated earlier in `ix_cc_qm_fini()`. The termination routine uses it to de-allocate the control block memory.

Output/Returns

Return Value Returns a valid `ix_error`.

- `IX_SUCCESS`—the operation succeeded.
- A valid `ix_error`—the operation failed

11.1.3 ix_cc_qm_pkt_handler()

Receives the packets sent to the queue manager component by other core components. This handler can be associated with several output communication IDs of the IPv4 Forwarder core component, interface TX core components or any other core component. The input to this component is defined in the system's `bindings.h` file as `IX_CC_QM_COMMON_PKT_INPUT` as well as `IX_CC_QM_EGRESS_PKT_INPUT`.

C Syntax

```
ix_error ix_cc_qm_pkt_handler(
    ix_buffer_handle arg_hDataToken,
    ix_uint32 arg_ExceptionCode,
    void *arg_pComponentContext);
```

Input

<code>arg_hDataToken</code>	A handle for the packet buffer.
<code>arg_ExceptionCode</code>	The exception code.
<code>arg_pComponentContext</code>	A pointer to a Queue Manager core component context which is passed to the core component when a packet arrives. This context is defined by the core component and passed to the framework through the <code>ix_cci_cc_add_packet_handler()</code> function called by the initialization function of the core component.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. A valid <code>ix_error</code>—the operation failed
--------------	--

11.1.4 Sending Packets

The Queue Manager core component sends packets to several outputs:

- Microblock Input Targets
- PCI Packet Input Target

The Queue Manager component sends a packet to a microblock by invoking the Core Component Infrastructure interface passing the microblock communication ID. The communication ID for outgoing packets is defined in the System Application file `bindings.h` with `IX_CC_QM_OUTPUT` as in the following line of code:

```
#define IX_CC_QM_UBLOCK_PKT_OUTPUT IX_UBLOCK_PKT_INPUT
```

For PCI the application must also map `IX_CC_EGRESS_PKT_OUTPUT` to the Egress Queue Manager's input ID, `IX_CC_QM_EGRESS_PKT_INPUT`, as in the following line of code:

C Syntax

```
#define IX_CC_EGRESS_PKT_OUTPUT IX_RM_COMM_MAKE_IN
      (IX_CC_QM_EGRESS_PKT_INPUT, IX_PEER_SUBSYSTEM, 0)
```

The Queue Manager component uses the Resource Manager function, `ix_rm_send_packet()`—see *Intel® Internet Exchange Architecture Portability Framework Reference Manual*, to send a packet to the destination microblock as in the following line of code:

```
ix_rm_send_packet(IX_CC_QM_UBLOCK_PKT_OUTPUT, 0, argPkt)
```

The first argument is the communication ID identifying the packet destination and the second argument is an `ix_buffer_handle` identifying the packet to send.

11.1.5 ix_cc_qm_msg_handler()

This message handler function processes messages of type `IX_CC_QM_MSG_GETPKTCOUNT`. The message is created to pass data into the core component using the messaging API as described in [Section 11.2, “Messaging API.”](#) This function is installed by the queue manager’s `ix_cc_qm_init()` function. The communication ID for this handler is defined in the System Application file `bindings.h` as `IX_QM_MSG_INPUT` as seen in the following line of code:

```
#define IX_QM_MSG_INPUT /*Maps Queue Manager component Message Input to*/
```

C Syntax

```
ix_error ix_cc_qm_msg_handler(
    ix_buffer_handle arg_Msg,
    ix_uint32 arg_MsgId,
    void *arg_pCtx)
```

Input

<code>arg_Msg</code>	A handle for the packet buffer containing the message to process.
<code>arg_MsgId</code>	The message identifier which, for this function, should be <code>IX_CC_QM_MSG_GETPKTCOUNT</code> .
<code>arg_pCtx</code>	A pointer to a context which is passed to the core component when a message arrives. This context is defined by the Queue Manager and passed to the framework during initialization of the core component.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. A valid <code>ix_error</code>—the operation failed
--------------	--

11.2 Messaging API

The Messaging API exposed by the Queue Manager reports the total number of packets processed by the Queue Manager core component. This API is used for debugging purposes.

Table 11-2 summarizes the Queue Manager core component Messaging API.

Table 11-2. Queue Manager Core Component Messaging API

Name	Description
<code>ix_cc_qm_async_get_packet_count()</code>	Returns the number of packets processed by the Queue Manager core component.

11.2.1 `ix_cc_qm_async_get_packet_count()`

Returns the number of packets processed by the Queue Manager core component.

C Syntax

```
ix_error ix_cc_qm_async_get_packet_count (
    ix_cc_qm_cb_pkt_count arg_Callback,
    void* arg_pContext)
```

Input

<code>arg_Callback</code>	The function pointer to the callback routine. See ix_cc_qm_cb_pkt_count .
<code>arg_pContext</code>	Specifies a calling application-defined context.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. A valid <code>ix_error</code>—the operation failed
--------------	--

11.2.1.1 `ix_cc_qm_cb_pkt_count`

The function prototype of a function pointer to the callback routine for calls to `ix_cc_qm_async_get_packet_count()`.

C Syntax

```
ix_error (*ix_cc_qm_cb_pkt_count)(
    ix_error arg_Result,
    void* arg_pContext
    ix_uint64 arg_pPktCount);
```

Input

<code>arg_Result</code>	Signals success, the value <code>IX_SUCCESS</code> , or an error condition for the call, signaled by a valid <code>ix_error</code> code.
<code>arg_pContext</code>	Pointer to the calling application-provided context passed in by the API call. This pointer is used by the calling application to identify the originator of the request.
<code>arg_pPktCount</code>	Pointer provided by the calling application specifying the location for the packet counter data—the number of packets.

11.3 Library API

The Library API exposed by the Queue Manager reports the total number of packets processed by the Queue Manager core component. This API is used for debugging purposes.

Table 11-3 shows a summary of the Queue Manager core component Library API.

Table 11-3. Queue Manager Core Component Library API

Name	Description
<code>ix_cc_qm_get_packet_count()</code>	Reports the number of packets processed by a Queue Manager core component.

11.3.1 `ix_cc_qm_get_packet_count()`

This function reports the number of packets processed by a Queue Manager core component. For example, use a call to this operation to report the number of packets processed by the Ingress Queue Manager core component.

Note: This function reports only packets processed by the core component and not those processed by the the Queue Manager microblocks.

C Syntax

```
ix_error ix_cc_qm_get_packet_count (ix_uint64* arg_pPktCount)
```

Output/Returns

<code>arg_pPktCount</code>	A pointer to the 64-bit counter. On return, this counter contains the number of packets processed by the Queue Manager core component.
Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. A valid <code>ix_error</code>—the operation failed

Egress Queue Manager (DiffServ) 12

The Egress Queue Manager core component for DiffServ is identical to the Queue Manager (QM) core component detailed in [Chapter 11, “Queue Manager”](#) of this document.

The difference in the Egress Queue manager (Diffserv) from the QM core component is the following:

- The queue descriptor size is 8 LW. The core component must take this into account while allocating SRAM memory for the queue descriptor table.
- During initialization, the core component inserts into the system repository the \\QM\\QD_SRAM_BASE property. The property contains the address of the queue descriptor table in SRAM. WRED core component reads the property and uses the value to patch QD_SRAM_BASE symbol in the WRED microblock. Note that this requires that the QM core component is initialized before the WRED core component.

For a complete description of the Egress Queue Manager (DiffServ) Microblock, see [Chapter 21, “Egress Queue Manager \(DiffServ\) Microblock”](#) in *Intel® Internet Exchange Architecture Software Building Blocks Developer’s Manual*.

Scheduler

The Scheduler includes the following core components:

- [Chapter 13, “Scheduler”](#)

Two scheduler core components exist in the system—Ingress Scheduler core component and Egress Scheduler core component. The Ingress Scheduler core component configures the CSIX Scheduler microblock, and is called as CSIX Scheduler core component; Egress Scheduler core component configures the Packet Scheduler microblock and is called the Packet Scheduler core component.

- [Chapter 14, “Egress Scheduler \(DiffServ\)”](#)

The Scheduler core component for DiffServ is virtually identical to the Scheduler core component described in [Chapter 13, “Scheduler”](#).

The Scheduler core component initializes and configures its corresponding microblock. In a typical application, there are two Scheduler core components -- the CSIX Scheduler runs on the ingress side, while the Packet Scheduler runs on the egress side.

The Scheduler on the ingress side configures the scheduling algorithm for the CSIX Scheduler microblock and is called the CSIX Scheduler core component. The Scheduler on the egress side configures the scheduling algorithm for the Packet Scheduler microblock and is called Packet Scheduler core component.

Both the CSIX and Packet Scheduler core components have the following common functionality:

- Starts and configures itself as a standard core component
- Read configuration data on initialization from system registry or from the common header file in the absence of the registry

The CSIX Scheduler also has the following specific functionality:

- The CSIX Scheduler core component creates an array of entries in shared memory representing <port, class> maps for WRR algorithm in the microblock. The number of entries is application dependent, and is currently 1024. The number of entries can be changed based on the system design. See [Chapter 17, “Fabric Scheduler For OC-48”](#) of the *Intel® Internet Exchange Architecture Software Building Blocks Developer's Manual* for a description of array entries.
- Patches the CSIX Scheduler microblock with base address of the <port, class> map array

The Packet Scheduler also has the following specific functionality:

- The Packet Scheduler core component creates a region of shared memory combined from one array of 16 entries (representing the class weight for each output port and an array of 256 entries) representing credit for each class queue (16 class queues per each of the 16 ports) to support the DRR algorithm in the microblock. See [Chapter 19, “OC-48 WRR/DRR Packet Scheduler”](#) of the *Intel® Internet Exchange Architecture Software Building Blocks Developer's Manual* for a description of the array entries.
- Patches the Egress Packet Scheduler microblock with the base address of the combined weight array and the base address of the <port, queue> map array

The Scheduler core component does not receive packets from other microblocks or core components.

For complete details on this core component, see [Chapter 50, “Scheduler Core Component”](#) of the *Intel® Internet Exchange Architecture (IXA) Software Building Blocks Developer's Manual*.

13.1 Core Component Infrastructure API

Table 13-1 summarizes the Scheduler core component Infrastructure API.

Table 13-1. Scheduler Core Component Infrastructure API

Name	Description
<code>ix_cc_scheduler_init()</code>	Initializes the scheduler core component.
<code>ix_cc_scheduler_fini()</code>	Terminates the scheduler component.

13.1.1 `ix_cc_scheduler_init()`

This function initializes the Scheduler core component. It is called as result of the `ix_cci_cc_create()` Core Component Infrastructure function—see *Intel® Internet Exchange Architecture Portability Framework Reference Manual*—which is called by the Execution Engine. This function must return successfully before any other function in the core component is called. It performs static configuration for the microblock by allocating and patching the required variables and a memory block into the microblock using the Resource Manager API.

This function checks the registry to determine if the core component is running on the Ingress or the Egress side. All property values for both Scheduler core components need to be set in the registry or defined before initialization. Based on the type of Scheduler—that is, Ingress or Egress—different symbols are patched into microblocks.

C Syntax

```
ix_error ix_cc_scheduler_init(
    ix_cc_handle arg_hCcHandle,
    void **arg_ppContext);
```

Input

`arg_hCcHandle` The handle to the core component.

Output/Returns

`arg_ppContext` A pointer to where the Scheduler core component internal context pointer is stored. The internal context pointer is allocated by the core component. The Scheduler context contains configuration data for the microblock. This pointer is used later by the `ix_cc_scheduler_fini()` function to free memory when this core component is destroyed.

Return Value Returns a valid `ix_error`.

- `IX_SUCCESS`—the operation succeeded.
- A valid `ix_error`—the operation failed

13.1.2 ix_cc_scheduler_fini()

This function terminates the scheduler component. It executes when the execution engine running this core component is shut down. This function frees all allocated memory and resources obtained in the `ix_cc_scheduler_init()` function.

C Syntax

```
ix_error ix_cc_scheduler_fini(  
    ix_cc_handle arg_hCcHandle,  
    void* arg_pContext);
```

Input

<code>arg_hCcHandle</code>	A handle to the Scheduler core component.
<code>arg_pContext</code>	A pointer, allocated during a prior call to <code>ix_cc_scheduler_init()</code> , to control block memory. This function uses the pointer <code>arg_pContext</code> to de-allocate the control block memory.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none">• <code>IX_SUCCESS</code>—the operation succeeded.• A valid <code>ix_error</code>—the operation failed
--------------	---

The Scheduler core component for DiffServ is virtually identical to the Scheduler core component described in [Chapter 13, “Scheduler”](#).

The only difference is that the Scheduler (DiffServ) core component has the additional new patching symbol listed in [Table 14-1](#).

Table 14-1. New Patching Symbols in Scheduler (DiffServ) Core Component

Variable	Default	Description
SCHED_HIGH_PRIORITY_MASK	0x00FF	Bit mask denoting high priority queues on a port. The value is defined in SCHED/HIGH_PRIORITY_MASK property.

For complete information on the Egress Scheduler Microblock for DiffServ, refer to [Chapter 22, “Egress Scheduler \(DiffServ\) Microblock”](#) in *Intel® Internet Exchange Architecture Software Building Blocks Developer’s Manual*.

Forwarder

The following chapters are included in this section:

- [Chapter 15, “IPv4 Forwarder”](#)

IPv4 Forwarder - IPv4 Forwarder is a core component that assists IPv4 microblock to forward packet buffers. IPv4 Forwarder core component together with the microblock are implementation of near-RFC1812 compliant unicast capability on IXP2X00.

- [Chapter 16, “IPv6 Forwarder”](#)

IPv6 Forwarder - IPv6 Forwarder is a core component that assists IPv6 microblock to forward packets. IPv6 Forwarder core component together with the microblock is an implementation of RFC2460-compliant unicast capability on IXP2X00.

- [Chapter 17, “IPv6 to IPv4 Tunneling”](#)

IPv6 to IPv4 Tunneling - The IPv6-IPv4 Tunneling Core Component assists the IPv6-IPv4 Tunneling microblocks to provide IPv6 over IPv4 tunneling as specified in RFC 2893 and RFC 3056. .

- [Chapter 18, “NAT-PT Translation”](#)

The translation core component assists the translation microblock to implement the IPv6 to IPv4 (and vice-versa) address and protocol translation as defined in RFC2766 Network Address Translation-Port Translation specification (NAT-PT).

A single core component supports the translation microblock. The translation core component can receive packets from the translation microblock only. Messages can be received from other core components and from other system components responsible for configuring the forwarding plane.

The IPv4 Forwarder core component performs the following functions:

- Configures the IPv4 microblock (static configuration)
- Provides message and packet handlers to receive messages and packets from other core components and from the IPv4 microblock.
- Generates ICMP error messages
- Performs RFC1812 and RFC2644 checks
- Validates the IP header
- Handles fragmentation
- Handles packets with IP options

The IPv4 Forwarder core component receives packets and messages from several components of the system, including the IPv4 microblock, the POS RX core component, and the Stack Driver.

For complete details see [Chapter 52, “IPv4 Forwarder Core Component”](#) of the *Intel® Internet Exchange Architecture (IXA) Software Building Blocks Developer’s Manual*.

15.1 Data Structures, Types and Macros

Table 15-1 shows the data structures, types, and macros for the IPV4 Forwarder core component.

Table 15-1. IPV4 Forwarder Data Structures, Types, and Macros

Data Structures, Types, Macros	Description
<code>IX_CC_RTMV4_DUMP_ROUTE_SIZE</code>	Calculates the size of memory required to dump Route Table Manager routes
<code>IX_CC_RTMV4_DUMP_NEXT_HOP_SIZE</code>	Calculates the size of memory to dump Route Table Manager next hops
<code>ix_cc_rtmv4_nhid</code>	Type definition for next hop identifier
<code>Reserved Next Hop Ids</code>	Adds special next hops during initialization
<code>IX_CC_RTMV4_NHID_NO_ROUTE</code>	Structure definition for next hop information
<code>ix_cc_rtmv4_next_hop_info</code>	Defines the reserved next hop ID
<code>ix_cc_ipv4_dump_data</code>	Defines the structure describing memory dump information
<code>ix_cc_ipv4_stats_data</code>	Defines the structure describing counter information

15.1.1 `IX_CC_RTMV4_DUMP_ROUTE_SIZE`

This macro calculates the size of memory required to dump Route Table Manager routes. See [Section 23.1.2.1](#), “`IX_CC_RTMV4_DUMP_ROUTE_SIZE()`.”

15.1.2 `IX_CC_RTMV4_DUMP_NEXT_HOP_SIZE`

This macro calculates the size of memory to dump Route Table Manager next hops. See [Section 23.1.2.2](#), “`IX_CC_RTMV4_DUMP_NEXT_HOP_SIZE`.”

15.1.3 `ix_cc_rtmv4_nhid`

Type definition for next hop identifier. See [Section 23.1.1.2](#), “`ix_cc_rtmv4_nhid`.”

15.1.4 Reserved Next Hop Ids

The IPv4 Forwarder core component needs to add these special next hops during initialization.

```
#define IX_CC_IPV4_NH_BROADCAST 0xFFFFFFFFB
#define IX_CC_IPV4_NH_MULTICAST 0xFFFFFFFFC
#define IX_CC_IPV4_NH_DROP 0xFFFFFFFFD
#define IX_CC_IPV4_NH_LOCAL 0xFFFFFFFFE
```

15.1.4.1 `IX_CC_RTMV4_NHID_NO_ROUTE`

This macro defines the reserved next hop ID as defined in [Section 23.1.2.3](#), “`IX_CC_RTMV4_NHID_NO_ROUTE`.”

15.1.5 `ix_cc_rtmv4_next_hop_info`

This is the structure definition for next hop information, defined in [Section 23.1.1.3](#), “`ix_cc_rtmv4_next_hop_info`.”

15.1.6 `ix_cc_ipv4_dump_data`

The IPv4 Forwarder core component defines the structure describing memory dump information.

C Syntax

```
typedef struct ix_s_cc_ipv4_dump_data
{
    char* pBuffer;
    ix_uint32 size;
} ix_cc_ipv4_dump_data;
```

15.1.7 ix_cc_ipv4_stats_data

The IPv4 Forwarder core component defines the structure describing counter information.

```
typedef struct ix_s_cc_ipv4_stats_data
{
    ix_uint64  ipv4MbRcvdPkts;
    ix_uint64  ipv4MbFwdPkts;
    ix_uint64  ipv4MbDropPkts;
    ix_uint64  ipv4MbExcpPkts;
    ix_uint64  ipv4MbBadHeaderPkts;
    ix_uint64  ipv4MbBadTotalLengthPkts;
    ix_uint64  ipv4MbBadTTLPkts;
    ix_uint64  ipv4MbNoRoutePkts;
    ix_uint64  ipv4MbLengthTooSmallPkts;
    ix_uint32  ipv4CoreInvalidHeaderPkts;
    ix_uint32  ipv4CoreInvalidAddressPkts;
    ix_uint32  ipv4CoreRcvdPkts;
    ix_uint32  ipv4CoreFwdPkts;
    ix_uint32  ipv4CoreLocalDeliveryPkts;
    ix_uint32  ipv4CoreNoRoutePkts;
    ix_uint32  ipv4CoreInvalidFragPkts;
    ix_uint32  ipv4CoreCreatedFragmentedPkts;
    ix_uint32  ipv4CoreCreatedICMPMsgPkts;
    ix_uint32  ipv4CoreICMPSendFailed;
    ix_uint32  ipv4CoreCreatedICMPDestUnReachErrorPkts;
    ix_uint32  ipv4CoreCreatedTimeExceedPErrorkts;
    ix_uint32  ipv4CoreCreatedParamProblemErrorPkts;
    ix_uint32  ipv4CoreCreatedRedirectErrorPkts;
    ix_uint32  ipv4NumberOfRoutes;
    ix_uint32  ipv4NumberOfNextHops;
} ix_cc_ipv4_stats_data;
```

15.2 Core Component Infrastructure API

Table 15-2 shows the IPV4 Forwarder core component Infrastructure API.

Table 15-2. IPV4 Forwarder Core Component Infrastructure API

API	Description
<code>ix_cc_ipv4_init()</code>	Initializes the IPv4 Forwarder core component.
<code>ix_cc_ipv4_fini()</code>	Terminates services from the IPv4 Forwarder core component.
<code>ix_cc_ipv4_msg_handler()</code>	Passes messages to the IPv4 Forwarder core component.
<code>ix_cc_ipv4_microblock_high_priority_pkt_handler()</code>	Receives exception packets from IPv4 microblock.
<code>ix_cc_ipv4_microblock_low_priority_pkt_handler()</code>	Receive exception packets from IPv4 microblock
<code>ix_cc_ipv4_stackdrv_pkt_handler()</code>	Receives packets from the Stack Driver core component.
<code>ix_cc_ipv4_common_pkt_handler()</code>	Receives packets from any core component other than the Stack Driver.

15.2.1 `ix_cc_ipv4_init()`

This function is called and returned before requesting any service from the IPv4 Forwarder component. The `ix_cc_ipv4_init()` function should be called only once to initialize the IPv4 Forwarder component. This function performs the following:

- Allocates memory for symbols to be patched
- Creates 64 bit counters
- Registers packet and message handlers
- Initializes and configures Route Table Manager
- Allocates and initializes internal data structures
- Creates an event handler for ICMP

The calling application needs to initialize the IXA software framework and the Core Components Infrastructure before calling this function.

C Syntax

```
ix_error ix_cc_ipv4_init(
    ix_cc_handle arg_hCcHandle,
    void** arg_ppContext);
```

Input

`arg_hCcHandle` Handle to IPV4 core component, created by the Core Component Infrastructure; this is used to get other services (to add event handler) from the Core Component Infrastructure.

Input/Output

- `arg_ppContext` This is an INPUT and an OUTPUT parameter.
- As an input parameter, it is the handle of buffer free list, and dynamic property data is given by the system application during initialization.
 - As an output parameter, it represents the location where the pointer to the control block allocated by IPv4 Forwarder core component is stored. The control block is internal to IPv4 Forwarder core component and contains information about IPv4 internal data structures, allocated memory, and other relevant information. Later this pointer is passed into the `ix_cc_ipv4_fini` function to free memory and for other house keeping operations when IPv4 component is terminated.

Output/Returns

- Return Value Returns a valid `ix_error`.
- `IX_CC_ERROR_NULL`—the operation failed, due to null input parameter
 - `IX_CC_ERROR_OOM`—the operation failed, due to memory allocation failure
 - `IX_CC_IPV4_ERROR_FAILED_PATCHING`—the operation failed, due to patching failures
 - `IX_CC_ERROR_OOM_64BIT_COUNTER`—the operation failed, due to 64 bit counter creation failure
 - `IX_CC_IPV4_ERROR_REGISTRY`—the operation failed, due to invalid information from the registry
 - `IX_CC_IPV4_ERROR_RTM`—the operation failed, due to failure from Route Table Manager core component
 - `IX_CC_IPV4_ERROR_CCI`—the operation failed, due to failure from Core Component Infrastructure

15.2.2 ix_cc_ipv4_fini()

This function is called to terminate services from the IPv4 Forwarder core component. The function performs the following:

- Frees memory allocated during initialization
- Shuts down Route Table Manager
- Frees all the created resources—deletes 64 bit counter

The calling application must stop the microengines before calling this function. If the calling application claims for any services from IPv4 after calling this function, then the behavior is undefined.

C Syntax

```
ix_error ix_cc_ipv4_fini(
    ix_cc_handle arg_hCcHandle,
    void* arg_pContext);
```

Input

arg_hCcHandle	Handle to the core component.
arg_pContext	Pointer to the control block memory allocated earlier in ix_cc_ipv4_init() function.

Output/Returns

Return Value	<p>Returns a valid <code>ix_error</code>.</p> <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • <code>IX_CC_IPV4_ERROR_INVALID_INPUT_PARAM</code>—the operation failed, due to invalid data in <code>arg_pContext</code> • <code>IX_CC_ERROR_OOM</code>—the operation failed, due to failure to free memory • <code>IX_CC_IPV4_ERROR_RTM</code>—the operation failed, due to failure from Route Table Manager core component • <code>IX_CC_IPV4_ERROR_CCI</code>—the operation failed, due to failure from Core Component Infrastructure • <code>IX_CC_ERROR_OOM_64BIT_COUNTER</code>—the operation failed, due to 64 bit counter deletion failure • <code>IX_CC_ERROR_NULL</code>—the operation failed, due to invalid input pointer
--------------	--

15.2.3 ix_cc_ipv4_msg_handler()

This function is the message handler for the IPv4 Forwarder core component and updates the dynamic properties. The IPv4 Forwarder core component receives messages from the other core components through this message handler function and calls the appropriate library function internally to process the message.

C Syntax

```
ix_error ix_cc_ipv4_msg_handler(ix_buffer_handle arg_hDataToken,
                               ix_uint32 arg_UserData,
                               void* arg_pComponentContext);
```

Input

arg_hDataToken	Buffer handle embedding information for the message passed in arg_UserData.
arg_pComponentContext	Pointer to the IPv4 Forwarder core component-context that is passed to the core component when a message arrives. This context is defined by the core component and passed to the Core Components Infrastructure through the ix_cc_ipv4_init function.
arg_UserData	Message type.

Table 15-3 shows the IPV4 Forwarder core component messages.

Table 15-3. IPV4 Forwarder Core Component Messages

Messages	Description
IX_CC_IPV4_MSG_ADD_ROUTE	Helps client to add a route into the RTM (Route table Manager).
IX_CC_IPV4_MSG_DELETE_ROUTE	Helps client to delete a route from the RTM.
IX_CC_IPV4_MSG_UPDATE_ROUTE	Helps client to update a route in the RTM.
IX_CC_IPV4_MSG_LOOKUP_ROUTE	Helps client to lookup route information for a given IP address.
IX_CC_IPV4_MSG_PURGE_ROUTES	Helps client to delete all routes from the RTM.
IX_CC_IPV4_MSG_DUMP_ROUTES	Helps to dump all routes in memory
IX_CC_IPV4_MSG_ADD_NEXT_HOP	Helps client to add a next hop into the RTM.
IX_CC_IPV4_MSG_DELETE_NEXT_HOP	Helps client to delete a next hop from the RTM.
IX_CC_IPV4_MSG_UPDATE_NEXT_HOP	Helps client to update a next hop information in the RTM.
IX_CC_IPV4_MSG_GET_NEXT_HOP	Helps client to retrieve next hop information from the RTM.
IX_CC_IPV4_MSG_DUMP_NEXT_HOPS	Helps to dump all next hops in memory.
IX_CC_IPV4_MSG_PURGE_RTM	Helps client to delete all routes and next hops from the RTM.

Table 15-3. IPV4 Forwarder Core Component Messages (Continued)

Messages	Description
IX_CC_IPV4_MSG_SET_MTU	Helps client to set mtu for a next hop.
IX_CC_IPV4_MSG_SET_FLAGS	Helps client to set flags for a next hop.
IX_CC_IPV4_MSG_ADD_SOURCE_BROADCAST	Helps client to add source broadcast ip address in source broadcast table.
IX_CC_IPV4_MSG_DELETE_SOURCE_BROADCAST	Helps client to delete source broadcast ip address from source broadcast table
IX_CC_IPV4_MSG_GET_SLEEP_TIME	Helps client to get icmp sleep time.
IX_CC_IPV4_MSG_SET_SLEEP_TIME	Helps client to set icmp sleep time.
IX_CC_IPV4_MSG_GET_QUEUE_DEPTH	Helps client to get icmp queue depth.
IX_CC_IPV4_MSG_GET_PACKETS_TO_DRAIN	Helps client to get icmp get packets to drain quantity.
IX_CC_IPV4_MSG_SET_PACKETS_TO_DRAIN	Helps client to set icmp get packets to drain quantity.
IX_CC_IPV4_MSG_GET_STATISTICS	Helps client to get IPV4 statistics.

Output/Returns

Return Value Returns a valid `ix_error`.

- `IX_SUCCESS`—the operation succeeded
- `IX_CC_ERROR_NULL`—the operation failed, due to invalid pointer
- `IX_CC_ERROR_UNDEFINED_MSG`—the operation failed, due to unsupported message
- `IX_CC_IPV4_ERROR_MSG_LIBRARY`—the operation failed, due to error from message support library
- `IX_CC_IPV4_ERROR_BUFFER_FREE`—the operation failed, due to error from resource manager (RM) for freeing buffer

15.2.4 `ix_cc_ipv4_microblock_high_priority_pkt_handler()`

This is the registered function to receive exception packets from high priority queue of IPv4 microblock. This function sends packet to stack driver, that is, receives only local delivery packets.

C Syntax

```
ix_error ix_cc_ipv4_microblock_high_priority_pkt_handler (
    ix_buffer_handle arg_hDataToken,
    ix_uint32 arg_ExceptionCode,
    void* arg_pComponentContext)
```

Input

`arg_hDataToken` Handle to a buffer which contains exception packets from IPv4 microblock.

Input (Continued)

`arg_ExceptionCode` Exception codes generated by the IPv4 Forwarder microblock.

`arg_pComponentContext` Pointer to the IPv4 Forwarder core component context.

Output/Returns

Return Value Returns a valid `ix_error`.

- `IX_SUCCESS`—The operation succeeded
- `IX_CC_ERROR_SEND_FAIL`—operation failed, due to error from Core Component Infrastructure

15.2.5 `ix_cc_ipv4_microblock_low_priority_pkt_handler()`

This is the registered function to receive exception packets from the IPv4 microblock. This function internally calls/performs different functions/operations based on the exception codes for the given packet.

C Syntax

```
ix_error ix_cc_ipv4_microblock_low_priority_pkt_handler(
    ix_buffer_handle arg_hDataToken,
    ix_uint32 arg_ExceptionCode,
    void* arg_pComponentContext)
```

Input

`arg_hDataToken` Handle to a buffer which contains exception packets from IPv4 microblock.

`arg_ExceptionCode` Exception codes generated by IPv4 forwarder microblock.

`arg_pComponentContext` Pointer to IPV4 Forwarder core component context.

Output/Returns

Return Value Returns a valid `ix_error`.

- `IX_SUCCESS`—The operation succeeded
- `IX_CC_ERROR_SEND_FAIL`—operation failed, due to error from CCI
- `IX_CC_ERROR_UNDEFINED_EXCEP`—operation failed, due to unsupported exception code
- `IX_CC_ERROR_INTERNAL`—operation failed, due to internal error
- `IX_CC_ERROR_ALIGN`—operation failed, due to pointer not aligned properly
- `IX_CC_ERROR_NULL`—operation failed, due to invalid pointer
- `IX_CC_ERROR_RTM`—operation failed, due to RTM error

15.2.6 ix_cc_ipv4_stackdrv_pkt_handler()

This function receives packets from the Stack Driver core component. Packets coming from the Stack Driver need the following special processing by the IPv4 core component:

- no TTL decrement
- no IP header validation

C Syntax

```
ix_error ix_cc_ipv4_stackdrv_pkt_handler(ix_buffer_handle arg_hDataToken,
                                         ix_uint32 arg_ExceptionCode,
                                         void* arg_pComponentContext)
```

Input

arg_hDataToken	Handle to a buffer which contains packet from Stack Driver core component.
arg_ExceptionCode	Is Ignored
arg_pComponnetContext	Pointer to the IPv4 Forwarder core component context.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—The operation succeeded • <code>IX_CC_ERROR_SEND_FAIL</code>—the operation failed, due to error from Core Component Infrastructure • <code>IX_CC_ERROR_INTERNAL</code>—the operation failed, due to internal error • <code>IX_CC_ERROR_ALIGN</code>—the operation failed, due to pointer not being aligned properly • <code>IX_CC_ERROR_NULL</code>—the operation failed, due to invalid pointer
--------------	--

15.2.7 ix_cc_ipv4_common_pkt_handler()

This function receives packets from all the core components except from the Stack Driver—the interface RX core component.

C Syntax

```
ix_error ix_cc_ipv4_common_pkt_handler (ix_buffer_handle arg_hDataToken,
                                         ix_uint32 arg_ExceptionCode,
                                         void* arg_pComponentContext)
```

Input

<code>arg_hDataToken</code>	Handle to a buffer which contains exception packets from IPv4 microblock.
<code>arg_ExceptionCode</code>	Exception codes generated by IPv4 Forwarder microblock.
<code>arg_pComponnetContext</code>	Pointer to IPV4 Forwarder core component context.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—The operation succeeded <code>IX_CC_ERROR_INTERNAL</code>—the operation failed, due to internal error <code>IX_CC_ERROR_ALIGN</code>—the operation failed, due to pointer not being aligned properly <code>IX_CC_ERROR_NULL</code>—the operation failed, due to invalid pointer <code>IX_CC_ERROR_SEND_FAIL</code>—the operation failed, due to error from CCI
--------------	--

15.3 Message Helper API

Table 15-4 shows the IPV4 Forwarder Message Helper API.

Table 15-4. IPV4 Forwarder Message Helper API

API	Description
<code>ix_cc_ipv4_async_add_route()</code>	Adds a route in the Route Table Manager
<code>ix_cc_ipv4_async_delete_route()</code>	Deletes a route in the RTM
<code>ix_cc_ipv4_async_update_route()</code>	Update an existing route in the RTM
<code>ix_cc_ipv4_async_lookup_route()</code>	Looks up routing information for a given IP address
<code>ix_cc_ipv4_async_purge_routes()</code>	Removes all routes from the RTM
<code>ix_cc_ipv4_async_dump_routes()</code>	Dumps all routes of RTM in memory
<code>ix_cc_ipv4_async_add_next_hop()</code>	Adds the next hop information to the RTM database
<code>ix_cc_ipv4_async_delete_next_hop()</code>	Deletes the next hop information from the RTM database
<code>ix_cc_ipv4_async_update_next_hop()</code>	Updates the next hop information into the RTM database
<code>ix_cc_ipv4_async_get_next_hop()</code>	Retrieves the next hop information from the RTM database
<code>ix_cc_ipv4_async_dump_next_hops()</code>	Dumps all next hops of RTM in memory
<code>ix_cc_ipv4_async_purge_rtm()</code>	Removes all routes and next hops from the RTM database

Table 15-4. IPV4 Forwarder Message Helper API (Continued)

API	Description
ix_cc_ipv4_async_set_mtu()	Updates MTU for a given next hop
ix_cc_ipv4_async_set_flags()	Updates flags for a given next hop.
ix_cc_ipv4_async_get_sleep_time()	Retrieves the number of seconds allocated for calling the ICMP event
ix_cc_ipv4_async_set_sleep_time()	Sets the number of seconds for calling the ICMP event handler
ix_cc_ipv4_async_get_queue_depth()	Retrieves the depth of the ICMP error message queue
ix_cc_ipv4_async_get_packets_to_drain()	Retrieves the maximum number of ICMP error messages to send
ix_cc_ipv4_async_set_packets_to_drain()	Sets the maximum number of ICMP error messages to send
ix_cc_ipv4_async_get_statistics()	Retrieves the statistical report from the IPv4 Forwarder core component

15.3.1 [ix_cc_ipv4_async_add_route\(\)](#)

This message helper function adds a route in the Route Table Manager. The calling application needs to successfully add `nextHopId` to Route Table Manager by calling the `ix_cc_ipv4_async_add_next_hop` messaging helper function before sending this message to the IPv4 Forwarder core component.

C Syntax

```
ix_error ix_cc_ipv4_async_add_route (
    ix_uint32 arg_IpAddr,
    ix_uint32 arg_NetMask,
    ix_cc_rtmv4_nhid arg_NextHopId,
    ix_cc_ipv4_cb_route_op arg_Callback,
    void* arg_pUserContext);
```

Input

<code>arg_IpAddr</code>	Network IP address identifying the route.
<code>arg_NetMask</code>	Network mask for the route.
<code>arg_NextHopId</code>	Next hop identifier for route.
<code>arg_Callback</code>	Pointer to the calling application-provided callback function.
<code>arg_pUserContext</code>	Pointer to the calling application-defined context. See ix_cc_ipv4_cb_route_op .

Note: The `ix_cc_ipv4_cb_route_op` callback is a generic callback type and can be used for IPV4 message helpers that do not require data to be returned in the callback. Only the result and context are relevant.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> .
	<ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded • <code>IX_CC_ERROR_NULL</code>—the operation failed, due to invalid pointer • <code>IX_CC_IPV4_ERROR_MSG_LIBRARY</code>—the operation failed, an error was signaled by the Message Support Library.

15.3.1.1 `ix_cc_ipv4_cb_route_op`

The function prototype of the callback for `ix_cc_ipv4_async_add_route()`.

C Syntax

```
ix_error (*ix_cc_ipv4_cb_route_op) (
    ix_error arg_Result,
    void* arg_pContext
);
```

Input

<code>arg_Result</code>	Indicates error conditions for the call
<code>arg_pContext</code>	Pointer to the calling application-provided context that was sent by an API call. Used by the calling application to identify the instance of the request.

15.3.2 `ix_cc_ipv4_async_delete_route()`

This message helper function deletes a route in the Route Table Manager. The calling application needs to delete all routes associated with a next hop to delete the next hop by calling `ix_cc_ipv4_async_delete_next_hop()` message helper function.

C Syntax

```
ix_error ix_cc_ipv4_async_delete_route(
    ix_uint32 arg_IpAddr,
    ix_uint32 arg_NetMask,
    ix_cc_ipv4_cb_route_op arg_Callback,
    void* arg_pUserContext);
```

Input

<code>arg_IpAddr</code>	Network IP address which identifies the route.
<code>arg_NetMask</code>	Network mask for the route.
<code>arg_Callback</code>	Pointer to the calling application-provided callback function.
<code>arg_pUserContext</code>	Pointer to the calling application-defined context. See ix_cc_ipv4_cb_route_op .

Note: The [ix_cc_ipv4_cb_route_op](#) callback is a generic callback type and can be used for IPV4 message helpers that do not require data to be returned in the callback. Only the result and context are relevant.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded <code>IX_CC_ERROR_NULL</code>—the operation failed, due to invalid pointer <code>IX_CC_IPV4_ERROR_MSG_LIBRARY</code>—the operation failed, an error was signaled by the Message Support Library.
--------------	--

15.3.2.1 [ix_cc_ipv4_cb_route_op](#)

The function prototype of the callback for [ix_cc_ipv4_async_delete_route\(\)](#).

C Syntax

```
ix_error (*ix_cc_ipv4_cb_route_op) (
    ix_error arg_Result,
    void* arg_pContext
);
```

Input

<code>arg_Result</code>	Indicates an error condition for the call.
<code>arg_pContext</code>	Pointer to the calling application-provided context which was sent by an API call. Used by the calling application to identify the instance of the request.

15.3.3 ix_cc_ipv4_async_update_route()

This message helper function updates an existing route in the Route Table Manager. Client needs to successfully add next hop and route relate to the next hop to the RTM by calling [ix_cc_ipv4_async_add_next_hop\(\)](#) and [ix_cc_ipv4_async_add_route\(\)](#) messaging helper functions before sending this message to IPv4 Forwarder core component.

C Syntax

```
ix_error ix_cc_ipv4_async_update_route (
    ix_uint32 arg_IpAddr,
    ix_uint32 arg_NetMask,
    ix_cc_rtmv4_nhid arg_NextHopId,
    ix_cc_ipv4_cb_route_op arg_Callback,
    void* arg_pUserContext);
```

Input

arg_IpAddr	Network IP address identifying the route.
arg_NetMask	Network mask for the route.
arg_NextHopId	Next hop identifier for route.
arg_Callback	Pointer to the calling application-provided callback function. See ix_cc_ipv4_cb_route_op .
arg_pUserContext	Pointer to the calling application-defined context.

Note: The [ix_cc_ipv4_cb_route_op](#) callback is a generic callback type and can be used for IPV4 message helpers that do not require data to be returned in the callback. Only the result and context are relevant.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> IX_SUCCESS—the operation succeeded IX_ERROR_INVALID_POINTER—the operation failed, due to invalid pointer IX_CC_IPV4_ERROR_MSG_LIBRARY—the operation failed, an error was signaled by the Message Support Library.
--------------	---

15.3.3.1 ix_cc_ipv4_cb_route_op

The function prototype of the callback for `ix_cc_ipv4_async_update_route()`.

C Syntax

```
ix_error (*ix_cc_ipv4_cb_route_op)(
    ix_error arg_Result,
    void* arg_pContext
);
```

Input

<code>arg_Result</code>	The error conditions for the call
<code>arg_pContext</code>	Pointer to user provided context, that was sent by the API call. Used by the calling application to identify the instance of the request

15.3.4 ix_cc_ipv4_async_lookup_route()

This function looks up routing information for a given IP address. If there is a successful match, then the information given to the calling application are `nextHopId` (including the blade identifier), `portId`, `MTU` and flags. If there is no match, then `IX_CC_RTMV4_NHID_NO_ROUTE` is assigned to the `nextHopId` field of the `pNextHopData` and all the other information is invalid.

C Syntax

```
ix_error ix_cc_ipv4_async_lookup_route (
    ix_uint32 arg_IpAddr,
    ix_cc_ipv4_cb_lookup_route arg_Callback,
    void* arg_pUserContext);
```

Input

<code>arg_IpAddr</code>	Network IP address identifying the route.
<code>arg_Callback</code>	Pointer to the calling application-provided callback function.
<code>arg_pUserContext</code>	Pointer to the calling application-defined context. See ix_cc_ipv4_cb_lookup_route .

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded <code>IX_CC_ERROR_NULL</code>—the operation failed, due to invalid pointer <code>IX_CC_IPV4_ERROR_MSG_LIBRARY</code>—the operation failed, an error was signaled by the Message Support Library.
--------------	--

15.3.4.1 ix_cc_ipv4_cb_lookup_route

The function prototype of the callback for `ix_cc_ipv4_async_lookup_route()`.

C Syntax

```
ix_error (*ix_cc_ipv4_cb_lookup_route) (
    ix_error arg_Result,
    void* arg_pContext,
    ix_cc_rtmv4_next_hop_info* arg_pNextHopInfo
);
```

Input

<code>arg_Result</code>	Indicates error conditions for the call.
<code>arg_pContext</code>	Pointer to the calling application-provided context, that was sent by an API call. Used by the calling application to identify the instance of the request.
<code>arg_pNextHopInfo</code>	Pointer to structure containing the next hop. The result of a successful lookup is placed in this structure.

15.3.5 ix_cc_ipv4_async_purge_routes()

This function removes all routes from the Route Table Manager.

C Syntax

```
ix_error ix_cc_ipv4_async_purge_routes();
```

Input

None.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded • <code>IX_CC_ERROR_NULL</code>—the operation failed, due to invalid pointer • <code>IX_CC_IPV4_ERROR_MSG_LIBRARY</code>—the operation failed, an error was signaled by the Message Support Library.
--------------	--

15.3.6 ix_cc_ipv4_async_dump_routes()

This function dumps all routes of Route Table Manager in memory.

C Syntax

```
ix_error ix_cc_ipv4_async_dump_routes (
    ix_cc_ipv4_cb_dump_data arg_Callback,
    void* arg_pUserContext);
```

Input

<code>arg_Callback</code>	Pointer to the calling application-provided callback function.
<code>arg_pUserContext</code>	Pointer to the calling application-defined context. See ix_cc_ipv4_cb_dump_data .

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded <code>IX_CC_ERROR_NULL</code>—the operation failed, due to invalid pointer <code>IX_CC_IPV4_ERROR_MSG_LIBRARY</code>—the operation failed, an error was signaled by the Message Support Library.
--------------	--

15.3.6.1 ix_cc_ipv4_cb_dump_data

Function prototype of the callback function for `ix_cc_ipv4_async_dump_routes()`.

C Syntax

```
ix_error (*ix_cc_ipv4_cb_dump_data) (
    ix_error arg_Result,
    void* arg_pContext,
    ix_cc_ipv4_dump_data* arg_pBuffer
);
```

Input

<code>arg_Result</code>	Indicates error conditions for the call.
<code>arg_pContext</code>	Pointer to the calling application-provided context, that was sent by an API call. Used by the calling application to identify the instance of the request.
<code>arg_pBuffer</code>	Pointer to the structure containing route information. The result of a successful operation is placed in this structure

15.3.7 `ix_cc_ipv4_async_add_next_hop()`

This function adds the next hop information to the Route Table Manager database. The calling application needs to add next hop to Route Table Manager in order to add routes referring to that next hop.

C Syntax

```
ix_error ix_cc_ipv4_async_add_next_hop(
    ix_cc_rtmv4_nhid arg_NextHopId,
    ix_cc_rtmv4_add_next_hop_info* arg_pNextHopData,
    ix_cc_ipv4_cb_route_op arg_Callback,
    void* arg_pUserContext);
```

Input

<code>arg_NextHopId</code>	Identifier of the next hop to be added
<code>arg_pNextHopData</code>	See Section 23.1.1.3 , “ <code>ix_cc_rtmv4_next_hop_info</code> ”.
<code>arg_Callback</code>	Pointer to the calling application-provided callback function. See ix_cc_ipv4_cb_route_op .
<code>arg_pUserContext</code>	Pointer to the calling application-defined context.

Note: The `ix_cc_ipv4_cb_route_op` callback is a generic callback type and can be used for IPV4 message helpers that do not require data to be returned in the callback. Only the result and context are relevant.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> .
	<ul style="list-style-type: none"> IX_SUCCESS—the operation succeeded IX_CC_ERROR_NULL—the operation failed, due to invalid pointer IX_CC_IPV4_ERROR_MSG_LIBRARY—the operation failed, an error was signaled by the Message Support Library

15.3.7.1 `ix_cc_ipv4_cb_route_op`

Function prototype of the callback function for `ix_cc_ipv4_async_add_next_hop()`.

C Syntax

```
ix_error (*ix_cc_ipv4_cb_route_op) (
    ix_error arg_Result,
    void* arg_pContext
);
```

Input

<code>arg_Result</code>	Indicates error conditions for the call
<code>arg_pContext</code>	Pointer to the calling application-provided context, that was sent by an API call. Used by the calling application to identify the instance of the request.

15.3.8 `ix_cc_ipv4_async_delete_next_hop()`

This message helper function deletes the next hop information from the Route Table Manager database. The calling application needs to remove all the routes associated with the next hop before calling this function. If not, the calling application receives an error.

C Syntax

```
ix_error ix_cc_ipv4_async_delete_next_hop (
    ix_cc_rtmv4_nhid arg_NextHopId,
    ix_cc_ipv4_cb_route_op arg_Callback,
    void* arg_pUserContext);
```

Input

<code>arg_NextHopId</code>	Identifier of the next hop to be removed.
<code>arg_Callback</code>	Pointer to the calling application-provided callback function.
<code>arg_pUserContext</code>	Pointer to the calling application-defined context. See ix_cc_ipv4_cb_route_op .

Note: The [ix_cc_ipv4_cb_route_op](#) callback is a generic callback type and can be used for IPV4 message helpers that do not require data to be returned in the callback. Only the result and context are relevant.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> .
	<ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded <code>IX_CC_ERROR_NULL</code>—the operation failed, due to invalid pointer <code>IX_CC_IPV4_ERROR_MSG_LIBRARY</code>—the operation failed, an error was signaled by the Message Support Library

15.3.8.1 ix_cc_ipv4_cb_route_op

Function prototype of the callback function for `ix_cc_ipv4_async_delete_next_hop()`.

C Syntax

```
ix_error (*ix_cc_ipv4_cb_route_op) (
    ix_error arg_Result,
    void* arg_pContext
);
```

Input

<code>arg_Result</code>	Indicates error conditions for the call.
<code>arg_pContext</code>	Pointer to the calling application-provided context, that was sent by an API call. Used by the calling application to identify the instance of the request.

15.3.9 ix_cc_ipv4_async_update_next_hop()

This function updates the next hop information into the Route Table Manager database. The calling application must add next hop information to the RTM in order to add routes referring to the next hop.

C Syntax

```
ix_error ix_cc_ipv4_async_updata_next_hop(
    ix_cc_rtmv4_nhid arg_NextHopId,
    ix_cc_rtmv4_add_next_hop_info* arg_pNextHopData,
    ix_cc_ipv4_cb_route_op arg_Callback,
    void* arg_pUserContext);
```

Input

<code>arg_NextHopId</code>	Identifier of next hop to be updated
<code>arg_pNextHopData</code>	Defined in Section 23.1.1.3 , “ <code>ix_cc_rtmv4_next_hop_info</code> ”.
<code>arg_Callback</code>	Pointer to calling application-provided callback function. See ix_cc_ipv4_cb_route_op .
<code>arg_pUserContext</code>	Pointer to the calling application-defined context.

Note: The callback ([ix_cc_ipv4_cb_route_op](#)) is a generic callback type and can be used for IPv4 message helpers that do not require data to be returned in the callback. Only the result and context are relevant.

Output/Returns

Return Value Returns a valid `ix_error`.

- `IX_SUCCESS`—the operation succeeded
- `IX_ERROR_INVALID_POINTER`—the operation failed, due to invalid pointer
- `IX_CC_IPV4_ERROR_MSG_LIBRARY`—the operation failed, an error was signaled by the Message Support Library

15.3.9.1 [ix_cc_ipv4_cb_route_op](#)

Function prototype of callback functions for calls to [ix_cc_ipv4_async_update_next_hop\(\)](#).

C Syntax

```
ix_error (*ix_cc_ipv4_cb_route_op) (
    ix_error arg_Result,
    void* arg_pContext
);
```

Input

<code>arg_Result</code>	The error conditions for the call
<code>arg_pContext</code>	Pointer to the calling application-provided context, that was sent by the API call. Used by the calling application to identify the instance of the request.

15.3.10 [ix_cc_ipv4_async_get_next_hop\(\)](#)

This message helper function retrieves the next hop information from the Route Table Manager database.

C Syntax

```
ix_error ix_cc_ipv4_async_get_next_hop (
    ix_cc_rtmv4_nhid arg_NextHopId,
    ix_cc_ipv4_cb_get_next_hop arg_Callback,
    void* arg_pUserContext);
```

Input

<code>arg_NextHopId</code>	Identifier of the next hop information to be retrieved from the Route Table Manager.
<code>arg_Callback</code>	Pointer to the calling application-provided callback function.

Input (Continued)

<code>arg_NextHopId</code>	Identifier of the next hop information to be retrieved from the Route Table Manager.
<code>arg_pUserContext</code>	Pointer to the calling application-defined context. See ix_cc_ipv4_cb_get_next_hop .

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded <code>IX_CC_ERROR_NULL</code>—the operation failed, due to invalid pointer <code>IX_CC_IPV4_ERROR_MSG_LIBRARY</code>—the operation failed, an error was signaled by the Message Support Library
--------------	---

15.3.10.1 `ix_cc_ipv4_cb_get_next_hop`

Function prototype of callback functions for calls to `ix_cc_ipv4_async_get_next_hop()`.

C Syntax

```
ix_error (*ix_cc_ipv4_cb_get_next_hop) (
    ix_error arg_Result,
    void* arg_pContext,
    ix_cc_rtmv4_next_hop_info* arg_pNextHopInfo
);
```

Input

<code>arg_Result</code>	Indicates error conditions for the call
<code>arg_pContext</code>	Pointer to the calling application-provided context, that was sent by API call. Used by the calling application to identify the instance of the request.
<code>arg_pNextHopInfo</code>	Pointer to the structure containing the next hop. The result of a successful lookup is placed in this structure.

15.3.11 `ix_cc_ipv4_async_dump_next_hops()`

This function dumps all next hops of Route Table Manager in memory.

C Syntax

```
ix_error ix_cc_ipv4_async_dump_next_hops (
    ix_cc_ipv4_cb_dump_data arg_Callback,
    void* arg_pUserContext);
```

Input

<code>arg_Callback</code>	Pointer to the calling application-provided callback function.
<code>arg_pUserContext</code>	Pointer to the calling application-defined context. See ix_cc_ipv4_cb_dump_data .

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—returned if the operation succeeds. <code>IX_CC_ERROR_NULL</code>—the operation failed, due to an invalid pointer. <code>IX_CC_IPV4_ERROR_MSG_LIBRARY</code>—Failure from Message Support Library
--------------	---

15.3.11.1 `ix_cc_ipv4_cb_dump_data`

Function prototype of callback functions for calls to `ix_cc_ipv4_async_dump_next_hops()`.

C Syntax

```
ix_error (*ix_cc_ipv4_cb_dump_data) (
    ix_error arg_Result,
    void* arg_pContext,
    ix_cc_ipv4_dump_data* arg_pBuffer
);
```

Input

<code>arg_Result</code>	Indicates error conditions for the call
<code>arg_pContext</code>	Pointer to the calling application-provided context, that was sent by API call. Used by the calling application to identify the instance of the request.
<code>arg_pBuffer</code>	Pointer to the structure containing next hops Information. The result of a successful operation is placed in this structure

15.3.12 `ix_cc_ipv4_async_purge_rtm()`

This function removes all routes and next hops from the Route Table Manager database.

C Syntax

```
ix_error ix_cc_ipv4_async_purge_rtm ();
```

Input

None.

Output/Returns

- | | |
|--------------|---|
| Return Value | <p>Returns a valid <code>ix_error</code>.</p> <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—returned if the operation succeeds. • <code>IX_CC_IPV4_ERROR_MSG_LIBRARY</code>—Failure from Message Support Library |
|--------------|---|

15.3.13 `ix_cc_ipv4_async_set_mtu()`

This function updates MTU for a given next hop. Before calling this message helper, the calling application needs to call `ix_cc_ipv4_async_add_next_hop()` to add next hop identifier in the Route Table Manager database and validate the MTU.

C Syntax

```
ix_error ix_cc_ipv4_async_set_mtu(
    ix_cc_rtmv4_nhid arg_NextHopId,
    ix_uint32 arg_Mtu,
    ix_cc_ipv4_cb_route_op arg_Callback,
    void* arg_pUserContext);
```

Input

- | | |
|-------------------------------|--|
| <code>arg_NextHopId</code> | Identifier of next hop. |
| <code>arg_Mtu</code> | New maximum transmission unit for the given next hop. |
| <code>arg_callback</code> | Pointer to the calling application callback function. See ix_cc_ipv4_cb_route_op . |
| <code>arg_pUserContext</code> | Pointer to the calling application-defined context. |

Note: The `ix_cc_ipv4_cb_route_op` callback is a generic callback type and can be used for IPV4 message helpers that do not require data to be returned in the callback. Only the result and context are relevant.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> .
	<ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeds. • <code>IX_CC_ERROR_NULL</code>—the operation failed, due to an invalid pointer. • <code>IX_CC_IPV4_ERROR_MSG_LIBRARY</code>—the operation failed, An error was signaled by the Message Support Library.

15.3.14 `ix_cc_ipv4_cb_route_op`

Function prototype of callback functions for calls to `ix_cc_ipv4_async_set_mtu()`.

C Syntax

```
ix_error (*ix_cc_ipv4_cb_route_op) (
    ix_error arg_Result,
    void* arg_pContext
);
```

Input

<code>arg_Result</code>	Indicates error conditions for the call
<code>arg_pContext</code>	Pointer to the calling application-provided context, that was sent by API call. Used by the calling application to identify the instance of the request

15.3.15 `ix_cc_ipv4_async_set_flags()`

This function updates flags for a given next hop. Before calling this message helper, the calling application needs to call the `ix_cc_ipv4_async_add_next_hop()` to add next hop identifier in the Route Table Manager database. The supported flags[BLDBLK] are defined by the IPv4 Forwarder microblock.

C Syntax

```
ix_error ix_cc_ipv4_async_set_flags(
    ix_cc_rtmv4_nhid arg_NextHopId,
    ix_uint32 arg_Flags,
    ix_cc_ipv4_cb_route_op arg_Callback,
    void* arg_pUserContext);
```

Input

<code>arg_NextHopId</code>	Identifier of next hop.
<code>arg_Flags</code>	New flags for the given next hop.

Input (Continued)

<code>arg_NextHopId</code>	Identifier of next hop.
<code>arg_Callback</code>	Pointer to the calling application-provided callback function. See ix_cc_ipv4_cb_route_op .
<code>arg_pUserContext</code>	Pointer to the calling application-defined context.

Note: The callback (`arg_callback`) is a generic callback type and can be used for IPV4 message helpers that do not require data to be returned in the callback. Only the result and context are relevant.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeds. <code>IX_CC_ERROR_NULL</code>—the operation failed, due to an invalid pointer. <code>IX_CC_IPV4_ERROR_INVALID_FLAGS</code>—the operation failed, due to unsupported flags. <code>IX_CC_IPV4_ERROR_MSG_LIBRARY</code>—the operation failed, an error was signaled by the Message Support Library.
--------------	---

15.3.15.1 [ix_cc_ipv4_cb_route_op](#)

Function prototype of callback functions for calls to [ix_cc_ipv4_set_flags\(\)](#).

C Syntax

```
ix_error (*ix_cc_ipv4_cb_route_op) (
    ix_error arg_Result,
    void* arg_pContext
);
```

Input

<code>arg_Result</code>	Indicates error conditions for the call
<code>arg_pContext</code>	Pointer to the user provided context, that was sent by the API call. Used by the calling application to identify the instance of the request

15.3.16 ix_cc_ipv4_async_get_sleep_time()

This message helper function retrieves the number of seconds allocated for calling the ICMP event handler, that is message dequeue and send.

C Syntax

```
ix_error ix_cc_ipv4_async_get_sleep_time (
    ix_cc_ipv4_cb_get_sleep_time arg_Callback,
    void* arg_pUserContext);
```

Input

arg_Callback	Pointer to the calling application-provided callback function.
arg_pUserContext	Pointer to the calling application-defined context. See ix_cc_ipv4_cb_get_sleep_time .

Output/Returns

Return Value	Returns a valid ix_error. <ul style="list-style-type: none"> • IX_SUCCESS—the operation succeeds. • IX_CC_ERROR_NULL—the operation failed, due to an invalid pointer. • IX_CC_IPV4_ERROR_MSG_LIBRARY—the operation failed, An error was signaled by the Message Support Library.
--------------	---

15.3.16.1 ix_cc_ipv4_cb_get_sleep_time

Function prototype of callback functions for calls to [ix_cc_ipv4_get_sleep_time\(\)](#).

C Syntax

```
ix_error (*ix_cc_ipv4_cb_get_sleep_time) (
    ix_error arg_Result,
    void* arg_pContext,
    ix_uint16 arg_SleepTime
);
```

Input

arg_Result	Indicates error conditions for the call.
arg_pContext	Pointer to calling application-provided context, that was sent by API call. Used by the calling application to identify the instance of the request.
arg_pSleepTime	Pointer to the sleep period currently set.

15.3.17 ix_cc_ipv4_async_set_sleep_time()

This message helper function sets the number of seconds for calling the ICMP event handler, that is, message dequeue and send. The lower the value of `arg_SleepTime`, higher the frequency of the ICMP message process.

C Syntax

```
ix_error ix_cc_ipv4_async_set_sleep_time (
    ix_uint16 arg_SleepTime);
```

Input

`arg_SleepTime` Minimum value is 1 second and maximum value is 32 seconds.

Output/Returns

Return Value Returns a valid `ix_error`.

- `IX_SUCCESS`—the operation is succeeded.
- `IX_CC_IPV4_ERROR_INVALID_SLEEP_TIME`—the operation failed, due to unsupported sleep time
- `IX_CC_IPV4_ERROR_MSG_LIBRARY`—the operation failed, an error was signaled by the Message Support Library.

15.3.18 ix_cc_ipv4_async_get_queue_depth()

This message helper function retrieves the depth of the ICMP error message queue. The calling application retrieves the maximum number of ICMP messages that can fit in the queue and not the number of messages currently in the queue.

C Syntax

```
ix_error ix_cc_ipv4_async_get_queue_depth (
    ix_cc_ipv4_cb_get_queue_depth arg_Callback,
    void* arg_pUserContext);
```

Input

`arg_Callback` Pointer to the calling application-provided callback function.

`arg_pUserContext` Pointer to the calling application-defined context.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> .
	<ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeds. • <code>IX_CC_ERROR_NULL</code>—the operation failed, due to an invalid pointer • <code>IX_CC_IPV4_ERROR_MSG_LIBRARY</code>—the operation failed, An error was signaled by the Message Support Library.

15.3.18.1 `ix_cc_ipv4_cb_get_queue_depth`

Function prototype of callback functions for calls to `ix_cc_ipv4_async_get_queue_depth()`.

C Syntax

```
ix_error (*ix_cc_ipv4_cb_get_queue_depth) (
    ix_error arg_Result,
    void* arg_pContext,
    ix_uint16 arg_QueueDepth
);
```

Input

<code>arg_Result</code>	Indicates error conditions for the call
<code>arg_pContext</code>	Pointer to the calling application-provided context, that was sent by an API call. Used by the calling application to identify the instance of the request.
<code>arg_QueueDepth</code>	Number of entries that the ICMP error message queue can hold.

15.3.19 `ix_cc_ipv4_async_get_packets_to_drain()`

This message helper function retrieves the maximum number of ICMP error messages to send each time the queue processing event is executed. The processing event sends the retrieved number of ICMP error packets from the queue before going back to sleep.

C Syntax

```
ix_error ix_cc_ipv4_async_get_packets_to_drain (
    ix_cc_ipv4_cb_get_packets_to_drain arg_Callback,
    void* arg_pUserContext);
```

Input

<code>arg_Callback:</code>	Pointer to the calling application-provided callback function.
<code>arg_pUserContext</code>	Pointer to the calling application-defined context. See ix_cc_ipv4_cb_get_packets_to_drain .

Output/Returns

- | | |
|--------------|--|
| Return Value | <p>Returns a valid <code>ix_error</code>.</p> <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeds. • <code>IX_CC_IPV4_ERROR_MSG_LIBRARY</code>—the operation failed, An error was signaled by the Message Support Library. |
|--------------|--|

15.3.19.1 `ix_cc_ipv4_cb_get_packets_to_drain`

Function prototype of callback functions for calls to `ix_cc_ipv4_async_get_packets_to_drain()`.

C Syntax

```
ix_error (*ix_cc_ipv4_cb_get_packets_to_drain) (
    ix_error arg_Result,
    void* arg_pContext,
    ix_uint16 arg_PacketsToDrain
);
```

Input

- | | |
|----------------------------------|---|
| <code>arg_Result</code> | Indicates error conditions for the call |
| <code>arg_pContext</code> | Pointer to the calling application-provided context, that was sent by an API call. Used by the calling application to identify the instance of the request. |
| <code>arg_pPacketsToDrain</code> | Pointer to a maximum number of packets to send each time an event is triggered for processing the queue. |

15.3.20 `ix_cc_ipv4_async_set_packets_to_drain()`

This message helper function sets a maximum number of ICMP error messages to send each time the queue processing event starts. The value to `arg_PacketsToDrain` must not exceed the depth of the queue.

C Syntax

```
ix_error ix_cc_ipv4_async_set_packets_to_drain (
    ix_uint16 arg_PacketsToDrain);
```

Input

- | | |
|---------------------------------|--|
| <code>arg_PacketsToDrain</code> | A maximum number of packets to send each time the queue processing event occurs. |
|---------------------------------|--|

Output/Returns

- | | |
|--------------|---|
| Return Value | <p>Returns a valid <code>ix_error</code>.</p> <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeds. • <code>IX_CC_IPV4_ERROR_INVALID_PACKETS_TO_DRAIN</code>—the operation failed, due to unsupported packets to drain value. • <code>IX_CC_IPV4_ERROR_MSG_LIBRARY</code>—the operation failed, An error was signaled by the Message Support Library. |
|--------------|---|

15.3.21 `ix_cc_ipv4_async_get_statistics()`

This message helper function retrieves the statistical report from the IPv4 Forwarder core component. This function gives the statistics from the IPv4 microblock, the Route Table Manager, and the IPv4 Forwarder core component.

C Syntax

```
ix_error ix_cc_ipv4_async_get_statistics (
    ix_cc_ipv4_cb_get_statistics arg_Callback,
    void *arg_pUserContext);
```

Input

- | | |
|-------------------------------|--|
| <code>arg_Callback</code> | Pointer to the calling application-provided callback. |
| <code>arg_pUserContext</code> | Pointer to the calling application-defined context. See ix_cc_ipv4_cb_get_statistics . |

Output/Returns

- | | |
|--------------|---|
| Return Value | <p>Returns a valid <code>ix_error</code>.</p> <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeds. • <code>IX_CC_ERROR_NULL</code>—the operation failed, due to invalid pointer. • <code>IX_CC_IPV4_ERROR_MSG_LIBRARY</code>—the operation failed, An error was signaled by the Message Support Library. |
|--------------|---|

15.3.21.1 ix_cc_ipv4_cb_get_statistics

Function prototype of callback functions for calls to `ix_cc_ipv4_async_get_statistics()`.

C Syntax

```
ix_error (*ix_cc_ipv4_cb_get_statistics) (
    ix_error arg_Result,
    void* arg_pContext,
    ix_cc_ipv4_stats_data* arg_pBuffer
);
```

Input

<code>arg_Result</code>	Indicates error conditions for the call
<code>arg_pContext</code>	Pointer to the calling application-provided context, that was sent by API call. Used by the calling application to identify the instance of the request.
<code>arg_pBuffer</code>	Pointer to memory to hold statistics data.

15.4 Library API

Table 15-5 shows the IPV4 Forwarder library API.

Table 15-5. IPV4 Forwarder Library API

API	Description
<code>ix_cc_ipv4_add_route()</code>	Adds a route in the Route Table Manager (RTM)
<code>ix_cc_ipv4_delete_route()</code>	Deletes a route from the RTM
<code>ix_cc_ipv4_update_route()</code>	Updates an existing route in the RTM
<code>ix_cc_ipv4_lookup_route()</code>	Looks up the routing information for a given IP address
<code>ix_cc_ipv4_purge_routes()</code>	Removes all routes from the RTM
<code>ix_cc_ipv4_dump_routes()</code>	Dumps all routes of memory in the RTM
<code>ix_cc_ipv4_add_next_hop()</code>	Adds the next hop information to the RTM
<code>ix_cc_ipv4_delete_next_hop()</code>	Deletes the next hop information from the RTM
<code>ix_cc_ipv4_update_next_hop()</code>	Updates the next hop information into the RTM database
<code>ix_cc_ipv4_get_next_hop()</code>	Retrieves the next hop information from the RTM
<code>ix_cc_ipv4_dump_next_hops()</code>	Dumps all next hops of the RTM
<code>ix_cc_ipv4_purge_rtm()</code>	Internally calls <code>ix_cc_rtmv4_purge()</code> API of the RTM

Table 15-5. IPV4 Forwarder Library API (Continued)

API	Description
<code>ix_cc_ipv4_set_mtu()</code>	Calls <code>ix_cc_rtmv4_set_mtu()</code> API of the RTM
<code>ix_cc_ipv4_set_flags()</code>	Internally calls <code>ix_cc_rtmv4_set_flags()</code> API of the RTM
<code>ix_cc_ipv4_get_rtm_handle()</code>	Returns the handle to the RTMv4.
<code>ix_cc_ipv4_get_sleep_time()</code>	Retrieves the number of seconds allocated for calling the ICMP event handler, that is, messages dequeued and sent
<code>ix_cc_ipv4_set_sleep_time()</code>	Sets the number of seconds for calling the ICMP event handler
<code>ix_cc_ipv4_get_queue_depth()</code>	Retrieves the maximum number of ICMP messages that can fit in the queue
<code>ix_cc_ipv4_get_packets_to_drain()</code>	Retrieves the maximum number of ICMP error messages to send
<code>ix_cc_ipv4_set_packets_to_drain()</code>	Sets the maximum number of ICMP error messages to send
<code>ix_cc_ipv4_set_property()</code>	Sets the dynamic properties of the IPV4 Forwarder core component
<code>ix_cc_ipv4_get_statistics()</code>	Retrieves the statistical report from the IPV4 Forwarder core component.
<code>ix_cc_ipv4_set_property()</code>	Sets the dynamic properties of the IPV4 Forwarder core component

15.4.1 ix_cc_ipv4_add_route()

This library function adds a route in the Route Table Manager. This API internally calls `ix_cc_rtmv4_add_route()` API of the Route Table Manager core component to add route in the Route Table Manager database.

C Syntax

```
ix_error ix_cc_ipv4_add_route(
    ix_uint32 arg_IpAddr,
    ix_uint32 arg_NetMask,
    ix_cc_rtmv4_nhid arg_NextHopId,
    void* arg_pContext);
```

Input

<code>arg_IpAddr</code>	Network IP address identifying the route
<code>arg_NetMask</code>	Network mask for the route
<code>arg_NextHopId</code>	Next hop identifier for route
<code>arg_pContext</code>	Pointer to IPv4 core component context

Output/Returns

Return Value	<p>Returns a valid <code>ix_error</code>.</p> <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • <code>IX_CC_ERROR_ENTRY_NOT_FOUND</code>—the operation failed, due to invalid next hop identifier • <code>IX_CC_ERROR_DUPLICATE_ENTRY</code>—the operation failed, route was already added to this NHID • <code>IX_CC_ERROR_CONFLICTING_ENTRY</code>—the operation failed, route was already added to another NHID • <code>IX_CC_IPV4_ERROR_RTM</code>—the operation failed, due to RTM internal failure
--------------	---

15.4.2 ix_cc_ipv4_delete_route()

This library function deletes a route from the Route Table Manager. This API internally calls `ix_cc_rtmv4_delete_route()` API of the Route Table Manager core component to delete route from the Route Table Manager database.

C Syntax

```
ix_error ix_cc_ipv4_delete_route(  
    ix_uint32 arg_IpAddr,  
    ix_uint32 arg_NetMask,  
    void* arg_pContext);
```

Input

<code>arg_IpAddr</code>	Network IP address identifying the route.
<code>arg_NetMask</code>	Network mask for the route.
<code>arg_pContext</code>	Pointer to IPv4 core component context

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none">• <code>IX_SUCCESS</code>—the operation succeeded.• <code>IX_CC_ERROR_ENTRY_NOT_FOUND</code>—the operation failed, the network address and mask pair do not exist in the Route Table Manager• <code>IX_CC_IPV4_ERROR_RTM</code>—the operation failed, due to Route Table Manager internal failure
--------------	---

15.4.3 ix_cc_ipv4_update_route()

This library function updates an existing route in the Route Table Manager. This API internally calls `ix_cc_rtmv4_update_route()` API of the Route Table Manager core component to update route in the RTM database.

C Syntax

```
ix_error ix_cc_ipv4_update_route(  
    ix_uint32 arg_IpAddr,  
    ix_uint32 arg_NetMask,  
    ix_cc_rtmv4_nhid arg_NextHopId,  
    void* arg_pContext);
```

Input

<code>arg_IpAddr</code>	Network IP address identifying the route to be updated.
-------------------------	---

Input (Continued)

<code>arg_NetMask</code>	Network mask for the route to be updated.
<code>arg_NextHopId</code>	Next hop identifier for the route to be updated.
<code>arg_pContext</code>	Pointer to IPv4 core component context.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. <code>IX_CC_ERROR_ENTRY_NOT_FOUND</code>—the operation failed, entry not in database <code>IX_CC_IPV4_ERROR_INVALID_MASK</code>—the operation failed, due to invalid mask <code>IX_CC_ERROR_CONFLICTING_ENTRY</code>—the operation failed, due to route was already added to another NHID <code>IX_CC_IPV4_ERROR_RTM</code>—the operation failed, due to RTM failure
--------------	---

15.4.4 `ix_cc_ipv4_lookup_route()`

This function looks up the routing information for a given IP address. If there is a successful match, then the information given to the calling application are the `nextHopId` (including the blade identifier), `portId`, `MTU`, and flags. If there is no match, then `IX_CC_RTMV4_NHID_NO_ROUTE` is assigned to the `nextHopId` field of `pNextHopData` and all other information is invalid.

C Syntax

```
ix_error ix_cc_ipv4_lookup_route (
    ix_uint32 arg_IpAddr,
    ix_cc_rtmv4_next_hop_info* arg_pNextHopInfo,
    void* arg_pContext);
```

Input

<code>arg_IpAddr</code>	Network IP address identifying the route
<code>arg_pContext</code>	Pointer to IPv4 core component context.

Output/Returns

<code>arg_pNextHopInfo</code>	The structure defined by the Route Table Manager core component. The result of the lookup is placed into the structure.
Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • <code>IX_CC_ERROR_NULL</code>—the operation failed, due to invalid pointer • <code>IX_CC_IPV4_ERROR_RTM</code>—the operation failed, due to Route Table Manager internal failure

15.4.5 `ix_cc_ipv4_purge_routes()`

This function removes all routes from the Route Table Manger.

C Syntax

```
ix_error ix_cc_ipv4_purge_routes (void* arg_pContext);
```

Input

`arg_pContext` Pointer to IPv4 core component context.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • <code>IX_CC_IPV4_ERROR_RTM</code>—the operation failed, due to Route Table Manager internal failure • <code>IX_CC_ERROR_NULL</code>—the operation failed, due to null pointer • <code>IX_CC_ERROR_ALIGN</code>—the operation failed, due to pointer not aligned properly
--------------	--

15.4.6 `ix_cc_ipv4_dump_routes()`

This function dumps all routes of memory in the Route Table Manager. The calling application uses `IX_CC_RTMV4_DUMP_ROUTE_SIZE` to determine the number of bytes to allocate.

C Syntax

```
ix_error ix_cc_ipv4_dump_routes (
    char* arg_pBuffer,
    ix_uint32 arg_Size,
    void* arg_pContext);
```


Input

<code>arg_pBuffer</code>	Pointer to a block of memory where routes are to be stored.
<code>arg_Size</code>	Size of the buffer in bytes.
<code>arg_pContext</code>	Pointer to IPv4 core component context.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. <code>IX_CC_IPV4_ERROR_RTM</code>—the operation failed, due to Route Table Manager internal failure <code>IX_CC_ERROR_NULL</code>—the operation failed, due to invalid pointer <code>IX_CC_ERROR_ALIGN</code>—the operation failed, due to pointer not aligned properly
--------------	---

15.4.7 `ix_cc_ipv4_add_next_hop()`

This function adds the next hop information to the Route Table Manager database. The calling application needs to add the next hop to the Route Table Manager in order to add routes referring to the next hop.

C Syntax

```
ix_error ix_cc_ipv4_add_next_hop (
    ix_cc_rtmv4_nhid arg_NextHopId,
    ix_cc_rtmv4_next_hop_info* arg_pNextHopInfo,
    void* arg_pContext);
```

Input

<code>arg_NextHopId</code>	Identifier of next hop to be added
<code>arg_pNextHopInfo</code>	Section 23.1.1.3, “ix_cc_rtmv4_next_hop_info”
<code>arg_pContext</code>	Pointer to IPv4 core component context

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • <code>IX_CC_ERROR_DUPLICATE_ENTRY</code>—the operation failed, due to the NHID was already added to this NHID • <code>IX_CC_ERROR_CONFLICTING_ENTRY</code>—the operation failed, the next hop was already added to another NHID. • <code>IX_CC_IPV4_ERROR_RTM</code> —the operation failed, due to RTM internal error • <code>IX_CC_ERROR_NULL</code>—the operation failed, due to invalid pointer • <code>IX_CC_IPV4_ERROR_INVALID_NH_INFO</code>—the operation failed, due to invalid next hop information
--------------	--

15.4.8 `ix_cc_ipv4_delete_next_hop()`

This library function deletes the next hop information from the Route Table Manager. This API internally calls `ix_cc_rtmv4_delete_next_hop()` API of the Route Table Manager core component to delete next hop information from the Route Table Manager.

C Syntax

```
ix_error ix_cc_ipv4_delete_next_hop(
    ix_cc_rtmv4_nhid arg_NextHopId,
    void* arg_pContext);
```

Input

<code>arg_NextHopId</code>	Identifier of next hop to be removed.
<code>arg_pContext</code>	Pointer to IPv4 core component context

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • <code>IX_CC_IPV4_ERROR_ENTRY_NOT_FOUND</code>—the operation failed, NHID was not able to find in Route Table Manager • <code>IX_CC_IPV4_ERROR_NEXT_HOP_ID_IN_USE</code>—the operation failed, NHID is in use • <code>IX_CC_IPV4_ERROR_RTM</code>—the operation failed, due to Route Table Manager internal failure
--------------	--

15.4.9 ix_cc_ipv4_update_next_hop()

This function updates next hop information into the RTM database. The calling application needs to add a next hop to the RTM in order to add routes referring to the next hop.

C Syntax

```
ix_error ix_cc_ipv4_update_next_hop(
    ix_cc_rtmv4_nhid arg_NextHopId,
    ix_cc_rtmv4_next_hop_info* arg_pNextHopInfo,
    void* arg_pContext);
```

Input

<code>arg_NextHopId</code>	Identifier of the next hop to be updated.
<code>arg_pNextHopInfo</code>	defined in Section 23.1.1.3 , “ <code>ix_cc_rtmv4_next_hop_info</code> ”
<code>arg_pContext</code>	Pointer to IPv4 core component context.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. <code>IX_CC_IPV4_ERROR_INVALID_NH_INFO</code>—the operation failed, due to invalid next hop information. <code>IX_CC_IPV4_ERROR_RTM</code>—the operation failed, due to RTM internal error <code>IX_CC_ERROR_NULL</code>—the operation failed, due to invalid pointer <code>IX_CC_ERROR_ENTRY_NOT_FOUND</code>—the operation failed, due to invalid next hop id
--------------	--

15.4.10 ix_cc_ipv4_get_next_hop()

This library function retrieves the next hop information from the Route Table Manager. This API internally calls `ix_cc_rtmv4_get_next_hop()` API of the Route Table Manager core component to retrieve the next hop information.

C Syntax

```
ix_error ix_cc_ipv4_get_next_hop(
    ix_cc_rtmv4_nhid arg_NextHopId,
    ix_cc_rtmv4_next_hop_info* arg_pNextHopInfo,
    void* arg_pContext);
```

Input

<code>arg_NextHopId</code>	Identifier of the next hop information to be retrieved from the Route Table Manager.
----------------------------	--

Input (Continued)

`arg_pContext` Pointer to IPv4 core component context.

Output/Returns

`arg_pNextHopInfo` Result is placed into this structure.

Return Value Returns a valid `ix_error`.

- `IX_SUCCESS`—the operation succeeded.
- `IX_CC_IPV4_ERROR_RTM`—the operation failed, due to RTM internal failure
- `IX_CC_ERROR_NULL`—the operation failed, due to invalid pointer
- `IX_CC_ERROR_ENTRY_NOT_FOUND`—the operation failed, due to invalid next hop id

15.4.11 `ix_cc_ipv4_dump_next_hops()`

This function dumps all next hops of the Route Table Manager in memory. This API internally calls `ix_cc_rtmv4_dump_next_hops()` API of the Route Table Manager core component. The calling application uses `IX_CC_RTMV4_DUMP_NEXT_HOP_SIZE` to determine the number of bytes to allocate.

C Syntax

```
ix_error ix_cc_ipv4_dump_next_hops(
    char* arg_pBuffer,
    ix_uint32 arg_Size,
    void* arg_pContext);
```

Input

`arg_pBuffer` Pointer to a block of memory where next hops are to be stored.

`arg_Size` Size of buffer in bytes

`arg_pContext` Pointer to IPv4 core component context.

Output/Returns

`arg_pNextHopInfo` Result is placed into this structure.

Output/Returns

Return Value

Returns a valid `ix_error`.

- `IX_SUCCESS`—the operation succeeded.
- `IX_CC_ERROR_NULL`—the operation failed, due to invalid pointer
- `IX_CC_ERROR_ALIGN`—the operation failed, due to pointer not aligned properly
- `IX_CC_ERROR_FULL`—the operation failed, due to buffer is too small

15.4.12 ix_cc_ipv4_purge_rtm()

This function removes all routes and next hops from the Route Table Manager database and internally calls `ix_cc_rtmv4_purge()` API of the Route Table Manager core component.

C Syntax

```
ix_error ix_cc_ipv4_purge_rtm (void* arg_pContext);
```

Input

`arg_pContext` Pointer to IPv4 core component context.

Output/Returns

Return Value	<p>Returns a valid <code>ix_error</code>.</p> <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • <code>IX_CC_IPV4_ERROR_RTM</code>—the operation failed, due to Route Table Manager internal failure
--------------	---

15.4.13 ix_cc_ipv4_set_mtu()

This function updates the MTU for a given next hop. Before calling this message helper, the calling application must call `ix_cc_ipv4_async_add_next_hop()` to add the next hop identifier in the Route Table Manager database. The MTU is then validated by the calling application.

This API internally calls `ix_cc_rtmv4_set_mtu()` API of the Route Table Manager core component.

C Syntax

```
ix_error ix_cc_ipv4_set_mtu (
    ix_cc_rtmv4_nhid arg_NextHopId,
    ix_uint32 arg_Mtu,
    void* arg_pContext);
```

Input

<code>arg_NextHopId</code>	Identifier of next hop.
<code>arg_Mtu</code>	New maximum transmission unit for the given next
<code>arg_pContext</code>	Pointer to IPv4 core component context.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • <code>IX_CC_ERROR_ENTRY_NOT_FOUND</code>—the operation failed, NHID was not able to find the Route Table Manager • <code>IX_CC_IPV4_ERROR_INVALID_NH_INFO</code>—the operation failed, due to invalid mtu • <code>IX_CC_IPV4_ERROR_RTM</code>—the operation failed, due to Route Table Manager internal failure
--------------	---

15.4.13.1 `ix_cc_ipv4_set_flags()`

This function updates the flags for a given next hop. Before calling this message helper, the calling application must call `ix_cc_ipv4_async_add_next_hop()` to add the next hop identifier in the Route Table Manager database. The supported flags are defined by the IPv4 Forwarder microblock.

C Syntax

```
ix_error ix_cc_ipv4_set_flags(
    ix_cc_rtmv4_nhid arg_NextHopId,
    ix_uint32 arg_Flags,
    void* arg_pContext);
```

Input

<code>arg_NextHopId</code>	Identifier of the next hop
<code>arg_Flags</code>	New flags[BLDBLK] for the given next hop
<code>arg_pUserContext</code>	Pointer to the calling application-defined context.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • <code>IX_CC_ERROR_ENTRY_NOT_FOUND</code>—the operation failed, due to invalid next hop id • <code>IX_CC_IPV4_ERROR_INVALID_FLAGS</code>—the operation failed, due to unsupported flags • <code>IX_CC_IPV4_ERROR_RTM</code>—the operation failed, due to RTM internal failure
--------------	--

15.4.14 ix_cc_ipv4_get_rtm_handle()

This function returns the handle to the RTMv4.

C Syntax

```
ix_error ix_cc_ipv4_get_rtm_handle (
    ix_cc_rtmv4 *arg_phRtmv4);
```

Input

None

Output/Returns

<code>arg_phRtm</code>	The handle to the RTMv4 that was created during initialization.
Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—returned if the operation succeeds. • A valid <code>ix_error</code>—returned if the operation fails

15.4.15 ix_cc_ipv4_get_sleep_time()

This library function retrieves the number of seconds allocated for calling the ICMP event handler, that is, messages dequeued and sent. The lower the value of `arg_pSleepTime`, higher the frequency of the ICMP message process (dequeued and sent).

C Syntax

```
ix_error ix_cc_ipv4_get_sleep_time(
    ix_uint16* arg_pSleepTime,
    void* arg_pContext);
```

Input

<code>arg_SleepTime</code>	Sleep period currently set.
<code>arg_pContext</code>	Pointer to IPv4 core component context.

Output

<code>arg_pSleepTime</code>	The sleep period currently set.
Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • <code>IX_ERROR_ERROR_NULL</code>—the operation failed, due to invalid pointer.

15.4.16 ix_cc_ipv4_set_sleep_time()

This library function sets the period for calling the ICMP error handler. The lower the value of `arg_SleepTime`, higher the frequency of the output queue process. The minimum value of `arg_SleepTime` is 1 second and maximum value can be 32 seconds.

C Syntax

```
ix_error ix_cc_ipv4_set_sleep_time(
    ix_uint16 arg_SleepTime,
    void* arg_pContext);
```

Input

<code>arg_SleepTime</code>	The number of seconds to wait between processing of the ICMP error message queue.
<code>arg_pContext</code>	Pointer to IPv4 core component context.

Output

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. <code>IX_CC_IPV4_ERROR_INVALID_SLEEP_TIME</code>—the operation failed, due to unsupported sleep time.
--------------	---

15.4.17 ix_cc_ipv4_get_queue_depth()

This library function retrieves the depth of the ICMP error message queue, that is, the maximum number of ICMP messages that can fit in the queue, not the number of messages currently in the queue.

C Syntax

```
ix_error ix_cc_ipv4_get_queue_depth(
    ix_uint16* arg_pQueueDepth,
    void* arg_pContext);
```

Input

<code>arg_pContext</code>	Pointer to IPv4 core component context.
---------------------------	---

Output

<code>arg_pQueueDepth</code>	The current number of entries that the ICMP error message queue can hold.
Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. <code>IX_CC_ERROR_NULL</code>—the operation failed, due to invalid pointer

15.4.17. `ix_cc_ipv4_get_packets_to_drain()`

This library function retrieves the maximum number of ICMP error messages to send (from the ICMP message queue) each time the queue processing event fires.

C Syntax

```
ix_error ix_cc_ipv4_get_packets_to_drain (
    ix_uint16* arg_pPacketsToDrain,
    void* arg_pContext);
```

Input

`arg_pContext` Pointer to IPv4 core component context.

Output

<code>arg_pPacketsToDrain</code>	The maximum number of packets to send each time the event fires for processing the queue.
Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. <code>IX_CC_ERROR_NULL</code>—the operation failed, due to invalid pointer

15.4.18 `ix_cc_ipv4_set_packets_to_drain()`

This library function sets the maximum number of ICMP error messages to send each time the queue processing event is executed. The value of `arg_PacketsToDrain` must not exceed the depth of the queue.

C Syntax

```
ix_error ix_cc_ipv4_set_packets_to_drain (
    ix_uint16 arg_PacketsToDrain,
    void* arg_pContext);
```

Input

<code>arg_PacketsToDrain</code>	The maximum number of packets to send each time the queue processing event fires.
<code>arg_pContext</code>	Pointer to IPv4 core component context.

Output

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. <code>IX_CC_IPV4_ERROR_INVALID_PACKETS_TO_DRAIN</code>—the operation failed, due to unsupported packets to drain value
--------------	--

15.4.19 `ix_cc_ipv4_get_statistics()`

This library function retrieves the statistical report from the IPv4 Forwarder core component. This function provides the statistics of the IPV4 microblock, the Route Table Manager and the IPv4 Forwarder core component.

C Syntax

```
ix_error ix_cc_ipv4_get_statistics (
    ix_cc_ipv4_stats_data* arg_pBuffer,
    void* arg_pContext);
```

Input

<code>arg_pContext</code>	Pointer to IPv4 core component context.
---------------------------	---

Output/Returns

<code>arg_pBuffer</code>	Pointer to memory to hold statistics data.
Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. <code>IX_CC_ERROR_NULL</code>—the operation failed, due to invalid pointer. <code>IX_CC_IPV4_ERROR_RTM</code>—the operation failed, due to RTM failure <code>IX_CC_ERROR_INTERNAL</code>—the operation failed, due to internal failure

15.4.20 ix_cc_ipv4_set_property()

This library function sets the dynamic properties of the IPv4 Forwarder core component.

C Syntax

```
ix_error ix_cc_ipv4_set_property (
    ix_uint32 arg_PropId,
    ix_cc_properties* arg_pProperties,
    void* arg_pContext);
```

Input

arg_PropId	Identifier of a property or properties(ORed) to be updated for a port.
arg_pProperties	Pointer to dynamic property data structure.
arg_pContext	Pointer to IPv4 core component context

Output/Returns

arg_PropId	Identifier of a property or properties (ORed) to be updated for a port.
arg_pProperties	Pointer to dynamic property data structure.
Return Value	<p>Returns a valid <code>ix_error</code>.</p> <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • <code>IX_CC_IPV4_ERROR_INVALID_INPUT_PARAM</code>—the operation failed, due to invalid property identifier • <code>IX_CC_ERROR_NULL</code>—the operation failed, due to invalid pointer • <code>IX_CC_ERROR_FULL</code>—the operation failed, due to table is full • <code>IX_CC_ERROR_ENTRY_NOT_FOUND</code>—the operation failed, due to entry not found in table

The IPv6 core component performs the following functions:

- Configures IPv6 microblock (static configuration)
- Provides message and packet handlers to receive messages and packets from other core components and IPV6 microblock.
- Generates ICMPv6 error messages
- Validates IP header
- Supports handling of extension headers
- Supports Neighbor Discovery
- Supports Stateless Address Auto Configuration

For complete details see [Chapter 53, “IPv6 Forwarder Core Component”](#) of the *Intel® Internet Exchange Architecture Software Building Blocks Developer’s Manual*.

16.1 Data Structures, Types and Macros

Table 16-1 lists the data structures, types and macros defined in IPv6 Forwarder core component.

Table 16-1. IPv6 Forwarder Data Structures, Types and Macros

Data Structures, Types and Macros	Description
<code>IX_CC_RTMV6_DUMP_ROUTE_SIZE</code>	Defines a macro for calculating the size of memory to dump RTMv6 routes
<code>ix_cc_rtmv6_nhid</code>	Defines the data type for next hop identifier
<code>Reserved Next Hop Ids</code>	Defines the special next hops added during initialization
<code>ix_cc_rtmv6_next_hop_info</code>	Defines the data structure for next hop information
<code>ix_cc_ipv6_dump_data</code>	Describes memory dump information.
<code>ix_cc_in_ipv6_stats_data</code>	Defines data structure for statistics
<code>ix_cc_out_ipv6_stats_data</code>	Defines data structure for statistics
<code>ix_cc_ipv6_stats_data</code>	Defines counter information.
<code>ix_cc_ipv6_icmp_err_type</code>	Enumeration of the ICMPv6 error message types
<code>ix_cc_ipv6_icmp_err_code</code>	Enumeration of the ICMPv6 codes to accompany the error message types

16.1.1 IX_CC_RTMV6_DUMP_ROUTE_SIZE

This macro is defined for calculating the size of memory to dump RTMv6 routes It is defined in [Section 24.1.8, “IX_CC_RTMV6_DUMP_ROUTE_SIZE”](#) on page 516.

16.1.2 ix_cc_rtmv6_nhid

Typedef for next hop identifier. It is defined in [Section 24.1.2](#), “ix_cc_rtmv6_nhid” on page 514.

16.1.3 Reserved Next Hop Ids

IPv6 Forwarder core component needs to add these special next hops during initialization.

C Syntax

```
#define IX_CC_IPV6_NH_NO_ROUTE -1 /*No route exists for this destination */
#define IX_CC_IPV6_NH_DROP -2 /* Drop all packets for this destination */
#define IX_CC_IPV6_NH_LOCAL -3 /* All packets matching this ID are for
                                local delivery. */
```

16.1.4 ix_cc_rtmv6_next_hop_info

This is the Structure definition for next hop information, defined in [Section 24.1.3](#), “ix_cc_rtmv6_next_hop_info” on page 514.

Note: Only one flag can be set at a time.

16.1.5 ix_cc_ipv6_dump_data

The IPv6 Forwarder core component defines the structure describing memory dump information.

C Syntax

```
typedef struct ix_s_cc_IPv6_dump_data
{
    char* pBuffer;
    ix_uint32 size;
} ix_cc_ipv6_dump_data;
```

16.1.6 ix_cc_in_ipv6_stats_data

The IPv6 forwarder statistics are defined by the following structures. Fields marked “Microblock only” and “Core only” are only updated by the microblock and core respectively. Fields marked “Core and Microblock” are updated by both, the core and microblock. The “Core and Microblock” counters are 32-bits in length and support atomic operations. In addition, statistics for RTMv6 are also reported.

C Syntax

```
typedef struct ix_s_cc_in_ipv6_stats_data
{
    /* Microblock only */
    ix_uint64 ipv6MbInReceives;
    ix_uint64 ipv6MbInOctets;
    ix_uint32 ipv6MbInForwDatagrams;
    ix_uint32 ipv6MbInAddrErrors;
    ix_uint32 ipv6MbInTruncated;

    /* Core and Microblock */
    ix_uint32 ipv6CoreMbInHdrErrors;

    /* Core only */
    ix_uint32 ipv6CoreInNoRoutes;
    ix_uint32 ipv6CoreInUnknownProtos;
    ix_uint32 ipv6CoreInDiscards;
    ix_uint32 ipv6CoreInDelivers;
    ix_uint64 ipv6CoreInMulticast;
    ix_uint64 ipv6CoreInMulticastOctets;
} ix_cc_ipv6_in_stats_data;
```

16.1.7 ix_cc_out_ipv6_stats_data

See description above in [Section 16.1.6](#), “ix_cc_in_ipv6_stats_data”.

C Syntax

```
typedef struct ix_s_cc_out_ipv6_stats_data
{
    /* Microblock only */
    ix_uint32 ipv6MbOutForwDatagrams;
    ix_uint64 ipv6MbOutTransmits;
    ix_uint64 ipv6MbOutOctets;

    /* Core only */
    ix_uint32 ipv6CoreOutRequests;
    ix_uint32 ipv6CoreOutNoRoutes;
    ix_uint32 ipv6CoreOutDiscards;
    ix_uint64 ipv6CoreOutMulticast;
    ix_uint64 ipv6CoreOutMulticastOctets;
```

```
} ix_cc_out_ipv6_stats_data;
```

16.1.8 ix_cc_ipv6_stats_data

The IPv6 Forwarder core component defines the structure describing counter information.

C Syntax

```
typedef struct ix_s_cc_IPv6_stats_data
{
    /* Incoming stats */
    ix_cc_in_ipv6_stats_data
        ipv6InStats[MAX_NUMBER_OF_PORTS_PER_BLADE];

    /* Outgoing stats.
     * TOTAL_NUMBER_OF_PORTS =
     *     (MAX_NUMBER_OF_PORT_PER_BLADE * MAX_NUMBER_OF_BLADES)
     */
    ix_cc_out_ipv6_stats_data ipv6OutStats[TOTAL_NUMBER_OF_PORTS];

    /* RTMv6 Stats */
    ix_uint32 ipv6NumberOfRoutes;
    ix_uint32 ipv6NumberOfNextHops;
} ix_cc_ipv6_stats_data;
```

16.1.9 ix_cc_ipv6_icmp_err_type

Enumeration of the ICMPv6 error message types.

C Syntax

```
typedef enum ix_e_cc_ipv6_icmp_err_type{
    IX_CC_IPV6_ICMP_ERR_DEST_UNREACHABLE,
    IX_CC_IPV6_ICMP_ERR_PKT_TOO_BIG,
    IX_CC_IPV6_ICMP_ERR_TIME_EXCEEDED,
    IX_CC_IPV6_ICMP_ERR_PARAM_PROBLEM
}ix_cc_ipv6_icmp_err_type;
```

16.1.10 ix_cc_ipv6_icmp_err_code

Enumeration of the ICMPv6 codes to accompany the error message types.

C Syntax

```
typedef enum ix_e_cc_ipv6_icmp_err_code(
    IX_CC_IPV6_ICMP_CODE_DU_NO_ROUTE,
    IX_CC_IPV6_ICMP_CODE_DU_COMM_PROHIBITED,
    IX_CC_IPV6_ICMP_CODE_DU_ADDR_UNREACHABLE,
    IX_CC_IPV6_ICMP_CODE_DU_PORT_UNREACHABLE,
    IX_CC_IPV6_ICMP_CODE_PKT_TOO_BIG,
```



```
IX_CC_IPV6_ICMP_CODE_TE_HOP_LIMIT_EXCEEDED,
IX_CC_IPV6_ICMP_CODE_TE_REASSEMBLY_EXCEEDED,
IX_CC_IPV6_ICMP_CODE_PP_WRONG_HDR_FIELD,
IX_CC_IPV6_ICMP_CODE_PP_NEXT_HDR_UNKNOWN,
IX_CC_IPV6_ICMP_CODE_PP_OPTION_UNKNOWN
} ix_e_cc_ipv6_icmp_err_code;
```

Note:

- Destination Unreachable messages MUST use codes containing “DU”.
- Timer Exceeded messages MUST use codes containing “TE”.
- Parameter Problem messages MUST use codes containing “PP”.
- The Packet-Too-Big message MUST use the code—
IX_CC_IPV6_ICMP_CODE_PKT_TOO_BIG.

16.2 Core Component Infrastructure API

Table 16-2 lists the IPV6 Forwarder core component Infrastructure API.

Table 16-2. IPV6 Forwarder Core Component Infrastructure API

API	Description
<code>ix_cc_ipv6_init()</code>	Initializes the IPV6 Forwarder component
<code>ix_cc_ipv6_fini()</code>	Terminates services from the IPV6 Forwarder
<code>ix_cc_ipv6_msg_handler()</code>	Message handler function for IPV6 Forwarder
<code>ix_cc_ipv6_microblock_high_priority_pkt_handler()</code>	Receives exception packets from high priority queue of IPV6 microblock
<code>ix_cc_ipv6_microblock_low_priority_pkt_handler()</code>	Receive exception packets from IPV6 microblock
<code>ix_cc_ipv6_stackdrv_pkt_handler()</code>	Receives packets from stack driver core component
<code>ix_cc_ipv6_common_pkt_handler()</code>	Receive packets from any core component other than stack driver—interface Rx core component

16.2.1 `ix_cc_ipv6_init()`

This function is called and returned successfully before requesting any service from IPV6 Forwarder component, and should be called only once to initialize the IPV6 Forwarder core component. This function performs the following tasks:

- allocates memory for symbols to be patched
- creates 64-bit counters
- registers packet and message handlers
- initializes and configures RTMv6
- allocates and initializes internal data structures
- creates event handler for ICMP

The calling application needs to initialize the IXA software framework, Core Components Infrastructure before calling this primitive.

C Syntax

```
ix_error ix_cc_ipv6_init (
    ix_cc_handle arg_CcHandle,
    void** arg_ppContext);
```

Input

<code>arg_CcHandle</code>	Handle to IPv6 core component, created by the Core Component Infrastructure. This is used later to get other (to add event handler) services from the Core Component Infrastructure.
---------------------------	--

Input/Output

<code>arg_ppContext</code>	Location where the pointer to the control block allocated by IPv6 Forwarder core component will be stored. The control block is internal to IPv6 Forwarder core component and it contains information about IPv6 internal data structures, allocated memory and other relevant information. Later this pointer is passed into the <code>ix_cc_ipv6_fini</code> function for freeing memory and other house keeping operations when IPv6 component is being destroyed.
----------------------------	---

Output/Returns

Return Value	<p>Returns a valid <code>ix_error</code>.</p> <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • <code>IX_CC_ERROR_NULL</code>—operation failed, null input parameter • <code>IX_CC_ERROR_OOM</code>—operation failed, memory allocation failure • <code>IX_CC_IPV6_ERROR_FAILED_PATCHING</code>—operation failed, patching failures • <code>IX_CC_ERROR_OOM_64BIT_COUNTER</code>—operation failed, 64 bit counter creation failure • <code>IX_CC_IPV6_ERROR_REGISTRY</code>—operation failed, information from registry is invalid • <code>IX_CC_IPV6_ERROR_RTM</code>—operation failed, failure from RTMv6 core component • <code>IX_CC_IPV6_ERROR_CCI</code>—operation failed, failure from the Core Component Infrastructure
--------------	---

16.2.2 ix_cc_ipv6_fini()

This function is called to terminate services from the IPv6 Forwarder core component. This primitive frees memory allocated during initialization, shuts down RTMv6, and frees all the created resources—deletes 64 bit counter, deletes entries from directed broadcast table.

The calling application must stop the microengines before calling this function. If the calling application claims any services from the IPv6 Forwarder core component after successful completion of this function, then the behavior is undefined.

C Syntax

```
ix_error ix_cc_ipv6_fini (
    ix_cc_handle arg_CcHandle,
    void* arg_pContext);
```

Input

arg_CcHandle	Handle to the core component.
arg_pContext	Pointer to the control block memory allocated earlier in ix_cc_ipv6_init function.

Output/Returns

Return Value	<p>Returns a valid ix_error.</p> <ul style="list-style-type: none"> IX_SUCCESS—the operation succeeded. IX_CC_IPV6_ERROR_INVALID_INPUT_PARAM—operation failed, invalid data in arg_pContext IX_CC_ERROR_OOM—operation failed, memory free failure IX_CC_IPV6_ERROR_RTM—operation failed, failure from RTMv6 core component IX_CC_IPV6_ERROR_CCI—operation failed, failure from Core Component Infrastructure IX_CC_ERROR_OOM_64BIT_COUNTER—operation failed, 64 bit counter deletion failure
--------------	--

16.2.3 ix_cc_ipv6_msg_handler()

This function is the message handler for the IPv6 Forwarder core component. The IPV6 Forwarder core component receives messages from other core components through this message handler function and then internally calls the appropriate library function to process the message.

This message handler is used to update dynamic properties, defined in [Section 2.2.1, “Dynamic Properties and Clients” on page 33](#). Communication between the calling application and the core components for message passing and retrieving results is described in [Chapter 26, “Message Helper and Support Library”](#).

C Syntax

```
ix_error ix_cc_ipv6_msg_handler (
    ix_buffer_handle arg_hDataToken,
    ix_uint32 arg_UserData
    void* arg_pComponentContext);
```

Input

arg_hDataToken	Buffer handle embedding information for the message passed in arg_UserData.
arg_UserData	Message type. Table 16-3 lists the IPV6 Forwarder supported messages.
arg_pComponentContext	Pointer to the IPv6 Forwarder core component context that is passed to the core component when a message arrives. This context is defined by the core component and passed to the Core Components Infrastructure through ix_cc_ipv6_init() .

Table 16-3. IPv6 Forwarder Messages

Messages	Description
IX_CC_IPV6_MSG_ADD_PREFIX	This message helps client to add a route into RTMv6.
IX_CC_IPV6_MSG_DELETE_PREFIX	This message helps client to delete a route from RTMv6.
IX_CC_IPV6_MSG_LOOKUP_PREFIX	This message helps client to lookup route information for a given IP address.
IX_CC_IPV6_MSG_PURGE_PREFIX	This message helps client to delete all routes from RTMv6.
IX_CC_IPV6_MSG_DUMP_PREFIX	This message helps to dump all routes in memory.
IX_CC_IPV6_MSG_ADD_NEXT_HOP	This message helps client to add a next hop into RTMv6.
IX_CC_IPV6_MSG_DELETE_NEXT_HOP	This message helps client to delete a next hop from RTMv6.
IX_CC_IPV6_MSG_GET_NEXT_HOP	This message helps client to retrieve next hop information from RTMv6.

Table 16-3. IPv6 Forwarder Messages

Messages	Description
IX_CC_IPV6_MSG_DUMP_NEXT_HOPS	This message helps to dump all next hops in memory.
IX_CC_IPV6_MSG_PURGE_RTM	This message helps client to delete all routes and next hops from RTMv6.
IX_CC_IPV6_MSG_SET_MTU	This message helps client to set mtu for a next hop.
IX_CC_IPV6_MSG_SET_FLAGS	This message helps client to set flags for a next hop.
IX_CC_IPV6_MSG_GET_RATE_LIMIT_TIME	This message helps client to get icmp rate limiting time interval.
IX_CC_IPV6_MSG_SET_RATE_LIMIT_TIME	This message helps client to set icmp rate limiting time interval.
IX_CC_IPV6_MSG_GET_QUEUE_DEPTH	This message helps client to get icmp queue depth.
IX_CC_IPV6_MSG_GET_STATISTICS	This message helps client to get IPV6 statistics.
IX_CC_IPV6_MSG_PERFORM_ADDR_RESOLUTION	This message helps client to request address resolution for a neighbor.
IX_CC_IPV6_MSG_ADD_NBR	This message helps client to add a neighbor's information to the L2 table.
IX_CC_IPV6_MSG_DEL_NBR	This message helps client to delete an entry from the L2 table.
IX_CC_IPV6_MSG_SEND_ICMP_ERROR	This message helps client to send an ICMP error message.
IX_CC_IPV6_MSG_SEND_ICMP_INFO	This message helps client to send an ICMP informational message.

Output/Returns

Return Value

Returns a valid `ix_error`.

- `IX_SUCCESS`—the operation succeeded.
- `IX_CC_ERROR_NULL`—operation failed, invalid pointer
- `IX_CC_ERROR_UNDEFINED_MSG`—operation failed, unsupported message
- `IX_CC_IPV6_ERROR_MSG_LIBRARY`—operation failed, error from message support library
- `IX_CC_IPV6_ERROR_BUFFER_FREE`—operation failed, error from resource manager (RM) for freeing buffer

16.2.4 ix_cc_ipv6_microblock_high_priority_pkt_handler()

This is the registered function to receive exception packets from high priority queue of the IPv6 microblock

C Syntax

```
ix_error ix_cc_ipv6_microblock_high_priority_pkt_handler (
    ix_buffer_handle arg_hDataToken,
    ix_uint32 arg_ExceptionCode,
    void* arg_pComponentContext)
```

Input

arg_hDataToken	Handle to a buffer which contains exception packets from IPv6 microblock.
arg_ExceptionCode	Will be ignored
arg_pComponentContext	Pointer to IPV6 Forwarder core component context.

Output/Returns

Return Value	Returns a valid ix_error. <ul style="list-style-type: none"> IX_SUCCESS—the operation succeeded. IX_CC_ERROR_SEND_FAIL—operation failed, error from CCI IX_CC_ERROR_UNDEFINED_EXCEP—operation failed, unsupported exception code IX_CC_IPV6_ERROR_RTM—operation failed, RTMv6 error
--------------	---

16.2.5 ix_cc_ipv6_microblock_low_priority_pkt_handler()

This is the registered function to receive exception packets from the IPv6 microblock. This function internally calls/performs different functions/operations based on the exception code for a given packet.

C Syntax

```
ix_error ix_cc_ipv6_microblock_low_priority_pkt_handler (
    ix_buffer_handle arg_hDataToken,
    ix_uint32 arg_ExceptionCode,
    void* arg_pComponentContext)
```

Input

<code>arg_hDataToken</code>	Handle to a buffer which contains exception packets from the IPv6 microblock.
<code>arg_ExceptionCode</code>	Exception codes generated by the IPv6 Forwarder microblock.
<code>arg_pComponentContext</code>	Pointer to the IPV6 Forwarder core component context.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. <code>IX_CC_ERROR_SEND_FAIL</code>—operation failed, error from the Core Component Infrastructure <code>IX_CC_ERROR_UNDEFINED_EXCEP</code>—operation failed, unsupported exception code <code>IX_CC_IPV6_ERROR_RTM</code>—operation failed, RTMv6 error
--------------	---

16.2.6 `ix_cc_ipv6_stackdrv_pkt_handler()`

This is the registered function to receive packets from the Stack Driver core component. Packets coming from the Stack Driver need special processing by the IPv6 Forwarder core component. For such packets, the IPv6 core component does not do Hop Limit decrement or IP header validation.

C Syntax

```
ix_error ix_cc_ipv4_stackdrv_pkt_handler (
    ix_buffer_handle arg_hDataToken,
    ix_uint32 arg_ExceptionCode,
    void* arg_pComponentContext)
```

Input

<code>arg_hDataToken</code>	Handle to a buffer which contains packet from the Stack Driver core component.
<code>arg_ExceptionCode</code>	Will be ignored
<code>arg_pComponnetContext</code>	Pointer to the IPV6 Forwarder core component context.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • <code>IX_CC_ERROR_SEND_FAIL</code>—operation failed, error from CCI • <code>IX_CC_ERROR_INTERNAL</code>—operation failed, internal error • <code>IX_CC_ERROR_ALIGN</code>—operation failed, pointer not aligned properly • <code>IX_CC_ERROR_NULL</code>—operation failed, invalid pointer
--------------	---

16.2.7 `ix_cc_ipv6_common_pkt_handler()`

This is the registered function to receive packets from any core component other than stack driver—interface RX core component.

C Syntax

```
ix_error ix_cc_ipv6_common_pkt_handler (
    ix_buffer_handle arg_hDataToken,
    ix_uint32 arg_ExceptionCode,
    void* arg_pComponentContext)
```

Input

<code>arg_hDataToken</code>	Handle to a buffer which contains packet from any core component other than the Stack Driver.
<code>arg_ExceptionCode</code>	Will be Ignored.
<code>arg_pComponentContext</code>	Pointer to the IPV6 Forwarder core component context.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • <code>IX_CC_ERROR_INTERNAL</code>—operation failed, internal error • <code>IX_CC_ERROR_ALIGN</code>—operation failed, pointer not aligned properly • <code>IX_CC_ERROR_NULL</code>—operation failed, invalid pointer • <code>IX_CC_ERROR_SEND_FAIL</code>—operation failed, error from Core Component Infrastructure
--------------	---

16.3 Message Helper API

Table 16-4 lists the IPv6 Forwarder message helper API.

Table 16-4. IPv6 Forwarder Message Helper API

API	Description
<code>ix_cc_ipv6_async_add_prefix()</code>	Adds a prefix (route) in RTMv6
<code>ix_cc_ipv6_async_delete_prefix()</code>	Deletes a prefix (route) in the RTMv6 database
<code>ix_cc_ipv6_async_update_prefix()</code>	Updates an existing route in the RTMv6 database
<code>ix_cc_ipv6_async_lookup_prefix()</code>	Looks up routing information for a given IP address
<code>ix_cc_ipv6_async_purge_prefixes()</code>	Removes all prefixes from the RTMv6 database
<code>ix_cc_ipv6_async_dump_prefixes()</code>	Dumps all prefixes stored by the RTMv6 database in memory
<code>ix_cc_ipv6_async_add_next_hop()</code>	Adds next hop information to NH database
<code>ix_cc_ipv6_async_delete_next_hop()</code>	Deletes the next hop information from the RTMv6 database
<code>ix_cc_ipv6_async_update_next_hop()</code>	Updates the next hop information into the RTMv6 database
<code>ix_cc_ipv6_async_get_next_hop()</code>	Retrieves the next hop information from the RTMv6 database
<code>ix_cc_ipv6_async_dump_next_hops()</code>	Dumps all next hops of the RTMv6 in memory
<code>ix_cc_ipv6_async_purge_rtm()</code>	Removes all prefixes and next hops from the RTMv6 database
<code>ix_cc_ipv6_async_set_mtu()</code>	Updates MTU for a given next hop
<code>ix_cc_ipv6_async_set_flags()</code>	Updates flags for a given next hop
<code>ix_cc_ipv6_async_get_rate_limit_time()</code>	Retrieves the time interval for rate limiting in milliseconds
<code>ix_cc_ipv6_async_set_rate_limit_time()</code>	Sets the rate limit interval in milliseconds
<code>ix_cc_ipv6_async_get_queue_depth()</code>	Retrieves the depth of the ICMP error message queue
<code>ix_cc_ipv6_async_get_statistics()</code>	Retrieves statistics report from IPv6 Forwarder core component
<code>ix_cc_ipv6_async_perform_addr_resolution()</code>	Requests address resolution to be performed for the destination IP address contained in the packet
<code>ix_cc_ipv6_async_add_neighbor()</code>	Updates the neighbor cache and add an entry into the L2 table
<code>ix_cc_ipv6_async_del_neighbor()</code>	Deleted the contents located at the supplied L2Index in the L2 table
<code>ix_cc_ipv6_sync_add_neighbor()</code>	Updates the neighbor cache and adds an entry into the L2 table in a synchronous manner.
<code>ix_cc_ipv6_sync_del_neighbor()</code>	Delete the contents of the L2Index in the L2 table in a synchronous manner.
<code>ix_cc_ipv6_async_send_icmp_error()</code>	Sends an ICMP error message
<code>ix_cc_ipv6_async_send_icmp_info_message()</code>	Sends ICMP informational messages

16.3.1 ix_cc_ipv6_async_add_prefix()

This message helper function adds a prefix (route) in RTMv6. The calling application needs to successfully add nextHopId to RTMv6 by calling [ix_cc_ipv6_async_add_next_hop\(\)](#) message helper function before sending this message to the IPv6 Forwarder core component.

C Syntax

```
ix_error ix_cc_ipv6_async_add_prefix (
    ix_uint128 arg_IpAddr,
    ix_uint8 arg_PrefixLen,
    ix_cc_rtmv6_nhid arg_NextHopId,
    ix_cc_ipv6_cb_route_op arg_Callback,
    void* arg_pUserContext);
```

Input

arg_IpAddr	Network IPv6 address identifying the prefix.
arg_PrefixLen	The prefix length.
arg_NextHopId	Next hop identifier for the prefix.
arg_Callback	Pointer to a calling application-provided callback function. See ix_cc_ipv6_cb_route_op .
arg_pUserContext	Pointer to a calling application-defined context.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> IX_SUCCESS—the operation succeeded. IX_ERROR_INVALID_POINTER—operation failed, invalid pointer IX_CC_IPV6_ERROR_MSG_LIBRARY—operation failed, failure from Message Support Library
--------------	--

16.3.1.1 ix_cc_ipv6_cb_route_op

The function prototype for message-handler callback function provided by the calling application.

C Syntax

```
ix_error (*ix_cc_ipv6_cb_route_op) (
    ix_error arg_Result,
    void* arg_pContext);
```

Input

<code>arg_Result</code>	Indicates error conditions for the call
<code>arg_pContext</code>	Pointer to user provided context that was sent by API call. Used by the caller to identify the instance of the request.

Note: This is a generic callback type, which is used for IPV6 message helpers that do not require data to be returned in the callback—only the result and context are relevant.

16.3.2 `ix_cc_ipv6_async_delete_prefix()`

This message helper function deletes a prefix (route) in RTMv6.

C Syntax

```
ix_error ix_cc_ipv6_async_delete_prefix (
    ix_uint128 arg_IpAddr,
    ix_uint8 arg_PrefixLen,
    ix_cc_ipv6_cb_route_op arg_Callback,
    void* arg_pUserContext);
```

Input

<code>arg_IpAddr</code>	Network IPv6 address identifying the route.
<code>arg_PrefixLen</code>	Prefix length for the route.
<code>arg_Callback</code>	Pointer to a calling application-provided callback function. See ix_cc_ipv6_cb_route_op .
<code>arg_pUserContext</code>	Pointer to a calling application-defined context.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. <code>IX_ERROR_INVALID_POINTER</code>—operation failed, invalid pointer <code>IX_CC_IPV6_ERROR_MSG_LIBRARY</code>—operation failed, failure from Message Support Library
--------------	---

16.3.3 ix_cc_ipv6_async_update_prefix()

This message helper function updates an existing route in RTMv6. For this call to be successful, the calling application needs to first successfully add next hop and the related route to RTMv6 by calling `ix_cc_ipv6_async_add_next_hop()` and `ix_cc_ipv6_async_add_prefix()` message helper functions.

C Syntax

```
ix_error ix_cc_ipv6_async_update_prefix (
    ix_uint128 arg_IpAddr,
    ix_uint8 arg_prefixLength,
    ix_cc_rtmv6_nhid arg_NextHopId,
    ix_cc_ipv6_cb_route_op arg_Callback,
    void* arg_pUserContext);
```

Input

<code>arg_IpAddr</code>	Network IPv6 address identifying the route.
<code>arg_prefixLength</code>	Prefix length for the route.
<code>arg_NextHopId</code>	Next hop identifier for route.
<code>arg_Callback</code>	Pointer to a calling application-provided callback function. See ix_cc_ipv6_cb_route_op .
<code>arg_pUserContext</code>	Pointer to a calling application-defined context.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. <code>IX_CC_ERROR_NULL</code>—operation failed, invalid pointer <code>IX_CC_IPV6_ERROR_MSG_LIBRARY</code>—operation failed, failure from Message Support Library
--------------	---

16.3.4 ix_cc_ipv6_async_lookup_prefix()

This primitive helps a client to look up routing information for a given IP address. If there is a successful match, then the information given to the client is `nextHopId` (including the blade id), `portId`, `mtu` and `flags`. If there is no match, then `IX_CC_RTMV6_NHID_NO_ROUTE` will be assigned to `nextHopId` field of `pNextHopData` and all other information is invalid.

C Syntax

```
ix_error ix_cc_ipv6_async_lookup_route (
    ix_uint128 arg_IpAddr,
```

```
ix_cc_ipv6_cb_lookup_route arg_Callback,
void* arg_pUserContext);
```

Input

<code>arg_IpAddr</code>	Network IP address identifying the prefix.
<code>arg_Callback</code>	Pointer to a calling application-provided callback function. See ix_cc_ipv6_cb_lookup_route .
<code>arg_pUserContext</code> <code>t</code>	Pointer to a calling application-defined context.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. <code>IX_ERROR_INVALID_POINTER</code>—operation failed, invalid pointer <code>IX_CC_IPV6_ERROR_MSG_LIBRARY</code>—operation failed, failure from Message Support Library
--------------	---

16.3.4.1 [ix_cc_ipv6_cb_lookup_route](#)

The function prototype for message-handler callback function provided by the calling application.

C Syntax

```
ix_error (*ix_cc_ipv6_cb_lookup_route) (
    ix_error arg_Result,
    void* arg_pContext,
    ix_cc_rtmv6_next_hop_info* arg_pNextHopInfo);
```

Input

<code>arg_Result</code>	Indicates error conditions for the call
<code>arg_pContext</code>	Pointer to user provided context that was sent by API call. Used by the caller to identify the instance of the request.
<code>arg_pNextHopInfo</code> <code>o</code>	Pointer to structure containing the next hop. The result of a successful lookup will be placed in this structure.

16.3.5 ix_cc_ipv6_async_purge_prefixes()

This message helper function removes all prefixes from the RTMv6.

C Syntax

```
ix_error ix_cc_ipv6_async_purge_prefixes();
```

Input

None.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> .
	<ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • <code>IX_CC_IPV6_ERROR_INTERNAL</code>—operation failed, internal failures • <code>IX_CC_IPV6_ERROR_MSG_LIBRARY</code>—operation failed, failure from Message Support Library

16.3.6 ix_cc_ipv6_async_dump_prefixes()

This message helper function dumps all prefixes stored by RTMv6 in memory.

C Syntax

```
ix_error ix_cc_ipv6_async_dump_prefixes(
    ix_cc_ipv6_cb_dump_data arg_Callback,
    void* arg_pUserContext);
```

Input

<code>arg_Callback</code>	Pointer to a calling application-provided callback function. See ix_cc_ipv6_cb_dump_data .
<code>arg_pUserContext</code>	Pointer to a calling application-defined context.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> .
	<ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • <code>IX_ERROR_INVALID_POINTER</code>—operation failed, invalid pointer • <code>IX_CC_IPV6_ERROR_MSG_LIBRARY</code>—operation failed, failure from Message Support Library

16.3.6.1 `ix_cc_ipv6_cb_dump_data`

The function prototype for message-handler callback function provided by the calling application.

C Syntax

```
ix_error (*ix_cc_ipv6_cb_dump_data) (
    ix_error arg_Result,
    void* arg_pContext,
    ix_cc_ipv6_dump_data* arg_pBuffer);
```

Input

<code>arg_Result</code>	Error conditions for the call.
<code>arg_pContext</code>	Pointer to a calling application-provided context that was sent by an API call. Used by the calling application to identify the instance of the request.
<code>arg_pBuffer</code>	Pointer to the structure containing route information as defined in Section 16.1.5, “ix_cc_ipv6_dump_data” on page 246 . The result of a successful operation is placed in this structure

16.3.7 `ix_cc_ipv6_async_add_next_hop()`

This primitive helps a client to add next hop information to NH database. The client needs to add a next hop to the NHDB in order to add routes referring to that next hop.

C Syntax

```
ix_error ix_cc_ipv6_async_add_next_hop (
    ix_cc_rtmv6_nhid arg_NextHopId,
    ix_uint32 arg_NumNextHops,
    ix_cc_rtmv6_next_hop_info* arg_pNextHopData,
    ix_cc_ipv6_cb_route_op arg_Callback,
    void* arg_pUserContext);
```

Input

<code>arg_NextHopId</code>	The next hop ID to be added
<code>arg_NumNextHops</code>	The number of next hops being added for the NextHopId. There can be up to 4 next hops for a given Next Hop ID (this is used for ECMP).

Input

<code>arg_pNextHopData</code>	An array of next hop info structures. Refer to the structure definition in Section 24.1.3, “ix_cc_rtmv6_next_hop_info” on page 514.
<code>arg_Callback</code>	Pointer to a calling application-provided callback function. See ix_cc_ipv6_cb_route_op .
<code>arg_pUserContext</code>	Pointer to a calling application-defined context.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. <code>IX_ERROR_INVALID_POINTER</code>—operation failed, invalid pointer <code>IX_CC_IPV6_ERROR_INVALID_NHID_INFO</code>—operation failed, invalid next hop field <code>IX_CC_IPV6_ERROR_MSG_LIBRARY</code>—operation failed, failure from Message Support Library
--------------	---

16.3.8 ix_cc_ipv6_async_delete_next_hop()

This message helper function deletes a next hop information from the RTMv6 database. The calling application needs to remove all routes associated with the next hop before calling this function. If not, the calling application gets an error.

C Syntax

```
ix_error ix_cc_ipv6_async_delete_next_hop (
    ix_cc_rtmv6_nhid arg_NextHopId,
    ix_cc_ipv6_cb_route_op arg_Callback,
    void* arg_pUserContext);
```

Input

<code>arg_NextHopId</code>	Identifier of next hop to be removed.
<code>arg_Callback</code>	Pointer to a calling application-provided callback function. See ix_cc_ipv6_cb_route_op .
<code>arg_pUserContext</code>	Pointer to a calling application-defined context.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. <code>IX_ERROR_INVALID_POINTER</code>—operation failed, invalid pointer <code>IX_CC_IPV6_ERROR_MSG_LIBRARY</code>—operation failed, failure from Message Support Library
--------------	---

16.3.9 `ix_cc_ipv6_async_update_next_hop()`

This message helper function updates a next hop information into the RTMv6 database. The calling application needs to add next hop to the RTMv6 in order to add routes referring to that next hop.

C Syntax

```
ix_error ix_cc_ipv6_async_update_next_hop (
    ix_cc_rtmv6_nhid arg_NextHopId,
    ix_uint32 arg_NumNextHops,
    ix_cc_rtmv6_next_hop_info* arg_pNextHopData,
    ix_cc_ipv6_cb_route_op arg_Callback,
    void* arg_pUserContext);
```

Input

<code>arg_NextHopId</code>	Identifier of next hop to be updated
<code>arg_NumNextHops</code>	The number of next hops in the update. Existing next hops will be deleted and replaced by those specified in the <code>arg_pNextHopData</code> argument.
<code>arg_pNextHopData</code> <code>a</code>	An array of next hop info structures. There can be up to 4 elements in this array. Refer to section 4.13.3.1.3 for the structure definition.
<code>arg_Callback</code>	Pointer to a calling application-provided callback function. ix_cc_ipv6_cb_route_op .
<code>arg_pUserContext</code> <code>t</code>	Pointer to a calling application-defined context.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. <code>IX_CC_ERROR_NULL</code>—operation failed, invalid pointer <code>IX_CC_IPV6_ERROR_MSG_LIBRARY</code>—operation failed, failure from Message Support Library
--------------	---

16.3.10 ix_cc_ipv6_async_get_next_hop()

This message helper function retrieves a next hop information from the RTMv6 database.

C Syntax

```
ix_error ix_cc_ipv6_async_get_next_hop (
    ix_cc_rtmv6_nhid arg_NextHopId,
    ix_cc_ipv6_cb_get_next_hop arg_Callback,
    void* arg_pUserContext);
```

Input

arg_NextHopId	Identifier of the next hop information to be retrieved from RTMv6.
arg_Callback	Pointer to a calling application-provided callback function. See ix_cc_ipv6_cb_get_next_hop .
arg_pUserContext	Pointer to a calling application-defined context.

Output/Returns

Return Value	Returns a valid ix_error. <ul style="list-style-type: none"> IX_SUCCESS—the operation succeeded. IX_ERROR_INVALID_POINTER—operation failed, invalid pointer IX_CC_IPV6_ERROR_MSG_LIBRARY—operation failed, failure from Message Support Library
--------------	--

16.3.10.1 ix_cc_ipv6_cb_get_next_hop

The function prototype for message-handler callback function provided by the calling application.

C Syntax

```
ix_error (*ix_cc_ipv6_cb_get_next_hop) (
    ix_error arg_Result,
    void* arg_pContext,
    ix_cc_rtmv6_next_hop_info* arg_pNextHopInfo);
```

Input

arg_Result	Error conditions for the call
arg_pContext	Pointer to a calling application-provided context that was sent by an API call. Used by the calling application to identify the instance of the request.
arg_pNextHopInfo	Pointer to the structure containing the next hop. The result of a successful lookup is placed in this structure.

16.3.11 ix_cc_ipv6_async_dump_next_hops()

This message helper function dumps all next hops of the RTMv6 database in memory.

C Syntax

```
ix_error ix_cc_ipv6_async_dump_next_hops (
    ix_cc_ipv6_cb_dump_data arg_Callback,
    void* arg_pUserContext);
```

Input

arg_Callback	Pointer to a calling application-provided callback function. See ix_cc_ipv6_cb_dump_data .
arg_pUserContext	Pointer to a calling application-defined context.

Output/Returns

Return Value	Returns a valid ix_error. <ul style="list-style-type: none"> IX_SUCCESS—the operation succeeded. IX_ERROR_INVALID_POINTER—operation failed, invalid pointer IX_CC_IPV6_ERROR_MSG_LIBRARY—operation failed, failure from Message Support Library
--------------	--

16.3.12 ix_cc_ipv6_async_purge_rtm()

This message helper function removes all the prefixes and next hops from the RTMv6 database.

C Syntax

```
ix_error ix_cc_ipv6_async_purge_rtm ();
```

Input

None.

Output/Returns

Return Value	Returns a valid ix_error. <ul style="list-style-type: none"> IX_SUCCESS—the operation succeeded. IX_CC_IPV6_ERROR_INTERNAL—operation failed, internal failures IX_CC_IPV6_ERROR_MSG_LIBRARY—operation failed, failure from Message Support Library
--------------	---

16.3.13 ix_cc_ipv6_async_set_mtu()

This message helper function updates MTU for a given next hop. Before calling this message helper, the calling application must call `ix_cc_ipv6_async_add_next_hop()` to add the next hop identifier in the RTMv6 database. The calling application validates the MTU.

C Syntax

```
ix_error ix_cc_ipv6_async_set_mtu(
    ix_cc_rtmv6_nhid arg_NextHopId,
    ix_uint32 arg_Mtu,
    ix_cc_ipv6_cb_route_op arg_Callback,
    void* arg_pUserContext);
```

Input

<code>arg_NextHopId</code>	Identifier of next hop.
<code>arg_Mtu</code>	New maximum transmission unit for the given next hop.
<code>Arg_Callback</code>	Calling application-defined callback function.
<code>arg_pUserContext</code>	Pointer to a calling application-defined context.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. <code>IX_ERROR_INVALID_POINTER</code>—operation failed, invalid pointer <code>IX_CC_IPV6_ERROR_MSG_LIBRARY</code>—operation failed, failure from Message Support Library
--------------	---

16.3.14 ix_cc_ipv6_async_set_flags()

This message helper function updates the flags for a given next hop. Before calling this message helper, the calling application needs to call `ix_cc_ipv6_async_add_next_hop()` to add a next hop identifier in the RTMv6 database. The supported flags are defined by the IPv6 Forwarder microblock in *Intel® Internet Exchange Architecture Software Building Blocks Developer's Manual*.

C Syntax

```
ix_error ix_cc_ipv6_async_set_flags(
    ix_cc_rtmv6_nhid arg_NextHopId,
    ix_uint32 arg_Flags,
    ix_cc_ipv6_cb_route_op arg_Callback,
    void* arg_pUserContext);
```

Input

<code>arg_NextHopId</code>	Identifier of next hop.
<code>arg_Flags</code>	New flags for the given next hop
<code>Arg_Callback</code>	Calling application-defined callback function.
<code>arg_pUserContext</code> <code>t</code>	Pointer to a calling application-defined context

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • <code>IX_ERROR_INVALID_POINTER</code>—operation failed, invalid pointer • <code>IX_CC_IPV6_ERROR_INVALID_FLAGS</code>—operation failed, unsupported flags • <code>IX_CC_IPV6_ERROR_MSG_LIBRARY</code>—operation failed, failure from Message Support Library
--------------	--

16.3.15 `ix_cc_ipv6_async_get_rate_limit_time()`

This message helper function retrieves the time interval for rate limiting in milliseconds.

C Syntax

```
ix_error ix_cc_ipv6_async_get_rate_limit_time (
    ix_cc_ipv6_cb_get_rate_limit_time arg_Callback,
    void* arg_pUserContext);
```

Input

<code>arg_Callback</code>	Pointer to a calling application-provided callback function. See ix_cc_ipv6_cb_get_rate_limit_time .
<code>arg_pUserContext</code> <code>t</code>	Pointer to a calling application-defined context.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • <code>IX_ERROR_INVALID_POINTER</code>—operation failed, invalid pointer • <code>IX_CC_IPV6_ERROR_MSG_LIBRARY</code>—operation failed, failure from Message Support Library
--------------	---

16.3.15.1 ix_cc_ipv6_cb_get_rate_limit_time

The function prototype for message-handler callback function provided by the calling application.

C Syntax

```
ix_error (*ix_cc_ipv6_cb_get_rate_limit_time) (
    ix_error arg_Result,
    void* arg_pContext,
    ix_uint16* arg_pInterval);
```

Input

arg_Result	Indicates error conditions for the call
arg_pContext	Pointer to user provided context that was sent by API call. Used by the caller to identify the instance of the request.
arg_pInterval	Pointer to the interval currently set.

16.3.16 ix_cc_ipv6_async_set_rate_limit_time()

This message helper function sets the rate limit interval in milliseconds. This interval indicates how often an ICMP error packet is dequeued and sent.

C Syntax

```
ix_error ix_cc_ipv6_async_set_rate_limit_time (
    ix_uint16 arg_Interval);
```

Input

arg_Interval	The interval in milliseconds. Zero is not a valid value.
--------------	--

Output/Returns

Return Value	<p>Returns a valid ix_error.</p> <ul style="list-style-type: none"> IX_SUCCESS—the operation succeeded. IX_CC_IPV6_ERROR_INVALID_SLEEP_TIME—operation failed, unsupported sleep time IX_CC_IPV6_ERROR_MSG_LIBRARY—operation failed, failure from Message Support Library
--------------	---

16.3.17 ix_cc_ipv6_async_get_queue_depth()

This message helper function retrieves the depth of the ICMP error message queue. The calling application retrieves the maximum number of ICMP messages that can fit in the queue, not the number of messages currently in the queue. This value is set by the system during initialization.

C Syntax

```
ix_error ix_cc_ipv6_async_get_queue_depth (
    ix_cc_ipv6_cb_get_queue_depth arg_Callback,
    void* arg_pUserContext);
```

Input

arg_Callback	Pointer to a calling application-provided callback function. See ix_cc_ipv6_cb_get_queue_depth .
arg_pUserContext	Pointer to a calling application-defined context.

Output/Returns

Return Value	Returns a valid ix_error. <ul style="list-style-type: none"> IX_SUCCESS—the operation succeeded. IX_ERROR_INVALID_POINTER—operation failed, invalid pointer IX_CC_IPV6_ERROR_MSG_LIBRARY—operation failed, failure from Message Support Library
--------------	--

16.3.17.1 ix_cc_ipv6_cb_get_queue_depth

The function prototype for message-handler callback function provided by the calling application.

C Syntax

```
ix_error (*ix_cc_ipv6_cb_get_queue_depth) (
    ix_error arg_Result,
    void* arg_pContext,
    ix_uint16* arg_pQueueDepth);
```

Input

arg_Result	Error conditions for the call
arg_pContext	Pointer to the calling application-provided context that was sent by an API call. Used by the calling application to identify the instance of the request.
arg_pQueueDepth	Pointer to the current number of entries that the ICMP error message queue can hold.

16.3.18 ix_cc_ipv6_async_get_statistics()

This message helper function retrieves the statistics report from the IPv6 Forwarder core component. This function retrieves the statistics from IPv6 microblock, RTMv6 and IPV6 Forwarder core component.

C Syntax

```
ix_error ix_cc_ipv6_async_get_statistics (
    ix_cc_ipv6_cb_get_statistics arg_Callback,
    void *arg_pUserContext);
```

Input

arg_Callback	Pointer to a calling application-provided callback. See ix_cc_ipv6_cb_get_statistics .
arg_pUserContext	Pointer to a calling application-defined context.

Output/Returns

Return Value	Returns a valid ix_error. <ul style="list-style-type: none"> IX_SUCCESS—the operation succeeded. IX_ERROR_INVALID_POINTER—operation failed, invalid pointer IX_CC_IPV6_ERROR_MSG_LIBRARY—operation failed, failure from Message Support Library
--------------	--

16.3.18.1 ix_cc_ipv6_cb_get_statistics

The function prototype for message-handler callback function provided by the calling application.

C Syntax

```
ix_error (*ix_cc_ipv6_cb_get_statistics) (
    ix_error arg_Result,
    void* arg_pContext,
    ix_cc_ipv6_stats_data* arg_pBuffer);
```

Input

arg_Result	Indicates error conditions for the call
arg_pContext	Pointer to user provided context that was sent by API call. Used by the caller to identify the instance of the request.
arg_pBuffer	Pointer to memory to hold statistics data.

16.3.19 ix_cc_ipv6_async_perform_addr_resolution()

This message helper function requests the address resolution to be performed for the destination IP address contained in the packet. The buffer's meta-data should contain the index into the L2 table (the L2 ID) at which the L2 address should be stored once the Address Resolution is complete. It is expected that the ONLY the calling application is the Ethernet TX core component. For more information about Address Resolution, refer to [Section 53.4.5, “Neighbor Discovery”](#) in *Intel® Internet Exchange Architecture Software Building Blocks Developer’s Manual*.

Note: The callback is not invoked in this implementation.

C Syntax

```
ix_error ix_cc_ipv6_async_perform_addr_resolution(
    ix_buffer_handle arg_hBuffer,
    ix_cc_ipv6_cb_route_op arg_Callback,
    void* arg_pUserContext);
```

Input

arg_hBuffer	Handle of the hardware buffer containing the packet. Since the only client of this primitive is the Ethernet TX core component, this buffer is assumed to contain an entire Ethernet frame and is processed accordingly.
arg_Callback	Pointer to a calling application-provided callback function. See ix_cc_ipv6_cb_route_op (reserved).
arg_pUserContext	Pointer to a calling application-defined context (reserved).

Output/Returns

Return Value	Returns a valid ix_error. <ul style="list-style-type: none"> IX_SUCCESS—the operation succeeded. IX_CC_ERROR_OOM—operation failed, The system is out of memory. IX_CC_IPV6_ERROR_MSG_LIBRARY—operation failed, failure from Message Support Library IX_CC_IPV6_ERROR_INVALID_PARAMETERS—operation failed, Some of the input parameters were invalid.
--------------	--

16.3.19.1 ix_cc_ipv6_cb_route_op

The function prototype for message-handler callback function provided by the calling application.

C Syntax

```
ix_error (*ix_cc_ipv6_cb_route_op) (
    ix_error arg_Result,
    void* arg_pContext);
```

Input

<code>arg_Result</code>	Error conditions for the call
<code>arg_pContext</code>	Pointer to a calling application-provided context used to identify the instance of the request.

Note: This is a generic callback type and is used for IPv6 message helpers that do not require data to be returned in the callback—only the result and context are relevant. Also, a success indication by the callback may not necessarily mean that the address resolution succeeded. It means that the Neighbor Discovery module either successfully updated the L2 table with the stored information, or it successfully initiated the Address Resolution.

16.3.20 `ix_cc_ipv6_async_add_neighbor()`

This message helper function enables a calling application to update the neighbor cache and add an entry into the L2 table. For more information about Address Resolution, refer to [Section 53.4.5, “Neighbor Discovery”](#) in *Intel® Internet Exchange Architecture Software Building Blocks Developer’s Manual*.

C Syntax

```
ix_error ix_cc_ipv6_async_add_neighbor (
    ix_uint16 arg_L2Index,
    ix_uint128 arg_NbrIp,
    ix_uint16 arg_OutputPort,
    ix_uint8 *arg_L2Addr,
    ix_cc_ipv6_cb_route_op arg_Callback,
    void* arg_pUserContext);
```

Input

<code>arg_L2Index</code>	The index into the L2 table at which this information should be written.
<code>arg_NbrIp</code>	The IP address of the neighbor.
<code>arg_OutPutPort</code>	The egress port through which all packets destined for the neighbor should be transmitted.
<code>arg_L2Addr</code>	The link layer address of the neighbor and a pointer to an array of size 6 octets containing the MAC address.
<code>arg_Callback</code>	Pointer to a calling application-provided callback function. See ix_cc_ipv6_cb_route_op .
<code>arg_pUserContext</code>	Pointer to a calling application-defined context.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. <code>IX_CC_ERROR_OOM</code>—operation failed, the system is out of memory. <code>IX_CC_IPV6_INVALID_PARAMETERS</code>—operation failed, One or more input parameters are invalid. <code>IX_CC_IPV6_ERROR_MSG_LIBRARY</code>—operation failed, failure from Message Support Library
--------------	---

16.3.21 `ix_cc_ipv6_async_del_neighbor()`

This message helper function enables the calling application to delete the contents located at the supplied L2Index in the L2 table. Note that this may not result in the deletion of the corresponding Neighbor Cache entry maintained by the Ipv6 Forwarder core component. For more information about Address Resolution, refer to [Section 53.4.5, “Neighbor Discovery”](#) in *Intel® Internet Exchange Architecture Software Building Blocks Developer’s Manual*.

C Syntax

```
ix_error ix_cc_ipv6_async_del_neighbor (
    ix_uint16 arg_L2Index,
    ix_cc_ipv6_cb_route_op arg_Callback,
    void* arg_pUserContext);
```

Input

<code>arg_L2Index</code>	The index into the L2 table at which information should be deleted.
<code>arg_Callback</code>	Pointer to a calling application-provided callback function. See ix_cc_ipv6_cb_route_op .
<code>arg_pUserContext</code>	Pointer to a calling application-defined context.

I

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. <code>IX_CC_IPV6_ERROR_INVALID_PARAMETERS</code>—operation failed, One or more input parameters were invalid. <code>IX_CC_IPV6_ERROR_MSG_LIBRARY</code>—operation failed, failure from Message Support Library
--------------	---

16.3.22 ix_cc_ipv6_sync_add_neighbor()

This message helper function updates the neighbor cache and adds an entry into the L2 table in a synchronous manner. The calling application may set the `arg_L2Addr` to `NULL`, in which case Address resolution is performed to resolve the L2 address. For more information about Address Resolution, refer to [Section 53.4.5, “Neighbor Discovery”](#) in *Intel® Internet Exchange Architecture Software Building Blocks Developer's Manual*.

Note: This function must not be invoked by the core components.

C Syntax

```
ix_error ix_cc_ipv6_sync_add_neighbor(
    ix_uint16 arg_L2Index,
    ix_uint128 arg_NbrIp,
    ix_uint16 arg_OutputPort,
    ix_uint8 *arg_L2Addr);
```

Input

<code>arg_L2Index</code>	The index into the L2 table at which this information should be written.
<code>arg_NbrIp</code>	The IP address of the neighbor.
<code>arg_OutPutPort</code>	The egress port through which all packets destined for the neighbor should be transmitted.
<code>arg_L2Addr</code>	The link layer address of the neighbor. This is a pointer to an array of size 6 octets containing the MAC address. If set to <code>NULL</code> , Address Resolution will first be performed to resolve the L2 address.

Output/Returns

Return Value	<p>Returns a valid <code>ix_error</code>.</p> <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. <code>IX_CC_ERROR_OOM</code>—operation failed, the system is out of memory. <code>IX_CC_IPV6_INVALID_PARAMETERS</code>—operation failed, one or more input parameters are invalid. <code>IX_CC_IPV6_ERROR_MSG_LIBRARY</code>—operation failed, failure from Message Support Library
--------------	---

16.3.23 ix_cc_ipv6_sync_del_neighbor()

This message helper function deletes the contents of L2Index in the L2 table in a synchronous manner. However, this may not result in the deletion of the corresponding Neighbor Cache entry maintained by the Ipv6 Forwarder core component. For more information about Address Resolution, refer to [Section 53.4.5, “Neighbor Discovery”](#) in *Intel® Internet Exchange Architecture Software Building Blocks Developer’s Manual*.

Note: This function must NOT be invoked by the core components.

C Syntax

```
ix_error ix_cc_ipv6_sync_del_neighbor(
    ix_uint16 arg_L2Index)
```

Input

arg_L2Index	The index into the L2 table at which information should be deleted.
-------------	---

Output/Returns

Return Value	<p>Returns a valid ix_error.</p> <ul style="list-style-type: none"> • IX_SUCCESS—the operation succeeded. • IX_CC_IPV6_INVALID_PARAMETERS—operation failed, one or more input parameters are invalid. • IX_CC_IPV6_ERROR_MSG_LIBRARY—operation failed, failure from Message Support Library
--------------	--

16.3.24 ix_cc_ipv6_async_send_icmp_error()

This message helper function enables the calling application to send an ICMP error message. The hardware buffer is expected to be the IPv6 packet causing the error to be sent. The calling application must ensure that the meta-data is correctly configured and the original IPv6 header is present. The ICMP error message is sent to the source address contained in the IPv6 header. All ICMP error messages are rate limited.

C Syntax

```
ix_error ix_cc_ipv6_async_send_icmp_error (
    ix_cc_ipv6_icmp_err_type arg_IcmpErrType,
    ix_cc_ipv6_icmp_err_code arg_IcmpErrCode,
    ix_buffer_handle arg_hBuffer,
    ix_uint32 arg_Param,
    ix_cc_ipv6_cb_route_op arg_Callback,
    void* arg_pUserContext);
```

Input

<code>arg_IcmpErrType</code>	The “type” field to be included in the ICMP header. Defined in Section 16.1.9, “ix_cc_ipv6_icmp_err_type” on page 248.
<code>arg_IcmpErrCode</code>	The “code” field to be included in the ICMP header. Defined in Section 16.1.10, “ix_cc_ipv6_icmp_err_code” on page 248.
<code>arg_hBuffer</code>	The hardware buffer handle containing the IPv6 packet causing the error.
<code>arg_Param</code>	An optional parameter that is needed by some messages and ignored for other messages.
<code>arg_Callback</code>	Pointer to a calling application-provided callback function. ix_cc_ipv6_cb_route_op.
<code>arg_pUserContext</code>	Pointer to a calling application-defined context.

Output/Returns

Return Value	<p>Returns a valid <code>ix_error</code>.</p> <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • <code>IX_CC_IPV6_ERROR_INVALID_PARAMETERS</code>—operation failed, One or more of the input parameters are invalid. • <code>IX_CC_IPV6_ERROR_MSG_LIBRARY</code>—operation failed, failure from Message Support Library
--------------	---

16.3.25 ix_cc_ipv6_async_send_icmp_info_message()

This message helper functions sends ICMP informational messages. The calling application must ensure that the type and code provided are valid. The IPv6 Forwarder core component only does basic validity tests on the type and code fields, in the interest of forward compatibility. The ICMP informational messages are NOT rate limited.

C Syntax

```
ix_error ix_cc_ipv6_async_send_icmp_info_message (
    ix_uint128 arg_SrcIpAddr,
    ix_uint128 arg_DstIpAddr,
    ix_uint8 arg_Type,
    ix_uint8 arg_Code,
    ix_uint32 arg_PayloadLength,
    ix_uint8*arg_pPayload,
    ix_cc_ipv6_cb_route_op arg_Callback,
    void* arg_pUserContext);
```

Input

arg_SrcIpAddr	The source IP address to be used in the IPv6 header.
arg_DstIpAddr	The destination IP address to be used in the IPv6 header.
arg_Type	The ICMP message type to be inserted in the ICMP header.
arg_Code	The ICMP message code to be inserted in the ICMP header.
arg_PayloadLength	The length of the payload to be included in the ICMP packet.
arg_pPayload	Pointer to a buffer containing the payload to be transmitted.
arg_Callback	Pointer to a calling application-provided callback function.
arg_pUserContext	Pointer to a calling application-defined context.

Output/Returns

Return Value	<p>Returns a valid <code>ix_error</code>.</p> <ul style="list-style-type: none"> IX_SUCCESS—the operation succeeded. IX_CC_ERROR_RANGE—operation failed, the Type field does not fall into the appropriate range for ICMP informational messages. IX_CC_IPV6_ERROR_INVALID_PARAMETERS—operation failed, one or more input parameters were invalid. IX_CC_IPV6_ERROR_MSG_LIBRARY—operation failed, failure from Message Support Library
--------------	--

16.4 Library API

Table 16-5 lists the IPV6 Forwarder library API.

Table 16-5. IPV6 Forwarder Library API

API	Description
<code>ix_cc_ipv6_add_prefix()</code>	Adds a prefix in RTMv6
<code>ix_cc_ipv6_delete_prefix()</code>	Deletes a prefix (route) from RTMv6
<code>ix_cc_ipv6_update_prefix()</code>	Updates a prefix in RTMv6
<code>ix_cc_ipv6_lookup_prefix()</code>	Looks up routing information for a given IP address
<code>ix_cc_ipv6_purge_prefixes()</code>	Removes all prefixes from the RTMv6
<code>ix_cc_ipv6_dump_prefixes()</code>	Dumps all prefixes stored by RTMv6 in memory
<code>ix_cc_ipv6_add_next_hop()</code>	Adds next hop information to NH database
<code>ix_cc_ipv6_delete_next_hop()</code>	Deletes next hop information from RTMv6
<code>ix_cc_ipv6_update_next_hop()</code>	Updates next hop information into RTMv6 database
<code>ix_cc_ipv6_get_next_hop()</code>	Retrieves next hop information from RTMv6
<code>ix_cc_ipv6_dump_next_hops()</code>	Dumps all next hops of RTM in memory
<code>ix_cc_ipv6_purge_rtm()</code>	Removes all routes and next hops from RTM database
<code>ix_cc_ipv6_set_mtu()</code>	Updates MTU for a given next hop
<code>ix_cc_ipv6_set_flags()</code>	Updates flags for a given next hop
<code>ix_cc_ipv6_get_rate_limit_time()</code>	Retrieves the time interval for rate limiting in milliseconds
<code>ix_cc_ipv6_set_rate_limit_time()</code>	Sets the rate limit interval in milliseconds
<code>ix_cc_ipv6_get_queue_depth()</code>	Retrieves the depth of the ICMP error message queue
<code>ix_cc_ipv6_get_statistics()</code>	Retrieves statistics report from IPV6 Forwarder core component
<code>ix_cc_ipv6_set_property()</code>	Sets dynamic properties of IPV6 Forwarder core component
<code>ix_cc_ipv6_perform_addr_resolution()</code>	Requests address resolution to be performed for the destination IP address contained in the packet
<code>ix_cc_ipv6_add_neighbor()</code>	Updates the neighbor cache and add an entry into the L2 table
<code>ix_cc_ipv6_del_neighbor()</code>	Deletes the contents located at the supplied L2Index in the L2 table
<code>ix_cc_ipv6_send_icmp_error()</code>	Sends an ICMP error message
<code>ix_cc_ipv6_send_icmp_info_message()</code>	Sends ICMP informational messages

16.4.1 ix_cc_ipv6_add_prefix()

This library function adds a prefix in the RTMv6 table and internally calls `ix_cc_rtmv6_add_route()` API of RTMv6 core component to add route in RTMv6 database.

C Syntax

```
ix_error ix_cc_ipv6_add_prefix (
    ix_uint128 arg_IpAddr,
    ix_uint8 arg_PrefixLen,
    ix_cc_rtmv6_nhid arg_NextHopId,
    void* arg_pUserContext);
```

Input

<code>arg_IpAddr</code>	Network IP address identifying the prefix.
<code>arg_PrefixLen</code>	A prefix length.
<code>arg_NextHopId</code>	A next hop identifier for a prefix.
<code>arg_pUserContext</code>	A pointer to a calling application-defined context.

Output/Returns

Return Value	<p>Returns a valid <code>ix_error</code>.</p> <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • <code>IX_CC_IPV6_ERROR_INVALID_NHID</code>—operation failed, invalid next hop identifier • <code>IX_CC_IPV6_ERROR_DUPLICATE_ROUTE</code>—operation failed, route was already added to this NHID • <code>IX_CC_IPV6_ERROR_CONFLICTING_ROUTE</code>—operation failed, route was already added to another NHID • <code>IX_CC_IPV6_ERROR_RTM</code>—operation failed, RTMv6 internal failure
--------------	---

16.4.2 ix_cc_ipv6_delete_prefix()

This library function deletes a prefix (route) from RTMv6 and internally calls `ix_cc_rtmv6_delete_route()` API of RTMv6 core component to delete the prefix (route) from the RTMv6 database.

C Syntax

```
ix_error ix_cc_ipv6_delete_prefix (
    ix_uint128 arg_IpAddr,
    ix_uint8 arg_PrefixLen,
    void* arg_pUserContext);
```

Input

<code>arg_IpAddr</code>	Network IP address identifying the prefix.
<code>arg_PrefixLen</code>	Prefix length.
<code>arg_pUserContext</code> <code>t</code>	A pointer to a calling application-defined context.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. <code>IX_CC_IPV6_ERROR_INVALID_ROUTE</code>—operation failed, the network address and prefix length pair do not exist in the RTMv6 <code>IX_CC_IPV6_ERROR_RTM</code>—operation failed, RTMv6 internal failure
--------------	--

16.4.3 ix_cc_ipv6_update_prefix()

This library function updates a prefix in RTMv6 and internally calls `ix_cc_rtmv6_update_route()` API of RTMv6 core component to update the prefix in the RTMv6 database.

C Syntax

```
ix_error ix_cc_ipv6_update_prefix (
    ix_uint128 arg_IpAddr,
    ix_uint8 arg_PrefixLen,
    ix_cc_rtmv4_nhid arg_NextHopId,
    void* arg_pUserContext);
```

Input

<code>arg_IpAddr</code>	Network IP address identifying the prefix.
-------------------------	--

Input (Continued)

<code>arg_PrefixLen</code>	Prefix length.
<code>arg_NextHopId</code>	Next hop identifier for prefix.
<code>arg_pUserContext</code>	A pointer to a calling application-defined context.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. <code>IX_CC_IPV6_ERROR_INVALID_NHID</code>—operation failed, invalid next hop identifier <code>IX_CC_IPV6_ERROR_CONFLICTING_ROUTE</code>—operation failed, route was already added to another NHID <code>IX_CC_IPV6_ERROR_RTM</code>—operation failed, RTMv6 internal failure
--------------	---

16.4.4 `ix_cc_ipv6_lookup_prefix()`

This function looks up routing information for a given IP address. If there is no match, then `IX_CC_RTMV6_NHID_NO_ROUTE` is assigned to `nextHopId` field of `pNextHopInfo` and all other information is invalid.

C Syntax

```
ix_error ix_cc_ipv6_lookup_prefix (
    ix_uint128 arg_IpAddr,
    ix_cc_rtmv6_next_hop_info* arg_pNextHopInfo,
    void* arg_pUserContext);
```

Input

<code>arg_IpAddr</code>	Network IP address identifying the route
<code>arg_pUserContext</code>	A pointer to a calling application-defined context.

Output/Returns

<code>arg_pNextHopInfo</code>	The structure defined by RTMv6 core component. Result of lookup will be placed into the structure.
Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. <code>IX_ERROR_INVALID_POINTER</code>—operation failed, invalid pointer <code>IX_CC_IPV6_ERROR_RTM</code>—operation failed, RTMv6 internal failure

16.4.5 ix_cc_ipv6_purge_prefixes()

This function removes all prefixes from the RTMv6 database.

C Syntax

```
ix_error ix_cc_ipv6_purge_prefixes(void* arg_pUserContext);
```

Input

`arg_pUserContext` A pointer to a calling application-defined context.

Output/Returns

Return Value Returns a valid `ix_error`.

- `IX_SUCCESS`—the operation succeeded.
- `IX_CC_IPV6_ERROR_RTM`—operation failed, RTMv6 internal failure

16.4.6 ix_cc_ipv6_dump_prefixes()

This primitive helps to dump all prefixes stored by RTMv6 in memory. The calling application uses `IX_CC_RTMV6_DUMP_ROUTE_SIZE` to determine how many bytes to allocate.

C Syntax

```
ix_error ix_cc_ipv6_dump_prefixes (
    char* arg_pBuffer,
    ix_uint32 arg_Size,
    void* arg_pUserContext);
```

Input

`arg_pBuffer` Pointer to a block of memory where routes are to be stored.

`arg_Size` Size of buffer in bytes.

`arg_pUserContext` A pointer to a calling application-defined context.

Output/Returns

Return Value Returns a valid `ix_error`.

- `IX_SUCCESS`—the operation succeeded.
- `IX_CC_IPV6_ERROR_RTM`—operation failed, RTMv6 internal failure
- `IX_ERROR_INVALID_POINTER`—operation failed, invalid pointer

16.4.7 `ix_cc_ipv6_add_next_hop()`

This function adds a next hop information to the NH database. The calling application needs to add a next hop to the NHDB in order to add routes referring to that next hop.

C Syntax

```
ix_error ix_cc_ipv6_add_next_hop(
    ix_cc_rtmv6_nhid arg_NextHopId,
    ix_uint32 arg_NumNextHops,
    ix_cc_rtmv6_next_hop_info* arg_pNextHopInfo,
    void* arg_pUserContext);
```

Input

<code>arg_NextHopId</code>	The next hop ID to be added
<code>arg_NumNextHops</code>	The number of next hops being added for the NextHopId. There can be up to 4 next hops for a given Next Hop ID (this is used for ECMP).
<code>arg_pNextHopInfo</code>	Array of next hops. Defined in Section 24.1.3 , “ <code>ix_cc_rtmv6_next_hop_info</code> ” on page 514.
<code>arg_pUserContext</code>	A pointer to a calling application-defined context.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. <code>IX_CC_IPV6_ERROR_DUPLICATE_NEXT_HOP</code>—operation failed, the NHID is already in use <code>IX_CC_IPV6_ERROR_INVALID_NHID_INFO</code>—operation failed, invalid next hop field. <code>IX_CC_IPV6_ERROR_RTM</code>—operation failed, RTMv6 internal error <code>IX_ERROR_INVALID_POINTER</code>—operation failed, invalid pointer
--------------	---

16.4.8 `ix_cc_ipv6_delete_next_hop()`

This library function deletes the next hop information from RTMv6 and internally calls `ix_cc_rtmv6_delete_next_hop()` API of RTMv6 core component to delete the next hop information from the RTMv6 database.

C Syntax

```
ix_error ix_cc_ipv6_delete_next_hop (
    ix_cc_rtmv6_nhid arg_pNextHopInfo,
    void* arg_pUserContext);
```

Input

<code>arg_NextHopId</code>	Identifier of next hop to be removed.
<code>arg_pUserContext</code> <code>t</code>	A pointer to a calling application-defined context.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. <code>IX_CC_IPV6_ERROR_INVALID_NEXT_HOP_ID</code>—operation failed, NHID could not be found in RTMv6 <code>IX_CC_IPV6_ERROR_NEXT_HOP_ID_IN_USE</code>—operation failed, NHID is in use <code>IX_CC_IPV6_ERROR_RTM</code>—operation failed, RTMv6 internal failure
--------------	---

16.4.9 `ix_cc_ipv6_update_next_hop()`

This function updates the next hop information into the RTMv6 database. The calling application needs to add a next hop to the RTMv6 database in order to add routes referring to that next hop.

C Syntax

```
ix_error ix_cc_ipv6_update_next_hop (
    ix_cc_rtmv6_nhid arg_NextHopId,
    ix_uint32 arg_NumNextHops,
    ix_cc_rtmv6_next_hop_info* arg_pNextHopInfo,
    void* arg_pUserContext);
```

Input

<code>arg_NextHopId</code>	The next hop ID to be updated
<code>arg_NumNextHops</code>	The number of next hops added for the NextHopId. There can be up to 4 next hops for a given Next Hop ID (this is used for ECMP).
<code>arg_pNextHopInfo</code> <code>o</code>	Array of next hops. Defined in Section 24.1.3 , “ <code>ix_cc_rtmv6_next_hop_info</code> ” on page 514.
<code>arg_pUserContext</code> <code>t</code>	A pointer to a calling application-defined context.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • <code>IX_CC_IPV6_ERROR_INVALID_NHID_INFO</code>—operation failed, invalid next hop field. • <code>IX_CC_IPV6_ERROR_RTM</code>—operation failed, RTMv6 internal error • <code>IX_ERROR_INVALID_POINTER</code>—operation failed, invalid pointer
--------------	--

16.4.10 `ix_cc_ipv6_get_next_hop()`

This library function retrieves the next hop information from the RTMv6 database and internally calls `ix_cc_rtmv6_get_next_hop()` API of the RTMv6 core component to retrieve next hop information.

C Syntax

```
ix_error ix_cc_ipv6_get_next_hop (
    ix_cc_rtmv6_nhid arg_NextHopId,
    ix_cc_rtmv6_next_hop_info* arg_pNextHopInfo);
```

Input

<code>arg_NextHopId</code>	Identifier of the next hop information to be retrieved from RTMv6.
<code>arg_pUserContext</code>	A pointer to a calling application-defined context.

Output/Returns

<code>arg_pNextHopInfo</code>	Result will be placed into this structure.
Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • <code>IX_CC_IPV6_ERROR_INVALID_NEXT_HOP_ID</code>—operation failed, NHID could not able to find in RTMv6 • <code>IX_CC_IPV6_ERROR_RTM</code>—operation failed, RTMv6 internal failure

16.4.11 ix_cc_ipv6_dump_next_hops()

This library function dumps all next hops of RTM in memory and internally calls `ix_cc_rtmv6_dump_next_hops()` API of the RTMv6 core component. the calling application uses `IX_CC_RTMV6_DUMP_NEXT_HOP_SIZE` to determine how many bytes to allocate.

C Syntax

```
ix_error ix_cc_ipv6_dump_next_hops(
    char* arg_pBuffer,
    ix_uint32 arg_Size,
    void* arg_pUserContext);
```

Input

<code>arg_pBuffer</code>	Pointer to a block of memory where next hops are to be stored.
<code>arg_Size</code>	Size of buffer in bytes.
<code>arg_pUserContext</code>	A pointer to a calling application-defined context.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> .
	<ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. <code>IX_CC_IPV6_ERROR_RTM</code>—operation failed, RTMv6 internal failure <code>IX_ERROR_INVALID_POINTER</code>—operation failed, invalid pointer

16.4.12 ix_cc_ipv6_purge_rtm()

This function removes all routes and next hops from the RTMv6 database and internally calls `ix_cc_rtmv6_purge()` API of the RTMv6 core component.

C Syntax

```
ix_error ix_cc_ipv6_purge_rtm(void* arg_pUserContext);
```

Input

<code>arg_pUserContext</code>	A pointer to a calling application-defined context.
-------------------------------	---

Output/Returns

Return Value	Returns a valid <code>ix_error</code> .
	<ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. <code>IX_CC_IPV6_ERROR_RTM</code>—operation failed, RTMv6 internal failure

16.4.13 ix_cc_ipv6_set_mtu()

This function updates MTU for a given next hop and internally calls `ix_cc_rtmv6_set_mtu()` API of the RTMv6 core component. Before calling this function, the calling application client needs to call `ix_cc_ipv6_async_add_next_hop()` to add next hop identifier in RTMv6 database. The calling application validates the MTU.

C Syntax

```
ix_error ix_cc_ipv6_set_mtu(
    ix_cc_rtmv4_nhid arg_NextHopId,
    ix_uint32 arg_Mtu,
    void* arg_pUserContext);
```

Input

<code>arg_NextHopId</code>	Identifier of next hop.
<code>arg_Mtu</code>	New maximum transmission unit for the given next hop.
<code>arg_pUserContext</code>	A pointer to a calling application-defined context.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. <code>IX_CC_IPV6_ERROR_INVALID_NEXT_HOP_ID</code>—operation failed, NHID could not be found in RTMv6 <code>IX_CC_IPV6_ERROR_RTM</code>—operation failed, RTMv6 internal failure
--------------	--

16.4.14 ix_cc_ipv6_set_flags()

This function updates flags for a given next hop and internally calls `ix_cc_rtmv6_set_flags()` API of the RTMv6 core component.

Before calling this function, the calling application needs to call `ix_cc_ipv6_async_add_next_hop()` to add next hop identifier in the RTMv6 database.

C Syntax

```
ix_error ix_cc_ipv6_set_flags(
    ix_cc_rtmv4_nhid arg_NextHopId,
    ix_uint32 arg_Flags,
    void* arg_pUserContext);
```

Input

<code>arg_NextHopId</code>	Identifier of next hop
<code>arg_Flags</code>	New flags for the given next hop
<code>arg_pUserContext</code> <code>t</code>	A pointer to a calling application-defined context.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. <code>IX_CC_IPV6_ERROR_INVALID_NEXT_HOP_ID</code>—operation failed, NHID could not be found in RTMv6 <code>IX_CC_IPV6_ERROR_INVALID_FLAGS</code>—operation failed, unsupported flags <code>IX_CC_IPV6_ERROR_RTM</code>—operation failed, RTMv6 internal failure
--------------	---

16.4.15 `ix_cc_ipv6_get_rate_limit_time()`

This library function retrieves the time interval for rate limiting in milliseconds

C Syntax

```
ix_error ix_cc_ipv6_get_rate_limit_time (
    ix_uint16* arg_pInterval,
    void* arg_pUserContext);
```

Input

<code>arg_pUserContext</code> <code>t</code>	A pointer to a calling application-defined context.
---	---

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. <code>IX_CC_IPV6_ERROR_FAILED_GET_RATE_LIMIT_TIME</code>—operation failed, failed to get sleep time from ICMP module <code>IX_ERROR_INVALID_POINTER</code>—operation failed, invalid pointer
--------------	---

16.4.16 ix_cc_ipv6_set_rate_limit_time()

This function sets the rate limit interval in milliseconds. This interval indicates how often an ICMP error packet is dequeued and sent.

C Syntax

```
ix_error ix_cc_ipv6_set_rate_limit_time (
    ix_uint16 arg_Interval,
    void* arg_pUserContext);
```

Input

<code>arg_Interval</code>	Interval in milliseconds. Zero is not a valid value.
<code>arg_pUserContext</code> <code>t</code>	A pointer to a calling application-defined context.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • <code>IX_CC_IPV6_ERROR_INVALID_RATE_LIMIT_TIME</code>—operation failed, unsupported sleep time
--------------	--

16.4.17 ix_cc_ipv6_get_queue_depth()

This library function retrieves the depth of the ICMP error message queue. That is, the maximum number of ICMP messages that can fit in the queue, not the number of messages currently in the queue.

C Syntax

```
ix_error ix_cc_ipv6_get_queue_depth (
    ix_uint16* arg_pQueueDepth,
    void* arg_pUserContext);
```

Input

<code>arg_pUserContext</code> <code>t</code>	A pointer to a calling application-defined context.
---	---

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. <code>IX_CC_IPV6_ERROR_FAILED_GET_QUEUE_DEPTH</code>—operation failed, failed to get queue depth from ICMP module. <code>IX_ERROR_INVALID_POINTER</code>—operation failed, invalid pointer
--------------	---

16.4.18 `ix_cc_ipv6_get_statistics()`

This function retrieves statistics report from IPV6 Forwarder core component—gives the statistics of IPV6 microblock, RTMv6 and IPV6 Forwarder core component. The calling application needs to allocate memory to hold data.

C Syntax

```
ix_error ix_cc_ipv6_get_statistics (
    ix_cc_ipv6_stats_data* arg_pBuffer,
    void* arg_pUserContext);
```

Input

`arg_pUserContext` A pointer to a calling application-defined context.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. <code>IX_ERROR_INVALID_POINTER</code>—operation failed, invalid pointer
--------------	---

16.4.19 `ix_cc_ipv6_set_property()`

This function sets dynamic properties of the IPv6 Forwarder core component.

C Syntax

```
ix_error ix_cc_ipv6_set_property(
    ix_uint32 arg_PropId,
    ix_cc_properties* arg_pProperties,
    void* arg_pUserContext);
```

Input

<code>arg_PropId</code>	Identifier of a property or properties to be updated for a port.
<code>arg_pProperties</code>	Pointer to dynamic property data structure, defined in section Error! Reference source not found.
<code>arg_pUserContext</code> <code>t</code>	A pointer to a calling application-defined context.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. <code>IX_CC_IPV6_ERROR_INVALID_PROP_ID</code>—operation failed, invalid property identifier <code>IX_ERROR_INVALID_POINTER</code>—operation failed, invalid pointer
--------------	--

16.4.20 `ix_cc_ipv6_perform_addr_resolution()`

This function requests the address resolution to be performed for the destination IP address contained in the packet. The buffer's meta-data should contain the index into the L2 table (the L2 ID) at which the L2 address should be stored once the Address Resolution is complete. It is expected that the ONLY client of this primitive is the Ethernet TX core component. For more information about Address Resolution, refer to [Section 53.4.5, “Neighbor Discovery”](#) in *Intel® Internet Exchange Architecture Software Building Blocks Developer's Manual*.

C Syntax

```
ix_error ix_cc_ipv6_perform_addr_resolution (
    ix_buffer_handle arg_hBuffer,
    void* arg_pUserContext);
```

Input

<code>arg_hBuffer</code>	Handle of the hardware buffer containing the packet. Since the only client of this primitive is the Ethernet TX core component, this buffer is assumed to contain an entire Ethernet frame and is processed accordingly.
<code>arg_pUserContext</code> <code>t</code>	A pointer to a calling application-defined context.

Output/Returns

Return Value

Returns a valid `ix_error`.

- `IX_SUCCESS`—the operation succeeded.
- `IX_CC_ERROR_OOM`—operation failed, The system is out of memory.
- `IX_CC_IPV6_ERROR_INVALID_PARAMETERS`—operation failed, Some of the input parameters were invalid.
- `IX_CC_IPV6_ERROR_MSG_LIBRARY`—operation failed, failure from Message Support Library

16.4.21 ix_cc_ipv6_add_neighbor()

This function updates the neighbor cache and add an entry into the L2 table. For more information about Address Resolution, refer to [Section 53.4.5, “Neighbor Discovery”](#) in *Intel® Internet Exchange Architecture Software Building Blocks Developer’s Manual*.

C Syntax

```
ix_error ix_cc_ipv6_add_neighbor(
    ix_uint16 arg_L2Index,
    ix_uint128 arg_NbrIp,
    ix_uint16 arg_OutputPort,
    ix_l2_addr arg_L2Addr,
    void* arg_pUserContext);
```

Input

arg_L2Index	The index into the L2 table at which this information should be written.
arg_NbrIp	The IP address of the neighbor.
arg_OutPutPort	The egress port through which all packets destined for the neighbor should be transmitted.
arg_L2Add	The link layer address of the neighbor.
arg_pUserContext	A pointer to a calling application-defined context.

Output/Returns

Return Value	<p>Returns a valid <code>ix_error</code>.</p> <ul style="list-style-type: none"> IX_SUCCESS—the operation succeeded. IX_CC_ERROR_OOM—operation failed, the system is out of memory. IX_CC_IPV6_INVALID_PARAMETERS—operation failed, one or more input parameters are invalid. IX_CC_IPV6_ERROR_MSG_LIBRARY—operation failed, failure from Message Support Library
--------------	---

16.4.22 ix_cc_ipv6_del_neighbor()

This function deletes the contents located at the supplied L2Index in the L2 table. Note that this may not result in the deletion of the corresponding Neighbor Cache entry maintained by the Ipv6 Forwarder core component. For more information about Address Resolution, refer to [Section 53.4.5, “Neighbor Discovery”](#) in *Intel® Internet Exchange Architecture Software Building Blocks Developer's Manual*.

C Syntax

```
ix_error ix_cc_ipv6_del_neighbor (
    ix_uint16 arg_L2Index,
    void* arg_pUserContext);
```

Input

arg_L2Index	The index into the L2 table at which information should be deleted.
arg_pUserContext	A pointer to a calling application-defined context.

Output/Returns

Return Value	Returns a valid ix_error. <ul style="list-style-type: none"> IX_SUCCESS—the operation succeeded. IX_CC_IPV6_ERROR_INVALID_PARAMETERS—operation failed, One or more input parameters were invalid. IX_CC_IPV6_ERROR_MSG_LIBRARY—operation failed, failure from Message Support Library
--------------	--

16.4.23 ix_cc_ipv6_send_icmp_error()

This function sends an ICMP error message. The hardware buffer is expected to be the IPv6 packet causing the error to be sent. The calling application must ensure that the meta-data is correctly configured and the original IPv6 header is present. The ICMP error message is sent to the source address contained in the IPv6 header.

C Syntax

```
ix_error ix_cc_ipv6_send_icmp_error(
    ix_cc_ipv6_icmp_err_type arg_IcmpErrType,
    ix_cc_ipv6_icmp_err_code arg_IcmpErrCode,
    ix_buffer_handle arg_hBuffer,
    void* arg_pUserContext);
```


Input

<code>arg_IcmpErrType</code>	The “type” field to be included in the ICMP header. Defined in Section 16.1.9 , “ <code>ix_cc_ipv6_icmp_err_type</code> ” on page 248.
<code>arg_IcmpErrCode</code>	The “code” field to be included in the ICMP header. Defined in Section 16.1.10 , “ <code>ix_cc_ipv6_icmp_err_code</code> ” on page 248.
<code>arg_hBuffer</code>	The hardware buffer handle containing the IPv6 packet causing the error.
<code>arg_pUserContext</code>	A pointer to a calling application-defined context.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. <code>IX_CC_IPV6_ERROR_INVALID_PARAMETERS</code>—operation failed, one or more of the input parameters are invalid. <code>IX_CC_IPV6_ERROR_MSG_LIBRARY</code>—operation failed, failure from Message Support Library
--------------	---

16.4.24 `ix_cc_ipv6_send_icmp_info_message()`

This function sends ICMP informational messages. The calling application must ensure that the type and code provided are valid. The IPv6 Forwarder core component only does basic validity tests on the type and code fields, in the interest of forward compatibility. ICMP informational messages are NOT rate limited.

C Syntax

```
ix_error ix_cc_ipv6_send_icmp_info_message (
    ix_uint128 arg_SrcIpAddr,
    ix_uint128 arg_DstIpAddr,
    ix_uint8 arg_Type,
    ix_uint8 arg_Code,
    ix_uint32 arg_PayloadLength,
    ix_uint8*arg_pPayload,
    void* arg_pUserContext);
```

Input

<code>arg_SrcIpAddr</code>	The source IP address to be used in the IPv6 header.
<code>arg_DstIpAddr</code>	The destination IP address to be used in the IPv6 header.
<code>arg_Type</code>	The ICMP message type to be inserted in the ICMP header.

Input

<code>arg_Code</code>	The ICMP message code to be inserted in the ICMP header.
<code>arg_PayloadLength</code>	The length of the payload to be included in the ICMP packet.
<code>arg_pPayload</code>	Pointer to a buffer containing the payload to be transmitted.
<code>arg_pUserContext</code>	A pointer to a calling application-defined context.

The IPV6 to IPV4 Tunneling core component helps the tunneling microblocks implement the following types of tunneling:

- Configured tunnels as defined in RFC 2893
- Automatic tunnels as defined in RFC 2893
- 6to4 tunnels as defined in RFC 3056

The following functionality is provided:

- Configuration of the tunneling microblocks.
- Setup and management of tunneling next hop information.
- Provides packet handlers to receive packets from the tunneling microblocks and from other core components.
- Provides message handlers to allow configuration of tunneling information.
- Provides reassembly and decapsulation of fragmented tunneled packets.
- Responsible for sending ICMPv6 “Packet too Big” error messages if packet length exceeds recorded path MTU for a tunnel.
- Responsible for reflection of ICMP error messages to IPv6 sender.
- Responsible for reporting tunneling statistics.

For complete details see [Chapter 54, “IPv6 To IPv4 Tunneling Core Component”](#) of the *Intel® Internet Exchange Architecture Software Building Blocks Developer’s Manual*.

17.1 Data Structures, Types and Macros

Table 17-1 lists the IPv6-IPv4 tunneling data structures, types and macros.

Table 17-1. IPv6-IPv4 Tunneling Data Structures, Types and Macros

Data Structures, Types and Macros	Description
<code>ix_cc_v6v4_tunnel_handle</code>	Tunneling core component's handle to a start or end tunnel next hop information
<code>ix_cc_v6v4_end_tunnel_config</code>	Contains the configuration information for an end tunnel endpoint
<code>ix_cc_v6v4_end_tunnel_info</code>	Contains the information describing an end tunnel endpoint
<code>End Tunnel Flags</code>	Defined for end tunnel endpoints
<code>ix_cc_v6v4_ingress_source_entry</code>	Describes an entry in an ingress source validation list for an end tunnel endpoint
<code>ix_cc_v6v4_interface_id</code>	Defines an identifier for an interface used for tunneling
<code>ix_cc_v6v4_start_tunnel_config</code>	Contains the configuration information for a start tunnel endpoint

Table 17-1. IPv6-IPv4 Tunneling Data Structures, Types and Macros

Data Structures, Types and Macros	Description
<code>ix_cc_v6v4_start_tunnel_info</code>	Contains all information associated with a start tunnel endpoint
Start Tunnel Flags	defined for start tunnel endpoints
<code>ix_cc_v6v4_statistics</code>	Describes the packet counters maintained by the Tunneling core component
Option Values	Selects boolean options in the Tunneling core component API

17.1.1 `ix_cc_v6v4_tunnel_handle`

The Tunneling core component's handle to a start or end tunnel next hop information. This value is assigned by the core component when a tunnel is created.

C Syntax

```
typedef ix_uint32 ix_cc_v6v4_tunnel_handle;
```

17.1.2 `ix_cc_v6v4_end_tunnel_config`

This structure definition contains the configuration information for an end tunnel endpoint.

C Syntax

```
typedef struct ix_s_cc_v6v4_end_tunnel_config {
    ix_uint16 flags;
} ix_cc_v6v4_end_tunnel_config;
```

17.1.3 `ix_cc_v6v4_end_tunnel_info`

This structure definition contains the information describing an end tunnel endpoint.

C Syntax

```
typedef struct ix_s_cc_v6v4_end_tunnel_info {
    ix_cc_v6v4_tunnel_handle hTunnel;
    ix_cc_v6v4_end_tunnel_config config;
} ix_cc_v6v4_end_tunnel_info;
```

17.1.4 End Tunnel Flags

The following flags are defined for end tunnel endpoints:

C Syntax

```
#define IX_CC_V6V4_DECAP_AUTO0x0001
#define IX_CC_V6V4_SRC_VALIDATE0x0002
#define IX_CC_V6V4_DECAP_6TO40x0004
#define IX_CC_V6V4_DECAP_TOS_V60x0008
```

17.1.5 ix_cc_v6v4_ingress_source_entry

This structure definition describes an entry in an ingress source validation list for an end tunnel endpoint.

C Syntax

```
typedef struct ix_s_cc_v6v4_ingress_source_entry {
    ix_uint32 ipAddr; /* Source IP address */
    ix_uint8 maskLength; /* Number of meaningful bits in ipAddr */
} ix_cc_v6v4_ingress_source_entry;
```

17.1.6 ix_cc_v6v4_interface_id

This structure defines an identifier for an interface used for tunneling.

C Syntax

```
typedef struct ix_s_cc_v6v4_interface_id {
    ix_uint16 bladeId; /* Blade Id */
    ix_uint16 portId; /* Port Id */
} ix_cc_v6v4_interface_id;
```

17.1.7 ix_cc_v6v4_start_tunnel_config

This structure definition contains the configuration information for a start tunnel endpoint.

C Syntax

```
typedef struct ix_s_cc_v6v4_start_tunnel_config {
    ix_uint16 flags;
    ix_uint16 pathMtu;
    ix_cc_v6v4_interface_id localInterface;
    ix_uint32 remoteIpv4Address;
    ix_uint32 subnetBroadcast;
    ix_uint8 ttl;
    ix_uint8 tos;
} ix_cc_v6v4_start_tunnel_config;
```

17.1.7.1 IX_CC_V6V4_PATH_MTU_INTERFACE

When creating a start tunnel entry (See [Section 17.3.11 ix_cc_v6v4_async_add_start_tunnel\(\)](#), on page 321 and [Section 17.4.11 ix_cc_v6v4_add_start_tunnel\(\)](#), on page 342), the calling application can set the `pathMtu` to the special value `IX_CC_V6V4_PATH_MTU_INTERFACE`. This is done in order to specify that the core component should maintain the path MTU for the tunnel as the MTU for the interface, corresponding to the specified blade and port.

If the calling application later sets the path MTU to a different value (see [Section 17.3.16 ix_cc_v6v4_async_set_mtu\(\)](#), on page 327 and [Section 17.4.16 ix_cc_v6v4_set_mtu\(\)](#), on page 346), then the core component no longer maintains the tunnel MTU as the interface MTU. The calling application can reinstate this feature by setting the tunnel MTU back to `IX_CC_V6V4_PATH_MTU_INTERFACE`.

Note: The core component uses this feature to determine whether the encapsulation process should set the DON'T FRAGMENT bit in the IPv4 header. If the path MTU is maintained as an underlying interface, then the DF bit is set in the encapsulating header, as specified in RFC 2893, section 3.2. If the calling application is managing the path MTU, then the DF bit is set according to the algorithm in RFC2893 section 3.2.

When retrieving tunnel information, the core component always returns the actual path MTU currently in use for the tunnel.

C Syntax

```
#define IX_CC_V6V4_PATH_MTU_INTERFACE0
```

17.1.7.2 IX_CC_V6V4_BROADCAST_INTERFACE

When creating a start tunnel entry ([Section 17.3.11 ix_cc_v6v4_async_add_start_tunnel\(\)](#), on page 321 and [Section 17.4.11 ix_cc_v6v4_add_start_tunnel\(\)](#), on page 342), the calling application can set `subnetBroadcast` to the special value `IX_CC_V6V4_BROADCAST_INTERFACE`. This is done in order to specify that the core component should maintain the subnet broadcast address as the broadcast address of the interface, corresponding to the specified blade and port.

If the calling application later sets the subnet broadcast address to a different value (see [Section 17.3.17 ix_cc_v6v4_async_set_subnet_broadcast\(\)](#), on page 327 and [Section 17.4.17 ix_cc_v6v4_set_subnet_broadcast\(\)](#), on page 347), then the core component no longer maintains the tunnel subnet broadcast address as the interface broadcast address. The calling application can reinstate this feature by setting the tunnel subnet broadcast address back to `IX_CC_V6V4_BROADCAST_INTERFACE`.

Note: The subnet broadcast address is only used in the encapsulation process for header validation for automatic and v6tov4 tunnels. When retrieving tunnel information, the core component always returns the actual subnet broadcast address currently in use for the tunnel.

C Syntax

```
#define IX_CC_V6V4_BROADCAST_INTERFACE0
```

17.1.8 ix_cc_v6v4_start_tunnel_info

This structure definition contains all information associated with a start tunnel endpoint. This information is returned to the calling application from APIs that retrieve tunnel information.

C Syntax

```
typedef struct ix_s_cc_v6v4_start_tunnel_info {
    ix_cc_v6v4_tunnel_handlehTunnel;
    ix_cc_v6v4_start_tunnel_config config;
    ix_uint8 trackMtu;
    ix_uint8 trackBroadcast;
} ix_cc_v6v4_start_tunnel_info;
```

The additional fields `trackMtu` and `trackBroadcast` in this structure indicate whether the core component is currently maintaining the value as the corresponding interface value.

17.1.9 Start Tunnel Flags

The following flags are defined for start tunnel endpoints:

C Syntax

```
#define IX_CC_V6V4_ENCAP_AUTO0x0001
#define IX_CC_V6V4_ENCAP_6TO40x0002
#define IX_CC_V6V4_ENCAP_TOS_V60x0004
```

17.1.10 ix_cc_v6v4_statistics

This structure definition describes the packet counters maintained by the Tunneling core component:

C Syntax

```
typedef struct ix_cc_v6v4_statistics {
    ix_uint32 decapMbDroppedPkts; /* Packets dropped by decap block */
    ix_uint32 decapMbExcpPkts; /* Packets sent as exceptions
                                by decap block */
    ix_uint32 encapMbDroppedPkts; /* Packets dropped by encap block */
    ix_uint32 encapMbExcpPkts; /* Packets sent as exceptions
                                by encap block */
    ix_uint32 localPkts; /* Local non-tunneled packets sent to stack */
    ix_uint32 autoTunnelPkts; /* Automatic tunneled packets
                               sent to stack */
    ix_uint32 fragmentsRcvd; /* Tunneled packet fragments received */
    ix_uint32 reassembledPkts; /* Tunneled packets reassembled */
    ix_uint32 pktTooBigMsgs; /* Packet too Big messages sent */
    ix_uint32 encapsulatedPkts; /* Packets encapsulated */
    ix_uint32 decapsulatedPkts; /* Packets decapsulated */
};
```

17.1.11 Option Values

The following definitions are used to select boolean options in the Tunneling core component APIs.

C Syntax

```
#define IX_CC_V6V4_SELECT1 /* Selects an option */
#define IX_CC_V6V4_CLEAR 0 /* Deselects an option */
```

17.2 Core Component Infrastructure API

Table 17-2 lists the Tunnelling core component Infrastructure API.

Table 17-2. IPv6 to IPV4 Tunnelling Core Component Infrastructure AP

API	Description
<code>ix_cc_v6v4_init()</code>	Initializes the Tunneling core component
<code>ix_cc_v6v4_fini()</code>	terminate the services of the Tunneling core component
<code>ix_cc_v6v4_msg_handler()</code>	Receives messages from the calling applications
<code>ix_cc_v6v4_microblock_pkt_handler()</code>	Handles packets received from the tunneling microblocks
<code>ix_cc_v6v4_ipv6_pkt_handler()</code>	Handles packets received from the IPv6 forwarder that might need encapsulation
<code>ix_cc_v6v4_ipv4_pkt_handler()</code>	Handles packets received from the IPv4 forwarder that might need decapsulation

17.2.1 `ix_cc_v6v4_init()`

This function is called once by the Core Component Infrastructure to initialize the Tunneling core component. This routine performs all initialization required for IPv6 over IPv4 tunneling, including patching symbols for the microblocks and allocating and initializing all required memory.

C Syntax

```
ix_error ix_cc_v6v4_init (
    ix_cc_handle arg_hCcHandle,
    void ** arg_ppContext);
```

Input

`arg_hCcHandle` Handle to IPv6 to IPv4 Tunneling core component, created by the Core Component Infrastructure. This value is used subsequently to request services from the Core Component Infrastructure.

Input/Output

<code>arg_ppContext</code>	Both an input and output parameter <ul style="list-style-type: none"> Input—this points to a generic core component initialization context, from which the core component can obtain information that might be of interest to all core components. Output—On successful completion of this function the Tunneling core component updates this location with a pointer to its context information buffer. This buffer is allocated by the Tunneling core component and contains data that it uses internally. When the Tunneling core component is to be destroyed, this pointer is passed to the <code>ix_cc_v6v4_fini()</code> routine so that it can release its resources.
----------------------------	---

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. <code>IX_CC_ERROR_OOM</code>—operation failed, memory allocation failure <code>IX_CC_ERROR_OOR</code>—operation failed, resource allocation failure <code>IX_CC_ERROR_RANGE</code>—operation failed, a parameter from the registry has an invalid value <code>IX_CC_ERROR_NULL</code>—operation failed, <code>arg_ppContext</code> is NULL <code>IX_CC_ERROR_INTERNAL</code>—operation failed, internal error occurred
--------------	--

17.2.2 `ix_cc_v6v4_fini()`

This function is called by the Core Component Infrastructure to terminate the services of the Tunneling core component. This API releases all the resources allocated for IPv6 over IPv4 tunneling. It is assumed that the microengines have been shut down before this routine is called.

C Syntax

```
ix_error ix_cc_v6v4_fini(
    ix_cc_handle arg_hCcHandle
    void * arg_pContext);
```

Input

<code>arg_hCcHandle</code>	Handle to IPv6v4 Tunneling core component.
<code>arg_pContext</code>	Pointer to the core component context information, which was allocated in <code>ix_cc_v6v4_init()</code> .

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • <code>IX_CC_ERROR_NULL</code>—operation failed, invalid input pointer • <code>IX_CC_ERROR_RANGE</code>—operation failed, invalid data in <code>arg_pContext</code> • <code>IX_CC_ERROR_INTERNAL</code>—operation failed, internal error occurred
--------------	--

17.2.3 `ix_cc_v6v4_msg_handler()`

This function is the message handler routine for the Tunneling core component. The Tunneling core component receives messages from the calling applications through this function.

C Syntax

```
ix_error ix_cc_v6v4_msg_handler(
    ix_buffer_handle arg_hMsgInfo,
    ix_uint32 arg_MsgType,
    void * arg_pContext);
```

Input

<code>arg_hMsgInfo</code>	Handle to buffer containing information relevant to the message.
<code>arg_MsgType</code>	Message type. Table 17-3 lists the supported message types.
<code>arg_pContext</code>	Pointer to the core component context information allocated in <code>ix_cc_v6v4_init()</code> .

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • <code>IX_CC_ERROR_NULL</code>—operation failed, invalid pointer • <code>IX_CC_ERROR_UNDEFINED_MSG</code>—operation failed, invalid message type • <code>IX_CC_ERROR_OOM</code>—operation failed, out of memory • <code>IX_CC_ERROR_INTERNAL</code>—operation failed, internal error occurred
--------------	---

[Table 17-3](#) lists the supported message types.

Table 17-3. Message Types for the Tunneling Core Component

Messages	Description
IX_CC_V6V4_MSG_ADD_END_TUNNEL	Adds a new end tunnel endpoint.
IX_CC_V6V4_MSG_DELETE_END_TUNNEL	Deletes an end tunnel endpoint.
IX_CC_V6V4_MSG_SET_DECAP_TOS_OPTION	Enables or disables the TOS/DSCP option for an end tunnel endpoint.
IX_CC_V6V4_MSG_SET_SRC_VALIDATION	Enables or disables ingress source address validation for an end tunnel endpoint.
IX_CC_V6V4_MSG_GET_END_TUNNEL	Gets the configuration information for an end tunnel endpoint.
IX_CC_V6V4_MSG_ADD_ALLOWED_SOURCE	Add a prefix to the ingress source validation list for a configured tunnel.
IX_CC_V6V4_MSG_DELETE_ALLOWED_SOURCE	Delete a prefix from the ingress source validation list for a configured tunnel.
IX_CC_V6V4_MSG_GET_ALLOWED_SOURCES	Retrieves a list of all source prefixes allowed for an end tunnel endpoint.
IX_CC_V6V4_MSG_GET_ALLOWED_SOURCES	Retrieves a list of all source prefixes allowed for an end tunnel endpoint.
IX_CC_V6V4_MSG_DUMP_END_TUNNELS	Retrieves a list of all end tunnel endpoints.
IX_CC_V6V4_MSG_ADD_START_TUNNEL	Adds a new start tunnel endpoint.
IX_CC_V6V4_MSG_DELETE_START_TUNNEL	Deletes a start tunnel endpoint.
IX_CC_V6V4_MSG_SET_TTL	Sets the TTL for a start tunnel endpoint.
IX_CC_V6V4_MSG_SET_ENCAP_TOS_OPTION	Sets the TOS option for a start tunnel endpoint.
IX_CC_V6V4_MSG_SET_TOS	Sets the TOS value for a start tunnel endpoint.
IX_CC_V6V4_MSG_SET_MTU	Sets the path MTU for a start tunnel endpoint.
IX_CC_V6V4_MSG_SET_SUBNET_BROADCAST	Sets the subnet broadcast address for a start tunnel endpoint.
IX_CC_V6V4_MSG_GET_START_TUNNEL	Gets the configuration information for a start tunnel endpoint.
IX_CC_V6V4_MSG_DUMP_START_TUNNELS	Retrieves a list of all start tunnel endpoints.
IX_CC_V6V4_MSG_GET_STATISTICS	Retrieves microblock packet counters.
IX_CC_COMMON_MSG_ID_PROP_UPDATE	Updates interface properties (this is a message ID common to all core components—the Tunneling core component handles this in order to maintain interface addresses and MTU values).

17.2.4 ix_cc_v6v4_microblock_pkt_handler()

This function is the packet handler routine for packets received from the tunneling microblocks.

C Syntax

```
ix_error ix_cc_v6v4_microblock_pkt_handler(
    ix_buffer_handle arg_hPacket,
    ix_uint32 arg_ExceptionCode,
    void * arg_pContext);
```

Input

<code>arg_hPacket</code>	Handle to buffer containing exception packet.
<code>arg_ExceptionCode</code>	Exception code set by microblock.
<code>arg_pContext</code>	Pointer to the core component context information allocated in ix_cc_v6v4_init() .

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. <code>IX_CC_ERROR_NULL</code>—operation failed, invalid pointer <code>IX_CC_ERROR_UNDEFINED_EXCEP</code>—operation failed, exception code is invalid <code>IX_CC_ERROR_OOM</code>—operation failed, out of memory <code>IX_CC_ERROR_INTERNAL</code>—operation failed, internal error occurred
--------------	--

17.2.5 ix_cc_v6v4_ipv6_pkt_handler()

This function is the packet handler routine for packets received from the IPv6 forwarder that might need encapsulation.

C Syntax

```
ix_error ix_cc_v6v4_ipv6_pkt_handler(
    ix_buffer_handle arg_hPacket,
    ix_uint32 arg_ExceptionCode,
    void * arg_pContext);
```

Input

<code>arg_hPacket</code>	Handle to buffer containing IPv6 packet.
<code>arg_ExceptionCode</code>	Not used by the function.
<code>arg_pContext</code>	Pointer to the core component context information allocated in <code>ix_cc_v6v4_init()</code> .

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. <code>IX_CC_ERROR_NULL</code>—operation failed, invalid pointer <code>IX_CC_ERROR_UNDEFINED_EXCEP</code>—operation failed, exception code is invalid <code>IX_CC_ERROR_OOM</code>—operation failed, out of memory <code>IX_CC_ERROR_INTERNAL</code>—operation failed, internal error occurred
--------------	--

17.2.6 `ix_cc_v6v4_ipv4_pkt_handler()`

This function is the packet handler routine for packets received from the IPv4 forwarder that might need decapsulation.

C Syntax

```
ix_error ix_cc_v6v4_ipv4_pkt_handler (
    ix_buffer_handle arg_hPacket,
    ix_uint32 arg_ExceptionCode,
    void * arg_pContext);
```

Input

<code>arg_hPacket</code>	A handle to buffer containing IPv4 packet.
<code>arg_ExceptionCode</code>	Not used by the function.
<code>arg_pContext</code>	A pointer to the core component context information, which was allocated in <code>ix_cc_v6v4_init()</code> .

Output/Returns

Return Value	<p>Returns a valid <code>ix_error</code>.</p> <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • <code>IX_CC_ERROR_NULL</code>—operation failed, invalid pointer • <code>IX_CC_ERROR_UNDEFINED_EXCEP</code>—operation failed, exception code is invalid • <code>IX_CC_ERROR_OOM</code>—operation failed, out of memory • <code>IX_CC_ERROR_INTERNAL</code>—operation failed, internal error occurred
--------------	--

17.3 Message Helper API

The message helper API is used by calling applications to construct and send messages asynchronously to the Tunneling core component. [Table 17-4](#) lists the IPv6 to IPv4 Tunneling core component message helper API.

Table 17-4. IPv6 to IPv4 Tunneling Core Component Message Helper API

API	Description
<code>ix_cc_v6v4_async_add_end_tunnel()</code>	Sends a message of type <code>IX_CC_V6V4_MSG_ADD_END_TUNNEL</code> to the Tunneling core component
<code>ix_cc_v6v4_async_delete_end_tunnel()</code>	Sends a message of type <code>IX_CC_V6V4_MSG_DELETE_END_TUNNEL</code> to the Tunneling core component
<code>ix_cc_v6v4_async_set_decap_tos_option()</code>	Sends a message of type <code>IX_CC_V6V4_MSG_SET_DECAP_TOS_OPTION</code> to the Tunneling core component
<code>ix_cc_v6v4_async_set_src_validation()</code>	Sends a message of type <code>IX_CC_V6V4_MSG_SET_SRC_VALIDATION</code> to the Tunneling core component.
<code>ix_cc_v6v4_async_get_end_tunnel()</code>	Sends a message of type <code>IX_CC_V6V4_MSG_GET_END_TUNNEL</code> to the Tunneling core component.
<code>ix_cc_v6v4_async_add_allowed_source()</code>	Sends a message of type <code>IX_CC_V6V4_MSG_ADD_ALLOWED_SOURCE</code> to the Tunneling core component.
<code>ix_cc_v6v4_async_delete_allowed_source()</code>	Sends a message of type <code>IX_CC_V6V4_MSG_DELETE_ALLOWED_SOURCE</code> to the Tunneling core component.
<code>ix_cc_v6v4_async_get_allowed_sources()</code>	Sends a message of type <code>IX_CC_V6V4_MSG_GET_ALLOWED_SOURCES</code> to the Tunneling core component.
<code>ix_cc_v6v4_async_dump_end_tunnels()</code>	Sends a message of type <code>IX_CC_V6V4_MSG_DUMP_END_TUNNELS</code> to the Tunneling core component.
<code>ix_cc_v6v4_async_clear_allowed_sources()</code>	Sends a message of type <code>IX_CC_V6V4_MSG_CLEAR_ALLOWED_SOURCES</code> to the Tunneling core component.

Table 17-4. IPv6 to IPv4 Tunnelling Core Component Message Helper API (Continued)

API	Description
<code>ix_cc_v6v4_async_add_start_tunnel()</code>	Sends a message of type <code>IX_CC_V6V4_MSG_ADD_START_TUNNEL</code> to the Tunneling core component.
<code>ix_cc_v6v4_async_delete_start_tunnel()</code>	Sends a message of type <code>IX_CC_V6V4_MSG_DELETE_START_TUNNEL</code> to the Tunneling core component.
<code>ix_cc_v6v4_async_set_ttl()</code>	Sends a message of type <code>IX_CC_V6V4_MSG_SET_TTL</code> to the Tunneling core component.
<code>ix_cc_v6v4_async_set_encap_tos_option()</code>	Sends a message of type <code>IX_CC_V6V4_MSG_SET_ENCAP_TOS_OPTION</code> to the Tunneling core component.
<code>ix_cc_v6v4_async_set_tos()</code>	Sends a message of type <code>IX_CC_V6V4_MSG_SET_TOS</code> to the Tunneling core component.
<code>ix_cc_v6v4_async_set_mtu()</code>	Sends a message of type <code>IX_CC_V6V4_MSG_SET_MTU</code> to the Tunneling core component.
<code>ix_cc_v6v4_async_set_subnet_broadcast()</code>	Sends a message of type <code>IX_CC_V6V4_MSG_SET_SUBNET_BROADCAST</code> to the Tunneling core component.
<code>ix_cc_v6v4_async_get_start_tunnel()</code>	Sends a message of type <code>IX_CC_V6V4_MSG_GET_START_TUNNEL</code> to the Tunneling core component.
<code>ix_cc_v6v4_async_dump_start_tunnels()</code>	Sends a message of type <code>IX_CC_V6V4_MSG_DUMP_START_TUNNELS</code> to the Tunneling core component.
<code>ix_cc_v6v4_async_get_statistics()</code>	Sends a message of type <code>IX_CC_V6V4_MSG_GET_STATISTICS</code> to the Tunneling core component.

17.3.1 `ix_cc_v6v4_async_add_end_tunnel()`

Sends a message of type `IX_CC_V6V4_MSG_ADD_END_TUNNEL` to the Tunneling core component.

C Syntax

```
ix_error ix_cc_v6v4_async_add_end_tunnel (
    ix_cc_v6v4_end_tunnel_config * arg_pConfig,
    ix_cc_v6v4_cb_add_tunnel arg_Callback,
    void * arg_pClientContext);
```

Input

<code>arg_pConfig</code>	Pointer to configuration information for the tunnel.
<code>arg_Callback</code>	Pointer to a calling application-supplied callback function.
<code>arg_pClientContext</code>	Pointer to a calling application-defined context passed to the callback function. See ix_cc_v6v4_cb_add_tunnel .

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • <code>IX_CC_V6V4_ERROR_MSG_LIBRARY</code>—operation failed, failure from message support library • <code>IX_CC_ERROR_NULL</code>—operation failed, invalid pointer
--------------	---

17.3.1.1 `ix_cc_v6v4_cb_add_tunnel`

This is the function prototype for the callback functions used by the message helper APIs that add start or end tunnels.

C Syntax

```
void ix_cc_v6v4_cb_add_tunnel(
    ix_error arg_Result,
    void * arg_pClientContext,
    ix_cc_v6v4_tunnel_handle hTunnel);
```

Input

<code>arg_Result</code>	<ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—returned if the operation succeeds. • A valid <code>ix_error</code>—returned if the operation fails.
<code>arg_pClientContext</code>	Context pointer supplied by calling application in the original message API call.
<code>hTunnel</code>	Handle to tunnel. The tunnel handle must be used to identify the tunnel in subsequent message API calls.

17.3.2 `ix_cc_v6v4_async_delete_end_tunnel()`

Sends a message of type `IX_CC_V6V4_MSG_DELETE_END_TUNNEL` to the Tunneling core component.

C Syntax

```
ix_error ix_cc_v6v4_async_delete_end_tunnel(
    ix_cc_v6v4_tunnel_handle arg_hTunnel,
    ix_cc_v6v4_cb arg_Callback,
    void * arg_pClientContext);
```


Input

<code>arg_hTunnel</code>	Handle to end tunnel.
<code>arg_Callback</code>	Pointer to a calling application-supplied callback function. See ix_cc_v6v4_cb .
<code>arg_pClientContext</code>	Pointer to a calling application-defined context to the callback function.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. <code>IX_CC_V6V4_ERROR_MSG_LIBRARY</code>—operation failed, failure from message support library <code>IX_CC_ERROR_NULL</code>—operation failed, invalid pointer
--------------	---

17.3.2.1 [ix_cc_v6v4_cb](#)

This is the function prototype for the callback functions used by the message helper APIs when no message specific data needs to be returned to the calling application.

C Syntax

```
void ix_cc_v6v4_cb (
    ix_error arg_Result,
    void * arg_pClientContext);
```

Input

<code>arg_Result</code>	<ul style="list-style-type: none"> <code>IX_SUCCESS</code>—returned if the operation succeeds. A valid <code>ix_error</code>—returned if the operation fails.
<code>arg_pClientContext</code>	Context pointer supplied by a calling application in the original message API call.

17.3.3 ix_cc_v6v4_async_set_decap_tos_option()

Sends a message of type IX_CC_V6V4_MSG_SET_DECAP_TOS_OPTION to the Tunneling core component. If selected, this option causes the tunnel decapsulation process in the microblocks and the core component to copy the IPv4 TOS field into the IPv6 traffic class field. If not selected, the decapsulation process does not modify the IPv6 traffic class.

C Syntax

```
ix_error ix_cc_v6v4_async_set_decap_tos_option (
    ix_cc_v6v4_tunnel_handle arg_hTunnel,
    ix_uint8 arg_Selection,
    ix_cc_v6v4_cb arg_Callback,
    void * arg_pClientContext);
```

Input

arg_hTunnel	Handle to end tunnel.
arg_Selection	IX_CC_V6V4_SELECT to select the option or IX_CC_V6V4_CLEAR to deselect the option.
arg_Callback	Pointer to a calling application-supplied callback function. See ix_cc_v6v4_cb .
arg_pClientContext	Pointer to a calling application-defined context passed to the callback function.

Output/Returns

Return Value	Returns a valid ix_error. <ul style="list-style-type: none"> IX_SUCCESS—the operation succeeded. IX_CC_V6V4_ERROR_MSG_LIBRARY—operation failed, failure from message support library IX_CC_ERROR_NULL—operation failed, invalid pointer
--------------	--

17.3.4 ix_cc_v6v4_async_set_src_validation()

Sends a message of type `IX_CC_V6V4_MSG_SET_SRC_VALIDATION` to the Tunneling core component. If selected, this option causes the tunnel decapsulation process in the microblocks and the core component to validate the source address of the packets received on configured or 6to4 tunnels against a list of valid prefixes. If not selected, the decapsulation process does not perform this validation.

C Syntax

```
ix_error ix_cc_v6v4_async_set_src_validation(
    ix_cc_v6v4_tunnel_handle arg_hTunnel,
    ix_uint8 arg_Selection,
    ix_cc_v6v4_cb arg_Callback,
    void * arg_pClientContext);
```

Input

<code>arg_hTunnel</code>	Handle to end tunnel.
<code>arg_Selection</code>	<code>IX_CC_V6V4_SELECT</code> to select ingress source validation, or <code>IX_CC_V6V4_CLEAR</code> to deselect ingress source validation for the tunnel.
<code>arg_Callback</code>	Pointer to a calling application-supplied callback function. See ix_cc_v6v4_cb .
<code>arg_pClientContext</code>	Pointer to a calling application-defined context passed to the callback function.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. <code>IX_CC_V6V4_ERROR_MSG_LIBRARY</code>—operation failed, failure from message support library <code>IX_CC_ERROR_NULL</code>—operation failed, invalid pointer
--------------	---

17.3.5 ix_cc_v6v4_async_get_end_tunnel()

Sends a message of type `IX_CC_V6V4_MSG_GET_END_TUNNEL` to the Tunneling core component.

C Syntax

```
ix_error ix_cc_v6v4_async_get_end_tunnel (
    ix_cc_v6v4_tunnel_handle arg_hTunnel,
    ix_cc_v6v4_cb_end_tunnel arg_Callback,
    void * arg_pClientContext);
```

Input

<code>arg_hTunnel</code>	Handle to end tunnel.
<code>arg_Callback</code>	Pointer to a calling application-supplied callback function.
<code>arg_pClientContext</code>	Pointer to a calling application-defined context passed to the callback function. See ix_cc_v6v4_cb_end_tunnel .

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. <code>IX_CC_V6V4_ERROR_MSG_LIBRARY</code>—operation failed, failure from message support library <code>IX_CC_ERROR_NULL</code>—operation failed, invalid pointer
--------------	---

17.3.5.1 [ix_cc_v6v4_cb_end_tunnel](#)

The function prototype for message-handler callback function provided by the calling application.

C Syntax

```
ix_error (*ix_cc_v6v4_cb_end_tunnel) (
    ix_error arg_Result,
    void * arg_pClientContext,
    ix_cc_v6v4_end_tunnel_info * arg_pTunnelInfo);
```

Input

<code>arg_Result</code>	<ul style="list-style-type: none"> <code>IX_SUCCESS</code>—returned if the operation succeeds. A valid <code>ix_error</code>—returned if the operation fails.
<code>arg_pClientContext</code>	Context pointer supplied by a calling application in the original message API call.
<code>arg_pTunnelInfo</code>	Pointer to structure containing end tunnel configuration information.

17.3.6 ix_cc_v6v4_async_add_allowed_source()

Sends a message of type IX_CC_V6V4_MSG_ADD_ALLOWED_SOURCE to the Tunneling core component.

C Syntax

```
ix_error ix_cc_v6v4_async_add_allowed_source(
    ix_cc_v6v4_tunnel_handle arg_hTunnel,
    ix_cc_v6v4_ingress_source_entry * arg_pEntry,
    ix_cc_v6v4_cb arg_Callback,
    void * arg_pClientContext);
```

Input

arg_hTunnel	Handle to end tunnel.
arg_pEntry	Pointer to ingress source entry to be added.
arg_Callback	Pointer to a calling application-supplied callback function. See ix_cc_v6v4_cb .
arg_pClientContext	Pointer to a calling application-defined context passed to the callback function.

Output/Returns

Return Value	Returns a valid ix_error. <ul style="list-style-type: none"> IX_SUCCESS—the operation succeeded. IX_CC_V6V4_ERROR_MSG_LIBRARY—operation failed, failure from message support library IX_CC_ERROR_NULL—operation failed, invalid pointer
--------------	--

17.3.7 ix_cc_v6v4_async_delete_allowed_source()

Sends a message of type IX_CC_V6V4_MSG_DELETE_ALLOWED_SOURCE to the Tunneling core component.

C Syntax

```
ix_error ix_cc_v6v4_async_delete_allowed_source(
    ix_cc_v6v4_tunnel_handle arg_hTunnel,
    ix_cc_v6v4_ingress_source_entry * arg_pEntry,
    ix_cc_v6v4_cb arg_Callback,
    void * arg_pClientContext);
```

Input

<code>arg_hTunnel</code>	Handle to end tunnel.
<code>arg_pEntry</code>	Pointer to ingress source entry to be deleted.
<code>arg_Callback</code>	Pointer to a calling application-supplied callback function. See ix_cc_v6v4_cb .
<code>arg_pClientContext</code>	Pointer to a calling application-defined context passed to the callback function.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. <code>IX_CC_V6V4_ERROR_MSG_LIBRARY</code>—operation failed, failure from message support library <code>IX_CC_ERROR_NULL</code>—operation failed, invalid pointer
--------------	---

17.3.8 `ix_cc_v6v4_async_get_allowed_sources()`

Sends a message of type `IX_CC_V6V4_MSG_GET_ALLOWED_SOURCES` to the Tunneling core component.

C Syntax

```
ix_error ix_cc_v6v4_async_get_allowed_sources(
    ix_cc_v6v4_tunnel_handle arg_hTunnel,
    ix_cc_v6v4_cb_get_sources arg_Callback,
    void * arg_pClientContext);
```

Input

<code>arg_hTunnel</code>	Handle to end tunnel.
<code>arg_Callback</code>	Pointer to a calling application-supplied callback function. See ix_cc_v6v4_cb_get_sources .
<code>arg_pClientContext</code>	Pointer to a calling application-defined context passed to the callback function.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • <code>IX_CC_V6V4_ERROR_MSG_LIBRARY</code>—operation failed, failure from message support library • <code>IX_CC_ERROR_NULL</code>—operation failed, invalid pointer
--------------	---

17.3.8.1 `ix_cc_v6v4_cb_get_sources`

The function prototype for message-handler callback function provided by the calling application.

C Syntax

```
ix_error (*ix_cc_v6v4_cb_get_sources) (
    ix_error arg_Result,
    void * arg_pClientContext,
    ix_uint32 arg_NumEntries,
    ix_cc_v6v4_ingress_source_entry * arg_paEntries);
```

Input

<code>arg_Result</code>	<ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—returned if the operation succeeds. • A valid <code>ix_error</code>—returned if the operation fails.
<code>arg_pClientContext</code>	Context pointer supplied by the calling application in the original message API call.
<code>arg_NumEntries</code>	Number of entries in array pointed to by <code>arg_paEntries</code> .
<code>arg_paEntries</code>	Pointer to array of allowed source addresses.

17.3.9 `ix_cc_v6v4_async_dump_end_tunnels()`

Sends a message of type `IX_CC_V6V4_MSG_DUMP_END_TUNNELS` to the Tunneling core component.

C Syntax

```
ix_error ix_cc_v6v4_async_dump_end_tunnels(
    ix_cc_v6v4_cb_dump_end_tunnels arg_Callback,
    void * arg_pClientContext);
```

Input

<code>arg_Callback</code>	Pointer to a calling application-supplied callback function. See ix_cc_v6v4_cb_dump_end_tunnels .
<code>arg_pClientContext</code>	Pointer to a calling application-defined context passed to the callback function.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. <code>IX_CC_V6V4_ERROR_MSG_LIBRARY</code>—operation failed, failure from message support library <code>IX_CC_ERROR_NULL</code>—operation failed, invalid pointer
--------------	---

17.3.9.1 [ix_cc_v6v4_cb_dump_end_tunnels](#)

The function prototype for message-handler callback function provided by the calling application.

C Syntax

```
ix_error (*ix_cc_v6v4_cb_dump_end_tunnels) (
    ix_error arg_Result,
    void * arg_pClientContext,
    ix_uint32 arg_NumEntries,
    ix_cc_v6v4_end_tunnel_info * arg_paInfo);
```

Input

<code>arg_Result</code>	<ul style="list-style-type: none"> <code>IX_SUCCESS</code>—returned if the operation succeeds. A valid <code>ix_error</code>—returned if the operation fails.
<code>arg_pClientContext</code>	Context pointer supplied by a calling application in the original message API call.
<code>arg_NumEntries</code>	Number of entries in array pointed to by <code>arg_paInfo</code> .
<code>arg_paInfo</code>	Pointer to array of end tunnel endpoint descriptors.

17.3.10 [ix_cc_v6v4_async_clear_allowed_sources\(\)](#)

Sends a message of type `IX_CC_V6V4_MSG_CLEAR_ALLOWED_SOURCES` to the tunneling core component.

C Syntax

```
ix_error ix_cc_v6v4_async_clear_allowed_sources(
    ix_cc_v6v4_tunnel_handle arg_hTunnel,
    ix_cc_v6v4_cb arg_Callback,
    void * arg_pClientContext);
```

Input

<code>arg_hTunnel</code>	A handle to end tunnel.
<code>arg_Callback</code>	A pointer to calling application-supplied callback function.
<code>arg_pClientContext</code>	A pointer to calling application-defined context, which is passed to the callback function.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. <code>IX_CC_V6V4_ERROR_MSG_LIBRARY</code>—operation failed, failure from message support library <code>IX_CC_ERROR_NULL</code>—operation failed, invalid pointer
--------------	---

17.3.11 `ix_cc_v6v4_async_add_start_tunnel()`

Sends a message of type `IX_CC_V6V4_MSG_ADD_START_TUNNEL` to the Tunneling core component.

C Syntax

```
ix_error ix_cc_v6v4_async_add_start_tunnel(
    ix_cc_v6v4_start_tunnel_config * arg_pConfig,
    ix_cc_v6v4_cb_add_tunnel arg_Callback,
    void * arg_pClientContext);
```

Input

<code>arg_pConfig</code>	Pointer to configuration information about the tunnel endpoint.
<code>arg_Callback</code>	Pointer to a calling application-supplied callback function. See ix_cc_v6v4_cb_add_tunnel .
<code>arg_pClientContext</code>	Pointer to a calling application-defined context passed to the callback function.

Output/Returns

Return Value

Returns a valid `ix_error`.

- `IX_SUCCESS`—the operation succeeded.
- `IX_CC_V6V4_ERROR_MSG_LIBRARY`—operation failed, failure from message support library
- `IX_CC_ERROR_NULL`—operation failed, invalid pointer

17.3.12 `ix_cc_v6v4_async_delete_start_tunnel()`

Sends a message of type `IX_CC_V6V4_MSG_DELETE_START_TUNNEL` to the Tunneling core component.

C Syntax

```
ix_error ix_cc_v6v4_async_delete_start_tunnel(
    ix_cc_v6v4_tunnel_handle arg_hTunnel,
    ix_cc_v6v4_cb arg_Callback,
    void * arg_pClientContext);
```

Input

<code>arg_hTunnel</code>	Handle to start tunnel.
<code>arg_Callback</code>	Pointer to a calling application-supplied callback. See ix_cc_v6v4_cb .
<code>arg_pClientContext</code>	Pointer to a calling application-defined context passed to the callback function.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. <code>IX_CC_V6V4_ERROR_MSG_LIBRARY</code>—operation failed, failure from message support library <code>IX_CC_ERROR_NULL</code>—operation failed, invalid pointer
--------------	---

17.3.13 `ix_cc_v6v4_async_set_ttl()`

Sends a message of type `IX_CC_V6V4_MSG_SET_TTL` to the Tunneling core component.

C Syntax

```
ix_error ix_cc_v6v4_async_set_ttl (
    ix_cc_v6v4_tunnel_handle arg_hTunnel,
    ix_uint8 arg_Ttl,
    ix_cc_v6v4_cb arg_Callback,
    void * arg_pClientContext);
```

Input

<code>arg_hTunnel</code>	Handle to start tunnel.
<code>arg_Ttl</code>	Time-to-live value to set for the tunnel endpoint.
<code>arg_Callback</code>	Pointer to a calling application-supplied callback function. See ix_cc_v6v4_cb .
<code>arg_pClientContext</code>	Pointer to a calling application-defined context passed to the callback function.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. <code>IX_CC_V6V4_ERROR_MSG_LIBRARY</code>—operation failed, failure from message support library <code>IX_CC_ERROR_NULL</code>—operation failed, invalid pointer
--------------	---

17.3.14 `ix_cc_v6v4_async_set_encap_tos_option()`

Sends a message of type `IX_CC_V6V4_MSG_SET_ENCAP_TOS_OPTION` to the Tunneling core component.

C Syntax

```
ix_error ix_cc_v6v4_async_set_encap_tos_option(
    ix_cc_v6v4_tunnel_handle arg_hTunnel,
    ix_uint8 arg_Selection,
    ix_cc_v6v4_cb arg_Callback,
    void * arg_pClientContext);
```

Input

<code>arg_hTunnel</code>	Handle to start tunnel.
--------------------------	-------------------------

Input

<code>arg_Selection</code>	Selects behavior of encapsulation process with respect to setting the TOS byte in the IPv4 header. This value can be one of the following <ul style="list-style-type: none"> <code>IX_CC_V6V4_SELECT</code>—to set the TOS byte to the value of the IPv6 traffic class field. <code>IX_CC_V6V4_CLEAR</code>—to set the TOS byte to the TOS value set for the tunnel.
<code>arg_Callback</code>	Pointer to a calling application-supplied callback function. See ix_cc_v6v4_cb .
<code>arg_pClientContext</code>	Pointer to a calling application-defined context passed to the callback function.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. <code>IX_CC_V6V4_ERROR_MSG_LIBRARY</code>—operation failed, failure from message support library <code>IX_CC_ERROR_NULL</code>—operation failed, invalid pointer
--------------	---

17.3.15 `ix_cc_v6v4_async_set_tos()`

Sends a message of type `IX_CC_V6V4_MSG_SET_TOS` to the Tunneling core component.

C Syntax

```
ix_error ix_cc_v6v4_async_set_tos (
    ix_cc_v6v4_tunnel_handle arg_hTunnel,
    ix_uint8 arg_Tos,
    ix_cc_v6v4_cb arg_Callback,
    void * arg_pClientContext);
```

Input

<code>arg_hTunnel</code>	Handle to start tunnel.
<code>arg_Tos</code>	TOS value to set for the tunnel endpoint.
<code>arg_Callback</code>	Pointer to a calling application-supplied callback function. See ix_cc_v6v4_cb .
<code>arg_pClientContext</code>	Pointer to a calling application-defined context passed to the callback function.

Output/Returns

Return Value

Returns a valid `ix_error`.

- `IX_SUCCESS`—the operation succeeded.
- `IX_CC_V6V4_ERROR_MSG_LIBRARY`—operation failed, failure from message support library
- `IX_CC_ERROR_NULL`—operation failed, invalid pointer

17.3.16 `ix_cc_v6v4_async_set_mtu()`

Sends a message of type `IX_CC_V6V4_MSG_SET_MTU` to the Tunneling core component. If `arg_Mtu` is set to the `IX_CC_V6V4_PATH_MTU_INTERFACE`, the core component sets the MTU for the tunnel to the MTU of the associated interface. In addition, any subsequent updates to the interface MTU is inherited by the tunnel, until the calling application sets the MTU to a value other than `IX_CC_V6V4_PATH_MTU_INTERFACE`. If `arg_Mtu` is not `IX_CC_V6V4_PATH_MTU_INTERFACE`, then the core component sets the tunnel MTU to `arg_Mtu` and does not change it until the calling application updates the MTU again.

C Syntax

```
ix_error ix_cc_v6v4_async_set_mtu(
    ix_cc_v6v4_tunnel_handle arg_hTunnel,
    ix_uint16 arg_Mtu,
    ix_cc_v6v4_cb arg_Callback,
    void * arg_pClientContext);
```

Input

<code>arg_hTunnel</code>	Handle to start tunnel.
<code>arg_Mtu</code>	Path MTU value to set for the tunnel endpoint.
<code>arg_Callback</code>	Pointer to a calling application-supplied callback function. See ix_cc_v6v4_cb .
<code>arg_pClientContext</code>	Pointer to a calling application-defined context passed to the callback function.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. <code>IX_CC_V6V4_ERROR_MSG_LIBRARY</code>—operation failed, failure from message support library <code>IX_CC_ERROR_NULL</code>—operation failed, invalid pointer
--------------	---

17.3.17 `ix_cc_v6v4_async_set_subnet_broadcast()`

Sends a message of type `IX_CC_V6V4_MSG_SET_SUBNET_BROADCAST` to the Tunneling core component. If `arg_subnetBroadcast` is set to the `IX_CC_V6V4_BROADCAST_INTERFACE`, the core component sets the subnet broadcast address for the tunnel to the broadcast address of the associated interface. In addition, any subsequent updates to the interface broadcast address is inherited by the tunnel, until the calling application sets the subnet broadcast address to a value other than `IX_CC_V6V4_BROADCAST_INTERFACE`. If `arg_subnetBroadcast` is not `IX_CC_V6V4_BROADCAST_INTERFACE`, then the core component sets the subnet broadcast address for the tunnel to `arg_subnetBroadcast` and does not change it until the calling application updates the subnet broadcast address again.

C Syntax

```
ix_error ix_cc_v6v4_async_set_subnet_broadcast(
    ix_cc_v6v4_tunnel_handle arg_hTunnel,
    ix_uint32 arg_subnetBroadcast,
    ix_cc_v6v4_cb arg_Callback,
    void * arg_pClientContext);
```

Input

<code>arg_hTunnel</code>	Handle to start tunnel.
<code>arg_subnetBroadcast</code>	Subnet broadcast address to set for the tunnel endpoint.
<code>arg_Callback</code>	Pointer to a calling application-supplied callback function. See ix_cc_v6v4_cb .
<code>arg_pClientContext</code>	Pointer to a calling application-defined context passed to the callback function.

Output/Returns

Return Value	<p>Returns a valid <code>ix_error</code>.</p> <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. <code>IX_CC_V6V4_ERROR_MSG_LIBRARY</code>—operation failed, failure from message support library <code>IX_CC_ERROR_NULL</code>—operation failed, invalid pointer
--------------	---

17.3.18 ix_cc_v6v4_async_get_start_tunnel()

Sends a message of type `IX_CC_V6V4_MSG_GET_START_TUNNEL` to the Tunneling core component.

C Syntax

```
ix_error ix_cc_v6v4_async_get_start_tunnel (
    ix_cc_v6v4_tunnel_handle arg_hTunnel,
    ix_cc_v6v4_cb_start_tunnel arg_Callback,
    void * arg_pClientContext);
```

Input

<code>arg_hTunnel</code>	Handle to start tunnel.
<code>arg_Callback</code>	Pointer to a calling application-supplied callback function. See ix_cc_v6v4_cb_start_tunnel .
<code>arg_pClientContext</code>	Pointer to a calling application-defined context passed to the callback function.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. <code>IX_CC_V6V4_ERROR_MSG_LIBRARY</code>—operation failed, failure from message support library <code>IX_CC_ERROR_NULL</code>—operation failed, invalid pointer
--------------	---

17.3.18.1 ix_cc_v6v4_cb_start_tunnel

The function prototype for message-handler callback function provided by the calling application.

C Syntax

```
ix_error (*ix_cc_v6v4_cb_start_tunnel) (
    ix_error arg_Result,
    void * arg_pClientContext,
    ix_cc_v6v4_start_tunnel_info * pTunnelInfo);
```

Input

<code>arg_Result</code>	<ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—returned if the operation succeeds. • A valid <code>ix_error</code>—returned if the operation fails.
<code>arg_pClientContext</code>	Context pointer supplied by calling application in the original message API call.
<code>pTunnelInfo</code>	Pointer to structure containing start tunnel configuration information.

17.3.19 `ix_cc_v6v4_async_dump_start_tunnels()`

Sends a message of type `IX_CC_V6V4_MSG_DUMP_START_TUNNELS` to the Tunneling core component.

C Syntax

```
ix_error ix_cc_v6v4_async_dump_start_tunnels(
    ix_cc_v6v4_cb_dump_start_tunnels arg_Callback,
    void * arg_pClientContext);
```

Input

<code>arg_Callback</code>	Pointer to a calling application-supplied callback function. See ix_cc_v6v4_cb_dump_start_tunnels .
<code>arg_pClientContext</code>	Pointer to a calling application-defined context passed to the callback function.

Output/Returns

Return Value	<p>Returns a valid <code>ix_error</code>.</p> <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • <code>IX_CC_V6V4_ERROR_MSG_LIBRARY</code>—operation failed, failure from message support library • <code>IX_CC_ERROR_NULL</code>—operation failed, invalid pointer
--------------	---

17.3.19.1 `ix_cc_v6v4_cb_dump_start_tunnels`

The function prototype for message-handler callback function provided by the calling application.

C Syntax

```
ix_error (*ix_cc_v6v4_cb_dump_start_tunnels) (
    ix_error arg_Result,
    void * arg_pClientContext,
    ix_uint32 arg_NumEntries,
    x_cc_v6v4_start_tunnel_info * arg_paInfo);
```

Input

<code>arg_Result</code>	<ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—returned if the operation succeeds. • A valid <code>ix_error</code>—returned if the operation fails.
<code>arg_pClientContext</code>	Context pointer supplied by calling application in the original message API call.
<code>arg_NumEntries</code>	Number of entries in array pointed to by <code>arg_paInfo</code> .
<code>arg_paInfo</code>	Pointer to array of start tunnel endpoint descriptors.

17.3.20 `ix_cc_v6v4_async_get_statistics()`

Sends a message of type `IX_CC_V6V4_MSG_GET_STATISTICS` to the Tunneling core component.

C Syntax

```
ix_error ix_cc_v6v4_async_get_statistics (
    ix_cc_v6v4_cb_statistics arg_Callback,
    void * arg_pClientContext);
```

Input

<code>arg_Callback</code>	Pointer to a calling application-supplied callback function. See ix_cc_v6v4_cb_statistics .
<code>arg_pClientContext</code>	Pointer to a calling application-defined context passed to the callback function.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • <code>IX_CC_V6V4_ERROR_MSG_LIBRARY</code>—operation failed, failure from message support library • <code>IX_CC_ERROR_NULL</code>—operation failed, invalid pointer
--------------	---

17.3.20.1 ix_cc_v6v4_cb_statistics

The function prototype for message-handler callback function provided by the calling application.

C Syntax

```
ix_error (*ix_cc_v6v4_cb_statistics) (
    ix_error arg_Result,
    void * arg_pClientContext,
    ix_cc_v6v4_statistics *arg_pStats);
```

Input

<code>arg_Result</code>	<ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—returned if the operation succeeds. • A valid <code>ix_error</code>—returned if the operation fails.
<code>arg_pClientContext</code>	Context pointer supplied by calling application in the original message API call.
<code>arg_pStats</code>	Pointer to tunneling microblock statistics.

17.4 Library API

Table 17-5 lists the IPv6 to IPv4 Tunnelling library API.

Table 17-5. IPv6 to IPv4 Tunnelling Library API

API	Description
<code>ix_cc_v6v4_add_end_tunnel()</code>	Adds an end tunnel endpoint
<code>ix_cc_v6v4_delete_end_tunnel()</code>	Deletes an end tunnel endpoint.
<code>ix_cc_v6v4_set_decap_tos_option()</code>	Specifies the manner in which the TOS byte is handled during decapsulation
<code>ix_cc_v6v4_set_src_validation()</code>	Specifies whether ingress source validation is performed during decapsulation
<code>ix_cc_v6v4_get_end_tunnel()</code>	Retrieves information about a specific end tunnel endpoint
<code>ix_cc_v6v4_add_allowed_source()</code>	Adds an entry to the ingress source validation list for an end tunnel endpoint

Table 17-5. IPv6 to IPv4 Tunnelling Library API (Continued)

API	Description
<code>ix_cc_v6v4_delete_allowed_source()</code>	Deletes an entry from the ingress source validation list for an end tunnel endpoint
<code>ix_cc_v6v4_clear_allowed_sources()</code>	Deletes all entries from the ingress source validation list
<code>ix_cc_v6v4_get_allowed_sources()</code>	Retrieves the list of allowed source addresses and associated prefix lengths associated with an end tunnel endpoint
<code>ix_cc_v6v4_dump_end_tunnels()</code>	Retrieves the list of all currently configured end tunnel endpoints
<code>ix_cc_v6v4_add_start_tunnel()</code>	Adds a start tunnel endpoint
<code>ix_cc_v6v4_delete_start_tunnel()</code>	Deletes a start tunnel endpoint
<code>ix_cc_v6v4_set_ttl()</code>	Sets the TTL value for a start tunnel endpoint
<code>ix_cc_v6v4_set_encap_tos_option()</code>	Specifies how the TOS byte is handled during the encapsulation process
<code>ix_cc_v6v4_set_tos()</code>	Sets the TOS value for the tunnel
<code>ix_cc_v6v4_set_mtu()</code>	Sets the path MTU for a start tunnel endpoint
<code>ix_cc_v6v4_set_subnet_broadcast()</code>	Sets the subnet broadcast address for a start tunnel endpoint
<code>ix_cc_v6v4_get_start_tunnel()</code>	Retrieves information about a specific start tunnel endpoint
<code>ix_cc_v6v4_dump_start_tunnels()</code>	Retrieves a list of all currently configured start tunnel endpoints
<code>ix_cc_v6v4_get_statistics()</code>	Retrieves tunneling packet counters
<code>ix_cc_v6v4_set_property()</code>	Updates the dynamic property

17.4.1 `ix_cc_v6v4_add_end_tunnel()`

Adds an end tunnel endpoint.

C Syntax

```
ix_error ix_cc_v6v4_add_end_tunnel (
    ix_cc_v6v4_end_tunnel_config * arg_pConfig,
    void * arg_pContext,
    ix_cc_v6v4_tunnel_handle * arg_phTunnel);
```

Input

<code>arg_pConfig</code>	Pointer to end tunnel configuration information.
<code>arg_pContext</code>	Pointer to a tunneling context.

Output/Returns

<code>arg_phTunnel</code>	Pointer to the location, which holds the returned core component handle to the tunnel. This handle must be passed to the subsequent library API calls to identify the tunnel.
Return Value	<p>Returns a valid <code>ix_error</code>.</p> <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. <code>IX_CC_ERROR_NULL</code>—operation failed, invalid pointer <code>IX_CC_ERROR_FULL</code>—operation failed, out of space in tunnel configuration table <code>IX_CC_V6V4_ERROR_INVALID_INPUT_PARAM</code>—operation failed, the information pointed to by <code>pConfig</code> is invalid.

17.4.2 `ix_cc_v6v4_delete_end_tunnel()`

Deletes an end tunnel endpoint.

C Syntax

```
ix_error ix_cc_v6v4_delete_end_tunnel (
    ix_cc_v6v4_tunnel_handle arg_hTunnel,
    void * arg_pContext);
```

Input

<code>arg_hTunnel</code>	Handle to an end tunnel.
<code>arg_pContext</code>	Pointer to a tunneling context.

Output/Returns

Return Value	<p>Returns a valid <code>ix_error</code>.</p> <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. <code>IX_CC_ERROR_NULL</code>—operation failed, nvalid pointer <code>IX_CC_ERROR_ENTRY_NOT_FOUND</code>—operation failed, tunnel not valid <code>IX_CC_V6V4_ERROR_INVALID_INPUT_PARAM</code>—operation failed, <code>arg_hTunnel</code> does not represent a valid end tunnel endpoint
--------------	--

17.4.3 `ix_cc_v6v4_set_decap_tos_option()`

Specifies the manner in which the TOS byte is handled during decapsulation. If selected, this option causes the tunnel decapsulation process in the microblocks and core component to copy the IPv4 TOS field into the IPv6 traffic class field. If not selected, the decapsulation process does not modify the IPv6 traffic class.

C Syntax

```
ix_error ix_cc_v6v4_set_decap_tos_option (
    ix_cc_v6v4_tunnel_handle arg_hTunnel,
    ix_uint8 arg_Selection,
    void * arg_pContext);
```

Input

arg_hTunnel	Handle to an end tunnel.
arg_Selection	Selection option: IX_CC_V6V4_SELECT—select the option IX_CC_V6V4_CLEAR—deselect the option.
arg_pContext	Pointer to a tunneling context.

Output/Returns

Return Value	Returns a valid ix_error. <ul style="list-style-type: none"> IX_SUCCESS—the operation succeeded. IX_CC_ERROR_NULL—operation failed, invalid pointer IX_CC_V6V4_ERROR_INVALID_INPUT_PARAM—operation failed, arg_hTunnel does not represent a valid end tunnel endpoint, or arg_Selection is not a valid value
--------------	--

17.4.4 ix_cc_v6v4_set_src_validation()

Specifies whether ingress source validation is performed during decapsulation. If selected, this option causes the tunnel decapsulation process in the microblocks and core component to validate the source address of packets received on configured or 6to4 tunnels against a list of valid prefixes. If not selected, the decapsulation process does not perform this validation.

C Syntax

```
ix_error ix_cc_v6v4_set_src_validation(
    ix_cc_v6v4_tunnel_handle arg_hTunnel,
    ix_uint8 arg_Selection,
    void * arg_pContext);
```

Input

arg_hTunnel	Handle to end tunnel.
arg_Selection	Selection option <ul style="list-style-type: none"> IX_CC_V6V4_SELECT—select ingress source validation, IX_CC_V6V4_CLEAR—deselect ingress source validation for the tunnel.
arg_pContext	Pointer to a tunneling context.

Output/Returns

Return Value

Returns a valid `ix_error`.

- `IX_SUCCESS`—the operation succeeded.
- `IX_CC_ERROR_NULL`—operation failed, invalid pointer
- `IX_CC_V6V4_ERROR_INVALID_INPUT_PARAM`—operation failed, `arg_hTunnel` does not represent a valid end tunnel endpoint, or `arg_Selection` is not a valid value

17.4.5 ix_cc_v6v4_get_end_tunnel()

Retrieves information about a specific end tunnel endpoint.

C Syntax

```
ix_error ix_cc_v6v4_get_end_tunnel(
    ix_cc_v6v4_end_tunnel_info * arg_pTunnelInfo,
    void * arg_pContext);
```

Input

arg_pContext Pointer to a tunneling context.

Input/Output

arg_pTunnelInfo Pointer to a buffer to contain tunnel information.

- Input—the calling application initializes arg_pTunnelInfo->hTunnel with the handle to the tunnel it is interested in.
- Output,—the core component supplies the remaining information in the structure.

Output/Returns

Return Value Returns a valid ix_error.

- IX_SUCCESS—the operation succeeded.
- IX_CC_ERROR_NULL—operation failed, invalid pointer
- IX_CC_V6V4_ERROR_INVALID_INPUT_PARAM—operation failed, arg_hTunnel does not represent a valid end tunnel endpoint

17.4.6 ix_cc_v6v4_add_allowed_source()

Adds an entry to the ingress source validation list for an end tunnel endpoint.

C Syntax

```
ix_error ix_cc_v6v4_add_allowed_source (
    ix_cc_v6v4_tunnel_handle arg_hTunnel,
    ix_cc_v6v4_ingress_source_entry * arg_pEntry,
    void * arg_pContext);
```

Input

<code>arg_hTunnel</code>	Handle to end tunnel.
<code>arg_pEntry</code>	Pointer to an ingress source entry to be added.
<code>arg_pContext</code>	Pointer to a tunneling context.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. <code>IX_CC_ERROR_NULL</code>—operation failed, nvalid pointer <code>IX_CC_V6V4_ERROR_INVALID_INPUT_PARAM</code>—operation failed, <code>arg_hTunnel</code> does not represent a valid end tunnel endpoint <code>IX_CC_V6V4_ERROR_INVALID_ADDRESS</code>—operation failed, <code>arg_pEntry</code> does not point to a valid source address and/or mask length <code>IX_CC_ERROR_DUPLICATE_ENTRY</code>—operation failed, entry already allowed for the tunnel <code>IX_CC_ERROR_FULL</code>—operation failed, entry cannot be added because ingress list is full
--------------	---

17.4.7 ix_cc_v6v4_delete_allowed_source()

Deletes an entry from the ingress source validation list for an end tunnel endpoint.

C Syntax

```
ix_error ix_cc_v6v4_delete_allowed_source (
    ix_cc_v6v4_tunnel_handle arg_hTunnel,
    ix_cc_v6v4_ingress_source_entry * arg_pEntry,
    void * arg_pContext);
```

Input

<code>arg_hTunnel</code>	Handle to end tunnel.
<code>arg_pEntry</code>	Pointer to an ingress source entry to be deleted.
<code>arg_pContext</code>	Pointer to a tunneling context.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. <code>IX_CC_ERROR_NULL</code>—operation failed, invalid pointer <code>IX_CC_V6V4_ERROR_INVALID_INPUT_PARAM</code>—operation failed, <code>arg_hTunnel</code> does not represent a valid end tunnel endpoint <code>IX_CC_V6V4_ERROR_INVALID_ADDRESS</code>—operation failed, <code>arg_pEntry</code> does not point to a valid source address and/or mask length <code>IX_CC_ERROR_ENTRY_NOT_FOUND</code>—operation failed, <code>arg_pEntry</code> does not point to a source address and mask length currently allowed by the tunnel
--------------	--

17.4.8 `ix_cc_v6v4_clear_allowed_sources()`

Deletes all entries from the ingress source validation list for an end tunnel endpoint.

C Syntax

```
ix_error ix_cc_v6v4_delete_allowed_source (
    ix_cc_v6v4_tunnel_handle arg_hTunnel,
    void * arg_pContext);
```

Input

<code>arg_hTunnel</code>	A handle to end tunnel.
<code>arg_pContext</code>	A pointer to tunneling context.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. <code>IX_CC_ERROR_NULL</code>—operation failed, invalid pointer <code>IX_CC_V6V4_ERROR_INVALID_INPUT_PARAM</code>—operation failed, <code>arg_hTunnel</code> does not represent a valid end tunnel endpoint
--------------	--

17.4.9 ix_cc_v6v4_get_allowed_sources()

Retrieves the list of allowed source addresses and associated prefix lengths associated with an end tunnel endpoint.

If the error `IX_CC_V6V4_ERROR_BUFFER_TOO_SMALL` is returned, the calling application can allocate a larger buffer and call this function again.

C Syntax

```
ix_error ix_cc_v6v4_get_allowed_sources(
    ix_cc_v6v4_tunnel_handle arg_hTunnel,
    ix_uint32 * arg_pNumEntries,
    ix_cc_v6v4_ingress_source_entry * arg_paEntries,
    void * arg_pContext);
```

Input

<code>arg_hTunnel</code>	Handle to end tunnel.
<code>arg_pContext</code>	Pointer to a tunneling context.

Input/Output

<code>arg_pNumEntries</code>	Both an input and an output parameter: <ul style="list-style-type: none"> Input—this points to the number of entries that the buffer pointed to by <code>arg_paEntries</code> can accommodate. Output—this points to the actual number of entries written to the buffer. If <code>IX_CC_V6V4_ERROR_BUFFER_TOO_SMALL</code> is returned, then this number will be the total number of entries that would have been written to the buffer if it had been large enough.
------------------------------	--

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. <code>IX_CC_ERROR_NULL</code>—operation failed, invalid pointer <code>IX_CC_V6V4_ERROR_INVALID_INPUT_PARAM</code>—operation failed, <code>arg_hTunnel</code> does not represent a valid end tunnel endpoint <code>IX_CC_V6V4_ERROR_BUFFER_TOO_SMALL</code>—operation failed, more entries exist than could fit into the supplied buffer (non-fatal error)
--------------	---

17.4.10 ix_cc_v6v4_dump_end_tunnels()

Retrieves the list of all currently configured end tunnel endpoints. If the error `IX_CC_V6V4_ERROR_BUFFER_TOO_SMALL` is returned, the calling application can allocate a larger buffer and call this function again.

C Syntax

```
ix_error ix_cc_v6v4_dump_end_tunnels (
    ix_uint32 arg_pNumEntries,
    ix_cc_v6v4_end_tunnel_info * arg_paInfo,
    void * arg_pContext);
```

Input

`arg_pContext` Pointer to a tunneling context.

Input/Output

`arg_pNumEntries` Both an input and an output parameter:

- Input—this points to the number of entries that the array pointed to by `arg_paInfo` can accommodate.
- Output—this points to the actual number of entries written to the buffer. If `IX_CC_V6V4_ERROR_BUFFER_TOO_SMALL` is returned, then this number will be the total number of entries that would have been written to the buffer if it had been large enough.

Output/Returns

`arg_paInfo` Pointer to an array to return the information.

Return Value Returns a valid `ix_error`.

- `IX_SUCCESS`—the operation succeeded.
- `IX_CC_ERROR_NULL`—operation failed, invalid pointer
- `IX_CC_V6V4_ERROR_BUFFER_TOO_SMALL`—operation failed, more entries exist than could fit into the supplied buffer (non-fatal error)

17.4.11 ix_cc_v6v4_add_start_tunnel()

Adds a start tunnel endpoint.

C Syntax

```
ix_error ix_cc_v6v4_add_start_tunnel(
    ix_cc_v6v4_start_tunnel_config * arg_pConfig,
    void * arg_pContext,
    ix_cc_v6v4_tunnel_handle * arg_phTunnel);
```

Input

arg_pConfig	Pointer to the configuration information about tunnel endpoint.
arg_pContext	Pointer to a tunneling context.

Output/Returns

arg_phTunnel	Pointer to the location, which holds the returned core component handle of the tunnel. This handle must be passed to subsequent library API calls to identify the tunnel.
Return Value	<p>Returns a valid ix_error.</p> <ul style="list-style-type: none"> IX_SUCCESS—the operation succeeded. IX_CC_ERROR_FULL—operation failed, start tunnel table is full IX_CC_ERROR_NULL—operation failed, invalid pointer IX_CC_V6V4_ERROR_INVALID_INPUT_PARAM—operation failed, the information pointed to by arg_pConfig is invalid. IX_CC_ERROR_OOM—operation failed, out of memory IX_CC_ERROR_ENTRY_NOT_FOUND—operation failed, the local interface specified was not found

17.4.12 ix_cc_v6v4_delete_start_tunnel()

Deletes a start tunnel endpoint.

C Syntax

```
ix_error ix_cc_v6v4_delete_start_tunnel(
    ix_cc_v6v4_tunnel_handle arg_hTunnel,
    void * arg_pContext);
```

Input

arg_hTunnel	Handle to start tunnel.
arg_pContext	Pointer to a tunneling context.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • <code>IX_CC_ERROR_NULL</code>—operation failed, invalid pointer • <code>IX_CC_V6V4_ERROR_INVALID_INPUT_PARAM</code>—operation failed, <code>arg_hTunnel</code> does not represent a valid start tunnel endpoint
--------------	--

17.4.13 `ix_cc_v6v4_set_ttl()`

Sets the TTL value for a start tunnel endpoint.

C Syntax

```
ix_error ix_cc_v6v4_set_ttl(
    ix_cc_v6v4_tunnel_handle arg_hTunnel,
    ix_uint8 arg_Ttl,
    void * arg_pContext);
```

Input

<code>arg_hTunnel</code>	Handle to start tunnel.
<code>arg_Ttl</code>	Time-to-live value to set for the tunnel endpoint.
<code>arg_pContext</code>	Pointer to a tunneling context.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • <code>IX_CC_ERROR_NULL</code>—operation failed, invalid pointer • <code>IX_CC_V6V4_ERROR_INVALID_INPUT_PARAM</code>—operation failed, <code>arg_hTunnel</code> does not represent a valid start tunnel endpoint
--------------	--

17.4.14 `ix_cc_v6v4_set_encap_tos_option()`

Specifies how the TOS byte is handled during the encapsulation process.

C Syntax

```
ix_error ix_cc_v6v4_set_encap_tos_option(
    ix_cc_v6v4_tunnel_handle arg_hTunnel,
    ix_uint16 arg_Selection,
```

```
void * arg_pContext);
```

Input

<code>arg_hTunnel</code>	Handle to start tunnel.
<code>arg_Selection</code>	Selects behavior of encapsulation process with respect to setting the TOS byte in the IPv4 header.
<code>arg_pContext</code>	Pointer to a tunneling context.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. <code>IX_CC_ERROR_NULL</code>—operation failed, invalid pointer <code>IX_CC_V6V4_ERROR_INVALID_INPUT_PARAM</code>—operation failed, <code>arg_hTunnel</code> does not represent a valid start tunnel endpoint, or <code>arg_Selection</code> is not a valid value
--------------	--

17.4.15 `ix_cc_v6v4_set_tos()`

Sets the TOS value for the tunnel. If the tunnel is configured to encode the TOS byte from the IPv6 traffic class, then this value is recorded in the tunnel but not used during encapsulation. Otherwise, this value is recorded in the tunnel entry and encoded into the TOS byte during encapsulation.

C Syntax

```
ix_error ix_cc_v6v4_set_tos (
    ix_cc_v6v4_tunnel_handle arg_hTunnel,
    ix_uint8 arg_Tos,
    void * arg_pContext);
```

Input

<code>arg_hTunnel</code>	Handle to start tunnel.
<code>arg_Tos</code>	TOS value to set for the tunnel endpoint.
<code>arg_pContext</code>	Pointer to a tunneling context.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none">• <code>IX_SUCCESS</code>—the operation succeeded.• <code>IX_CC_ERROR_NULL</code>—operation failed, invalid pointer• <code>IX_CC_V6V4_ERROR_INVALID_INPUT_PARAM</code>—operation failed, <code>arg_hTunnel</code> does not represent a valid start tunnel endpoint.
--------------	---

17.4.16 ix_cc_v6v4_set_mtu()

Sets the path MTU for a start tunnel endpoint. If `arg_Mtu` is set to the special value `IX_CC_V6V4_PATH_MTU_INTERFACE`, the core component sets the MTU for the tunnel to the MTU of the associated interface. Furthermore, any subsequent updates to the interface MTU is inherited by the tunnel, until the calling application sets the MTU to a value other than `IX_CC_V6V4_PATH_MTU_INTERFACE`. If `arg_Mtu` is not `IX_CC_V6V4_PATH_MTU_INTERFACE`, then the core component simply sets the tunnel MTU to `arg_Mtu` and does not change it until the calling application updates the MTU again.

C Syntax

```
ix_error ix_cc_v6v4_set_mtu(
    ix_cc_v6v4_tunnel_handle arg_hTunnel,
    ix_uint16 arg_Mtu,
    void * arg_pContext);
```

Input

<code>arg_hTunnel</code>	Handle to start tunnel.
<code>arg_Mtu</code>	Path MTU value to set for the tunnel endpoint.
<code>arg_pContext</code>	Pointer to a tunneling context.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. <code>IX_CC_ERROR_NULL</code>—operation failed, invalid pointer <code>IX_CC_V6V4_ERROR_INVALID_INPUT_PARAM</code>—operation failed, <code>arg_hTunnel</code> does not represent a valid start tunnel endpoint
--------------	--

17.4.17 ix_cc_v6v4_set_subnet_broadcast()

Sets the subnet broadcast address for a start tunnel endpoint. If `arg_subnetBroadcast` is set to the special value `IX_CC_V6V4_BROADCAST_INTERFACE`, the core component sets the subnet broadcast address for the tunnel to the broadcast address of the associated interface.

In addition, any subsequent updates to the interface broadcast address will be inherited by the tunnel, until the calling application sets the subnet broadcast address to a value other than `IX_CC_V6V4_BROADCAST_INTERFACE`. If `arg_subnetBroadcast` is not `IX_CC_V6V4_BROADCAST_INTERFACE`, then the core component simply sets the subnet broadcast address for the tunnel to `arg_subnetBroadcast` and does not change it until the calling application updates the subnet broadcast address again.

C Syntax

```
ix_error ix_cc_v6v4_set_subnet_broadcast(
    ix_cc_v6v4_tunnel_handle arg_hTunnel,
    ix_uint32 arg_subnetBroadcast,
    void * arg_pContext);
```

Input

<code>arg_hTunnel</code>	Handle to start tunnel.
<code>arg_subnetBroadcast</code>	Subnet broadcast address to set for the tunnel endpoint.
<code>arg_pContext</code>	Pointer to a tunneling context.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. <code>IX_CC_ERROR_NULL</code>—operation failed, invalid pointer <code>IX_CC_V6V4_ERROR_INVALID_INPUT_PARAM</code>—operation failed, <code>arg_hTunnel</code> does not represent a valid start tunnel endpoint
--------------	--

17.4.18 ix_cc_v6v4_get_start_tunnel()

Retrieves information about a specific start tunnel endpoint.

C Syntax

```
ix_error ix_cc_v6v4_get_start_tunnel (
    ix_cc_v6v4_start_tunnel_info * arg_pTunnelInfo,
    void * arg_pContext);
```

Input

`arg_pContext` Pointer to a tunneling context.

Input/Output

`arg_pTunnelInfo` Pointer to buffer to contain tunnel information.

- Input—the calling application initializes `arg_pTunnelInfo->hTunnel` with the handle to the tunnel it is interested in.
- Output—the core component supplies the remaining information in the structure.

Output/Returns

Return Value Returns a valid `ix_error`.

- `IX_SUCCESS`—the operation succeeded.
- `IX_CC_ERROR_NULL`—operation failed, invalid pointer
- `IX_CC_V6V4_ERROR_INVALID_INPUT_PARAM`—operation failed, `arg_hTunnel` does not represent a valid start tunnel endpoint

17.4.19 ix_cc_v6v4_dump_start_tunnels()

Retrieves a list of all currently configured start tunnel endpoints.

If the error `IX_CC_V6V4_ERROR_BUFFER_TOO_SMALL` is returned, the calling application can allocate a larger buffer and call this function again.

C Syntax

```
ix_error ix_cc_v6v4_dump_start_tunnels(
    ix_uint32 * arg_pNumEntries,
    ix_cc_v6v4_start_tunnel_info * arg_paInfo,
    void * arg_pContext);
```

Input

`arg_pContext` Pointer to a tunneling context.

Input/Output

`arg_pNumEntries` Both an input and an output parameter:

- Input—this points to the number of entries that the buffer pointed to by `arg_paInfo` can accommodate.
- Output—this points to the actual number of entries written to the buffer. If `IX_CC_V6V4_ERROR_BUFFER_TOO_SMALL` is returned, then this number will be the total number of entries that would have been written to the buffer if it had been large enough.

Output/Returns

`arg_paInfo` Pointer to the buffer to return the information.

Return Value Returns a valid `ix_error`.

- `IX_SUCCESS`—the operation succeeded.
- `IX_CC_ERROR_NULL`—operation failed, invalid pointer
- `IX_CC_V6V4_ERROR_BUFFER_TOO_SMALL`—operation failed, more entries exist than could fit into the supplied buffer (non-fatal error)

17.4.20 `ix_cc_v6v4_get_statistics()`

Retrieves tunneling packet counters.

C Syntax

```
ix_error ix_cc_v6v4_get_statistics (
    ix_cc_v6v4_statistics * arg_pStats,
    void * arg_pContext);
```

Input

`arg_pStats` Pointer to tunneling microblock statistics.

`arg_pContext` Pointer to a tunneling context.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> .
	<ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • <code>IX_CC_ERROR_NULL</code>—operation failed, invalid pointer

17.4.21 `ix_cc_v6v4_set_property()`

This function implements the library API for dynamic property updates.

C Syntax

```
ix_error ix_cc_v6v4_set_property(
    ix_uint32 arg_propId,
    ix_cc_properties * arg_pProperty,
    void * arg_pContext);
```

Input

<code>arg_propId</code>	Identifies property to be changed.
<code>arg_pProperty</code>	Pointer to buffer containing property information.
<code>arg_pContext</code>	Pointer to a tunneling context.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> .
	<ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • <code>IX_CC_ERROR_NULL</code>—operation failed, invalid pointer

The translation core component assists the translation microblock to implement the IPv6 to IPv4 (and vice-versa) address and protocol translation as defined in RFC2766 Network Address Translation-Port Translation specification (NAT-PT).

A single core component supports the translation microblock. The following functionality is provided:

- Configuration of the translation microblock.
- Setup and management of translation information.
- Packet handlers to receive packets from the translation microblock and from other core components.
- Message handlers to allow configuration of translation information.
- Translation of ICMP and ICMPv6 packets.
- Translation of FTP Control packets (FTP ALG).
- Translation of DNS packets (DNS ALG).
- Fragmentation and Reassembly of translated packets.

The translation core component can receive packets from the translation microblock only. Messages can be received from other core components and from other system components responsible for configuring the forwarding plane.

For complete details see [Chapter 55, “NAT-PT Translation Core Components”](#) of the *Intel® Internet Exchange Architecture Software Building Blocks Developer's Manual*.

18.1 Data Structures, Types and Macros

Table lists the data structures, types and macros in translation core components

Table 18-1. Data Structures, Types and Macros in Translation Core Components

Data Structures, Types and Macros	Description
ix_cc_natpt_config_params	Contains the configuration parameters
ix_cc_natpt_v6addr	Specifies an IPv6 address
ix_cc_natpt_v4addr	Specifies an IPv4 address
ix_cc_natpt_static_v6v4map	Specifies a static IPv6 address to IPv4 address mapping
ix_cc_natpt_v4v6portmap	Specifies a static mapping of IPv6 address, IPv4 address and TCP/UDP Port number
ix_cc_natpt_naptmode	Turns ON/OFF NAPT-PT mode of operation
ix_cc_natpt_session	Contains information on a NAT-PT session

18.1.1 ix_cc_natpt_config_params

This structure contains the configuration parameters for the translation core component.

C Syntax

```
typedef struct ix_s_cc_natpt_config_params {  
    ix_uint32 idleCheckInterval;  
} ix_cc_natpt_config_params;
```

Input

idleCheckInterval	This value is given in seconds and is used by the core component to check at regular intervals check for any packet activity for a NAT-PT session. After detection of no activity, the session is removed. The default value is 300. The minimum value is 60.
-------------------	---

18.1.2 ix_cc_natpt_v6addr

This data type specifies an IPv6 address.

C Syntax

```
typedef ix_uint128 ix_cc_natpt_v6addr;
```

18.1.3 ix_cc_natpt_v4addr

This data type specifies an IPv4 address.

C Syntax

```
typedef ix_uint32 ix_cc_natpt_v4addr;
```

18.1.4 ix_cc_natpt_static_v6v4map

This structure specifies a static IPv6 address to IPv4 address mapping.

C Syntax

```
typedef struct ix_s_cc_natpt_static_v6v4map {  
    ix_cc_natpt_v6addr v6addr;  
    ix_cc_natpt_v4addr v4addr;  
} ix_cc_natpt_static_v6v4map;  
v6addrIPv6 address.  
v4addrIPv4 address.
```


18.1.5 ix_cc_natpt_v4v6portmap

This structure specifies a static mapping of IPv6 address, IPv4 address and TCP/UDP Port number. However, IPv4 address is not given since it is the NAT-PT IPv4 address already selected when NAT mode was turned ON.

C Syntax

```
typedef struct ix_s_cc_natpt_v4v6portmap {
    ix_cc_natpt_v6addr v6addr;
    ix_uint16 protocol;
    ix_uint16 portNumber;
} ix_cc_natpt_v4v6portmap;
v6addrIPv6 address.
```

Input

protocol	Protocol number—TCP/UDP.
portNumber	TCP/UDP port number.

18.1.6 ix_cc_natpt_naptmode

This data type turns ON/OFF NAT-PT mode of operation.

C Syntax

```
typedef enum ix_e_cc_natpt_naptmode {
    IX_CC_NATPT_NAPT_OFF,
    IX_CC_NATPT_NAPT_ON
} ix_cc_natpt_naptmode;
```

18.1.7 ix_cc_natpt_session

This structure contains information on a NAT-PT session.

C Syntax

```
typedef struct ix_s_cc_natpt_session {
    ix_cc_natpt_v6addr v6addr;
    ix_cc_natpt_v4addr v4addr;
    ix_uint16 protocol;
    ix_uint16 v6PortNumber;
    ix_uint16 v4PortNumber;
} ix_cc_natpt_session;
```

Input

v6addr	IPv6 address.
v4addr	IPv4 address.
protocol	TCP/UDP.
v6PortNumber	TCP/UDP IPv6 port number.
v4PortNumber	TCP/UDP IPv4 port number.

18.1.8 Translation CC Specific Error Codes

Table 18-2 lists the Translation-specific error codes.

Table 18-2. Translation-specific Error Codes

IX_CC_NATPT_ERROR_SUCCESS
IX_CC_NATPT_ERROR_FAILED_PATCHING
IX_CC_NATPT_ERROR_REGISTRY
IX_CC_NATPT_ERROR_BUFFER_FREE
IX_CC_NATPT_ERROR_CCI
IX_CC_NATPT_ERROR_INVALID_CONTEXT
IX_CC_NATPT_ERROR_INVALID_INPUT_PARAM
IX_CC_NATPT_ERROR_INVALID_BUFFER
IX_CC_NATPT_ERROR_MSG_LIBRARY
IX_CC_NATPT_ERROR_INVALID_PACKET
IX_CC_NATPT_ERROR_INVALID_PROP_ID
IX_CC_NATPT_ERROR_FAILED_GET_PORT_STATUS
IX_CC_NATPT_ERROR_CALLBACK
IX_CC_NATPT_ERROR_NOT_INITIALIZED
IX_CC_NATPT_ERROR_IGNORE_PACKET
IX_CC_NATPT_ERROR_BUFFER_TOO_SMALL
IX_CC_NATPT_ERROR_NOT_IMPLEMENTED
IX_CC_NATPT_ERROR_ICMP_TRANSLATE
IX_CC_NATPT_ERROR_UNKNOWN

18.2 Core Component Infrastructure API

Table 18-3 lists the translation Core Component Infrastructure API.

Table 18-3. Translation Core Component Infrastructure API

API	Description
<code>ix_cc_natpt_init()</code>	Initializes the translation core component
<code>ix_cc_natpt_fini()</code>	Terminate the services of the translation core component
<code>ix_cc_natpt_msg_handler()</code>	Message handler routine for the translation core component
<code>ix_cc_natpt_microblock_pkt_handler()</code>	Packet handler routine for packets received from the translation microblock.

18.2.1 `ix_cc_natpt_init()`

This function initializes the translation core component and performs all initialization required for IPv6/IPv4 translation, including patching symbols for the microblocks, allocating and initializing all required memory.

C Syntax

```
ix_error ix_cc_natpt_init(
    ix_cc_handle arg_hCcHandle,
    void ** arg_ppContext);
```

Input

<code>arg_hCcHandle</code>	Handle to IPv6/IPv4 translation core component, created by the core component infrastructure. This value is used subsequently to request services from the core component infrastructure.
----------------------------	---

Output/Returns

<code>arg_ppContext</code>	Location where the pointer to a buffer of context information allocated by the translation core component is stored. This buffer contains information used internally by the translation core component. When the translation core component is to be terminated, this pointer is passed to <code>ix_cc_natpt_fini()</code> to release resources.
----------------------------	---

Output/Returns (Continued)

Return Value	<p>Returns a valid <code>ix_error</code>.</p> <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • <code>IX_CC_ERROR_NULL</code>—operation failed, input parameter is null • <code>IX_CC_ERROR_OOM</code>—operation failed, memory allocation failure • <code>IX_CC_ERROR_OOR</code>—operation failed, resource allocation failure • <code>IX_CC_NATPT_ERROR_FAILED_PATCHING</code>—operation failed, patching failure • <code>IX_CC_NATPT_ERROR_REGISTRY</code>—operation failed, information from registry is invalid • <code>IX_CC_NATPT_ERROR_CCI</code>—operation failed, failure from CCI
--------------	---

18.2.2 `ix_cc_natpt_fini()`

This primitive is called by the core component infrastructure to terminate the services of the translation core component. This routine releases all resources allocated for IPv6/IPv4 translation. It is assumed that the microengines have been shut down before this routine is called.

C Syntax

```
ix_error ix_cc_natpt_fini(
    ix_cc_handle arg_hCcHandle
    void * arg_pContext);
```

Input

<code>arg_hCcHandle</code>	Handle to IPv6/IPv4 translation core component.
<code>arg_pContext</code>	Pointer to the core component context information allocated in <code>ix_cc_natpt_init()</code> .

Output/Returns

Return Value	<p>Returns a valid <code>ix_error</code>.</p> <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • <code>IX_CC_ERROR_NULL</code>—operation failed, invalid input pointer • <code>IX_CC_NATPT_ERROR_INVALID_INPUT_PARAM</code>—operation failed, invalid data in <code>arg_pContext</code> • <code>IX_CC_NATPT_ERROR_FAILED_MEMORY_FREE</code>—operation failed, memory free failure • <code>IX_CC_NATPT_ERROR_CCI</code>—operation failed, failure from CCI
--------------	---

18.2.3 ix_cc_natpt_msg_handler()

This function is the message handler for the translation core component. The translation core component receives messages from the calling applications through this function.

C Syntax

```
ix_error ix_cc_natpt_msg_handler(
    ix_buffer_handle arg_hMsgInfo,
    ix_uint32 arg_MsgType,
    void * arg_pContext);
```

Input

arg_hMsgInfo	Handle to buffer containing information pertinent to the message.
arg_MsgType	Message type— Table 18-4 lists the message types supported.
arg_pContext	Pointer to the core component context information, which was allocated in ix_cc_natpt_init() .

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> IX_SUCCESS—the operation succeeded. IX_CC_ERROR_NULL—operation failed, invalid pointer IX_CC_ERROR_OOM—operation failed, memory allocation failure IX_CC_ERROR_UNDEFINED_MSG—operation failed, invalid message type IX_CC_NATPT_ERROR_INVALID_BUFFER—operation failed, buffer handle is unusable IX_CC_NATPT_ERROR_INVALID_DATA—operation failed, message information is invalid IX_CC_NATPT_ERROR_BUFFER_FREE—operation failed, buffer free failure
--------------	--

[Table 18-4](#) lists the translation core component supported message types.

Table 18-4. Message Types for the Translation Core Component

Messages	Description
IX_CC_NATPT_MSG_SET_CONFIG_PARAM	Sets configuration parameters.
IX_CC_NATPT_MSG_GET_CONFIG_PARAM	Retrieves current configuration parameters.
IX_CC_NATPT_MSG_ADD_STATIC_MAP	Adds a static v6 to v4 address mapping.
IX_CC_NATPT_MSG_REM_STATIC_MAP	Removes one or all static v6 to v4 address mappings.
IX_CC_NATPT_MSG_GET_STATIC_MAP	Retrieves current list of static v6 to v4 address mappings.
IX_CC_NATPT_MSG_ADD_V4ADDR_POOL	Adds one or more v4 addresses to the pool of v4 addresses that are used for NAT or NAPT.

Table 18-4. Message Types for the Translation Core Component (Continued)

Messages	Description
IX_CC_NATPT_MSG_REM_V4ADDR_POOL	Removes one or more v4 addresses from the pool of v4 addresses.
IX_CC_NATPT_MSG_GET_V4ADDR_POOL	Retrieves current list of v4 addresses in the pool of v4 addresses.
IX_CC_NATPT_MSG_SET_NAPT_MODE	Turns NAPT mode ON or OFF.
IX_CC_NATPT_MSG_ADD_V4V6_PORT_MAP	Adds a static v4 to v6 address mapping for a specific TCP/UDP port number.
IX_CC_NATPT_MSG_REM_V4V6_PORT_MAP	Removes one or all static v4 to v6 address mappings with port number.
IX_CC_NATPT_MSG_GET_V4V6_PORT_MAP	Retrieves current list of static v4 to v6 address mappings with port number.
IX_CC_NATPT_MSG_GET_ACTIVE_SESSIONS	Retrieves current list of active sessions.

18.2.4 ix_cc_natpt_microblock_pkt_handler()

This function is the packet handler routine for packets received from the translation microblock.

C Syntax

```
ix_error ix_cc_natpt_microblock_pkt_handler(
    ix_buffer_handle arg_hPacket,
    ix_uint32 arg_ExceptionCode,
    void * arg_pContext);
```

Input

arg_hPacket	Handle to the buffer containing exception packet.
arg_ExceptionCode	Exception code set by microblock.
arg_pContext	Pointer to the core component context information allocated in ix_cc_natpt_init() .

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. <code>IX_CC_ERROR_NULL</code>—operation failed, invalid pointer <code>IX_CC_ERROR_INTERNAL</code>—operation failed, unknown internal error <code>IX_CC_ERROR_OOR</code>—operation failed, resource allocation failure <code>IX_CC_NATPT_ERROR_ICMP_TRANSLATE</code>—operation failed, error while translating ICMP header <code>IX_CC_ERROR_UNIMPL</code>—operation failed, not implemented <code>IX_CC_ERROR_UNDEFINED_EXCEP</code>—operation failed, invalid exception code <code>IX_CC_NATPT_ERROR_INVALID_BUFFER</code>—operation failed, buffer handle is unusable
--------------	---

18.3 Message Helper API

The message helper API is used by the calling application to construct and send messages asynchronously to the translation core component. [Table 18-5](#) lists the message helper API in the translation core component.

Table 18-5. Message Helper API in the Translation Core Component

Message Helper API	Description
<code>ix_cc_natpt_async_set_config_parameters()</code>	Sends a message of type <code>IX_CC_NATPT_SET_CONFIG_PARAM</code> to the translation core component.
<code>ix_cc_natpt_async_get_config_parameters()</code>	Sends a message of type <code>IX_CC_NATPT_GET_CONFIG_PARAM</code> to the translation core component.
<code>ix_cc_natpt_async_add_static_mapping()</code>	Sends a message of type <code>IX_CC_NATPT_ADD_STATIC_MAP</code> to the translation core component.
<code>ix_cc_natpt_async_remove_static_mapping()</code>	Sends a message of type <code>IX_CC_NATPT_REM_STATIC_MAP</code> to the translation core component.
<code>ix_cc_natpt_async_get_static_mapping()</code>	Sends a message of type <code>IX_CC_NATPT_GET_STATIC_MAP</code> to the translation core component.
<code>ix_cc_natpt_async_add_v4addr_pool()</code>	Sends a message of type <code>IX_CC_NATPT_ADD_V4ADDR_POOL</code> to the translation core component.
<code>ix_cc_natpt_async_remove_v4addr_pool()</code>	Sends a message of type <code>IX_CC_NATPT_REM_V4ADDR_POOL</code> to the translation core component.
<code>ix_cc_natpt_async_get_v4addr_pool()</code>	Sends a message of type <code>IX_CC_NATPT_GET_V4ADDR_POOL</code> to the translation core component.

Table 18-5. Message Helper API in the Translation Core Component

Message Helper API	Description
<code>ix_cc_natpt_async_set_natpt_mode()</code>	Sends a message of type <code>IX_CC_NATPT_SET_NAPT_MODE</code> to the translation core component.
<code>ix_cc_natpt_async_add_v4v6port_mapping()</code>	Sends a message of type <code>IX_CC_NATPT_ADD_V4V6_PORT_MAP</code> to the translation core component.
<code>ix_cc_natpt_async_remove_v4v6port_mapping()</code>	Sends a message of type <code>IX_CC_NATPT_REM_V4V6_PORT_MAP</code> to the translation core component.
<code>ix_cc_natpt_async_get_v4v6port_mapping()</code>	Sends a message of type <code>IX_CC_NATPT_GET_V6V4_PORT_MAP</code> to the translation core component.
<code>ix_cc_natpt_async_get_active_sessions()</code>	Sends a message of type <code>IX_CC_NATPT_GET_ACTIVE_SESSIONS</code> to the translation core component.

18.3.1 `ix_cc_natpt_async_set_config_parameters()`

Sends a message of type `IX_CC_NATPT_SET_CONFIG_PARAM` to the translation core component.

C Syntax

```
ix_error ix_cc_natpt_async_set_config_parameters(
    ix_cc_natpt_config_params * arg_pConfig,
    ix_cc_natpt_cb arg_Callback,
    void * arg_pClientContext);
```

Input

<code>arg_pConfig</code>	Pointer to configuration parameters structure.
<code>arg_Callback</code>	Pointer to the calling application-supplied callback function. See ix_cc_natpt_cb()
<code>arg_pClientContext</code>	Pointer to the calling application-defined context passed to the callback function.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. <code>IX_CC_NATPT_ERROR_MSG_LIBRARY</code>—operation failed, failure from message support library <code>IX_CC_ERROR_NULL</code>—operation failed, invalid pointer
--------------	--

18.3.1.1 **ix_cc_natpt_cb()**

This is the format for the callback functions used by the message helper API functions when no message-specific data needs to be returned to the calling application.

C Syntax

```
void ix_cc_natpt_cb(
    ix_error arg_Result,
    void * arg_pClientContext);
```

Input

<code>arg_pClientContext</code>	Context pointer supplied by the calling application in the original message API call.
---------------------------------	---

Output/Returns

<code>arg_Result</code>	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded.
-------------------------	--

18.3.2 **ix_cc_natpt_async_get_config_parameters()**

Sends a message of type `IX_CC_NATPT_GET_CONFIG_PARAM` to the translation core component.

C Syntax

```
ix_error ix_cc_natpt_async_get_config_parameters(
    ix_cc_natpt_cb_get_config_parameters arg_Callback,
    void * arg_pClientContext);
```

Input

<code>arg_Callback</code>	Pointer to the calling application-supplied callback function. See ix_cc_natpt_cb_get_config_parameters .
<code>arg_pClientContext</code>	Pointer to the calling application-defined context passed to the callback function.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • <code>IX_CC_NATPT_ERROR_MSG_LIBRARY</code>—operation failed, failure from message support library • <code>IX_CC_ERROR_NULL</code>—operation failed, invalid pointer • <code>IX_CC_ERROR_OOM</code>—operation failed, memory allocation failure
--------------	--

18.3.2.1 `ix_cc_natpt_cb_get_config_parameters`

This is the function prototype for the callback function used to return message-specific data (configuration parameters) to the calling application.

C Syntax

```
ix_error (*ix_cc_natpt_cb_get_config_parameters) (
    ix_error arg_Result,
    void * arg_pClientContext,
    ix_cc_natpt_config_params * arg_pConfig);
```

Input

<code>arg_pClientContext</code>	Context pointer supplied by the calling application in the original message API call.
<code>arg_pConfig</code>	Pointer to the structure containing configuration parameters. Pointer is not valid after the callback returns.

Output/Returns

<code>arg_Result</code>	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded.
-------------------------	--

18.3.3 `ix_cc_natpt_async_add_static_mapping()`

Sends a message of type `IX_CC_NATPT_ADD_STATIC_MAP` to the translation core component.

C Syntax

```
ix_error ix_cc_natpt_async_add_static_mapping(
    ix_cc_natpt_static_v6v4map * arg_pv6v4Map,
    ix_cc_natpt_cb arg_Callback,
    void * arg_pClientContext);
```

Input

<code>arg_pv6v4Map</code>	Pointer to a v6 v4 address mapping structure.
<code>arg_Callback</code>	Pointer to the calling application-supplied callback function. See <code>ix_cc_natpt_cb()</code> .
<code>arg_pClientContext</code>	Pointer to the calling application-defined context to the callback function.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. <code>IX_CC_NATPT_ERROR_MSG_LIBRARY</code>—operation failed, failure from message support library <code>IX_CC_ERROR_NULL</code>—operation failed, invalid pointer <code>IX_CC_ERROR_OOM</code>—operation failed, memory allocation failure
--------------	--

18.3.4 [`ix_cc_natpt_async_remove_static_mapping\(\)`](#)

Sends a message of type `IX_CC_NATPT_REM_STATIC_MAP` to the translation core component.

C Syntax

```
ix_error ix_cc_natpt_async_remove_static_mapping(
    ix_cc_natpt_static_v6v4map * arg_pv6v4Map,
    ix_cc_natpt_cb arg_Callback,
    void * arg_pClientContext);
:
```

Input

<code>arg_pv6v4Map</code>	Pointer to a v6 v4 address mapping structure.
<code>arg_Callback</code>	Pointer to the calling application-supplied callback function. See <code>ix_cc_natpt_cb()</code> .
<code>arg_pClientContext</code>	Pointer to the calling application-defined context passed to the callback function.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • <code>IX_CC_NATPT_ERROR_MSG_LIBRARY</code>—operation failed, failure from message support library • <code>IX_CC_ERROR_NULL</code>—operation failed, invalid pointer • <code>IX_CC_ERROR_OOM</code>—operation failed, memory allocation failure
--------------	--

18.3.5 `ix_cc_natpt_async_get_static_mapping()`

Sends a message of type `IX_CC_NATPT_GET_STATIC_MAP` to the translation core component.

C Syntax

```
ix_error ix_cc_natpt_async_get_static_mapping(
    ix_cc_natpt_cb_get_static_mapping arg_Callback,
    void * arg_pClientContext);
```

Input

<code>arg_Callback</code>	Pointer to the calling application-supplied callback function. See ix_cc_natpt_cb_get_static_mapping .
<code>arg_pClientContext</code>	Pointer to the calling application-defined context passed to the callback function.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • <code>IX_CC_NATPT_ERROR_MSG_LIBRARY</code>—operation failed, failure from message support library • <code>IX_CC_ERROR_NULL</code>—operation failed, invalid pointer • <code>IX_CC_ERROR_OOM</code>—operation failed, memory allocation failure
--------------	--

18.3.5.1 `ix_cc_natpt_cb_get_static_mapping`

This is the function prototype for the callback function used to return message-specific data (static mappings) to the calling application.

C Syntax

```
ix_error (*ix_cc_natpt_cb_get_static_mapping) (
    ix_error arg_Result,
    void * arg_pClientContext,
    ix_uint32 arg_NumItems,
    ix_cc_natpt_static_v6v4map * arg_pv6v4MapArray);
```

Input

<code>arg_pClientContext</code>	Context pointer supplied by the calling application in the original message API call.
<code>arg_NumItems</code>	Number of mappings in the array pointed to by the next argument.
<code>arg_pv6v4MapArray</code>	Pointer to an array of structures containing static v6 to v4 mappings. Pointer is not valid after the callback returns.

Output/Returns

<code>arg_Result</code>	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded.
-------------------------	--

18.3.6 `ix_cc_natpt_async_add_v4addr_pool()`

Sends a message of type `IX_CC_NATPT_ADD_V4ADDR_POOL` to the translation core component.

C Syntax

```
ix_error ix_cc_natpt_async_add_v4addr_pool(
    ix_uint32 arg_NumItems,
    ix_cc_natpt_v4addr * arg_pv4addrArray,
    ix_cc_natpt_cb arg_Callback,
    void * arg_pClientContext);
:
```

Input

<code>arg_NumItems</code>	Number of v4 addresses in the array pointed to by the next argument.
<code>arg_pv4addrArray</code>	Pointer to a v4 address array.
<code>arg_Callback</code>	Pointer to the calling application-supplied callback function. See ix_cc_natpt_cb() .

Input (Continued)

`arg_pClientContext` Pointer to the calling application-defined context passed to the callback function.

Output/Returns

Return Value Returns a valid `ix_error`.

- `IX_SUCCESS`—the operation succeeded.
- `IX_CC_NATPT_ERROR_MSG_LIBRARY`—operation failed, failure from message support library
- `IX_CC_ERROR_NULL`—operation failed, invalid pointer
- `IX_CC_ERROR_OOM`—operation failed, memory allocation failure

18.3.7 `ix_cc_natpt_async_remove_v4addr_pool()`

Sends a message of type `IX_CC_NATPT_REM_V4ADDR_POOL` to the translation core component.

C Syntax

```
ix_error ix_cc_natpt_async_remove_v4addr_pool(
    ix_unit32 arg_NumItems,
    ix_cc_natpt_v4addr * arg_pv4addrArray,
    ix_cc_natpt_cb arg_Callback,
    void * arg_pClientContext);
```

Input

`arg_NumItems` Number of v4 addresses in the array pointed to by the next argument.

`arg_pv4addrArray` Pointer to a v4 address array.

`arg_Callback` Pointer to the calling application-supplied callback function. See [ix_cc_natpt_cb\(\)](#).

`arg_pClientContext` Pointer to the calling application-defined context passed to the callback function.

Output/Returns

Return Value Returns a valid `ix_error`.

- `IX_SUCCESS`—the operation succeeded.
- `IX_CC_NATPT_ERROR_MSG_LIBRARY`—operation failed, failure from message support library
- `IX_CC_ERROR_NULL`—operation failed, invalid pointer
- `IX_CC_ERROR_OOM`—operation failed, memory allocation failure

18.3.8 ix_cc_natpt_async_get_v4addr_pool()

Sends a message of type IX_CC_NATPT_GET_V4ADDR_POOL to the translation core component.

C Syntax

```
ix_error ix_cc_natpt_async_get_v4addr_pool(
    ix_cc_natpt_cb_get_v4addr_pool arg_Callback,
    void * arg_pClientContext);
:
```

Input

arg_Callback	Pointer to the calling application-supplied callback function. See ix_cc_natpt_cb_get_v4addr_pool .
arg_pClientContext	Pointer to the calling application-defined context passed to the callback function.

Output/Returns

Return Value	Returns a valid ix_error. <ul style="list-style-type: none"> IX_SUCCESS—the operation succeeded. IX_CC_NATPT_ERROR_MSG_LIBRARY—operation failed, failure from message support library IX_CC_ERROR_NULL—operation failed, invalid pointer IX_CC_ERROR_OOM—operation failed, memory allocation failure
--------------	--

18.3.8.1 ix_cc_natpt_cb_get_v4addr_pool

This is the function prototype for the callback function used to return message-specific data (IPv4 address pool) to the calling application.

C Syntax

```
ix_error (*ix_cc_natpt_cb_get_v4addr_pool) (
    ix_error arg_Result,
    void * arg_pClientContext,
    ix_uint32 arg_NumItems,
    ix_cc_natpt_v4addr * arg_pv4addrArray);
```

Input

<code>arg_pClientContext</code>	Context pointer supplied by the calling application in the original message API call.
<code>arg_NumItems</code>	Number of v4 addresses in the array pointed to by the next argument.
<code>arg_pv4addrArray</code>	Pointer to an array of structures containing static v6 to v4 mappings. Pointer is not valid after the callback returns.

Output/Returns

<code>arg_Result</code>	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded.
-------------------------	--

18.3.9 `ix_cc_natpt_async_set_napt_mode()`

Sends a message of type `IX_CC_NATPT_SET_NAPT_MODE` to the translation core component.

C Syntax

```
ix_error ix_cc_natpt_async_set_napt_mode(
    ix_cc_natpt_naptmode arg_NaptMode,
    ix_cc_natpt_cb arg_Callback,
    void * arg_pClientContext);
:
```

Input

<code>arg_NaptMode</code>	NAPT mode flag.
<code>arg_Callback</code>	Pointer to the calling application-supplied callback function. See ix_cc_natpt_cb()
<code>arg_pClientContext</code>	Pointer to the calling application-defined context passed to the callback function.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. <code>IX_CC_NATPT_ERROR_MSG_LIBRARY</code>—operation failed, failure from message support library. <code>IX_CC_ERROR_NULL</code>—operation failed, invalid pointer.
--------------	--

18.3.10 `ix_cc_natpt_async_add_v4v6port_mapping()`

Sends a message of type `IX_CC_NATPT_ADD_V4V6_PORT_MAP` to the translation core component.

C Syntax

```
ix_error ix_cc_natpt_async_add_v4v6port_mapping(
    ix_cc_natpt_v4v6portmap * arg_pv4v6portMap,
    ix_cc_natpt_cb arg_Callback,
    void * arg_pClientContext);
```

Input

<code>arg_pv4v6portMap</code>	Pointer to a v4 to v6 address with port number mapping structure.
<code>arg_Callback</code>	Pointer to the calling application-supplied callback function. See ix_cc_natpt_cb() .
<code>arg_pClientContext</code>	Pointer to the calling application-defined context passed to the callback function.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. <code>IX_CC_NATPT_ERROR_MSG_LIBRARY</code>—operation failed, failure from message support library <code>IX_CC_ERROR_NULL</code>—operation failed, invalid pointer
--------------	--

18.3.11 `ix_cc_natpt_async_remove_v4v6port_mapping()`

Sends a message of type `IX_CC_NATPT_REM_V4V6_PORT_MAP` to the translation core component.

C Syntax

```
ix_error ix_cc_natpt_async_remove_v4v6port_mapping(
    ix_cc_natpt_v4v6portmap * arg_pv4v6portMap,
    ix_cc_natpt_cb arg_Callback,
    void * arg_pClientContext);
```

:

Input

<code>arg_pv4v6portMap</code>	Pointer to a v4 to v6 address with port number mapping structure.
<code>arg_Callback</code>	Pointer to the calling application-supplied callback function. See ix_cc_natpt_cb() .
<code>arg_pClientContext</code>	Pointer to the calling application-defined context passed to the callback function.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. <code>IX_CC_NATPT_ERROR_MSG_LIBRARY</code>—operation failed, failure from message support library <code>IX_CC_ERROR_NULL</code>—operation failed, invalid pointer
--------------	--

18.3.12 `ix_cc_natpt_async_get_v4v6port_mapping()`

Sends a message of type `IX_CC_NATPT_GET_V6V4_PORT_MAP` to the translation core component.

C Syntax

```
ix_error ix_cc_natpt_async_get_v4v6port_mapping(
    ix_cc_natpt_cb_get_v4v6port_mapping arg_Callback,
    void * arg_pClientContext);
```

Input

<code>arg_Callback</code>	Pointer to the calling application-supplied callback function. See ix_cc_natpt_cb_get_v4v6port_mapping .
<code>arg_pClientContext</code>	Pointer to the calling application-defined context passed to the callback function.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. <code>IX_CC_NATPT_ERROR_MSG_LIBRARY</code>—operation failed, failure from message support library <code>IX_CC_ERROR_NULL</code>—operation failed, invalid pointer
--------------	--

18.3.12.1 `ix_cc_natpt_cb_get_v4v6port_mapping`

This is the function prototype for the callback function used to return message-specific data (IPv4 address to IPv6 address with port number mappings) to the calling application.

C Syntax

```
ix_error (*ix_cc_natpt_cb_get_v4v6port_mapping) (
    ix_error arg_Result,
    void * arg_pClientContext,
    ix_uint32 arg_NumItems,
    ix_cc_natpt_v4v6portmap * arg_pv4v6portMapArray);
```

Input

<code>arg_pClientContext</code>	Context pointer supplied by the calling application in the original message API call.
<code>arg_NumItems</code>	Number of mappings in the array pointed by the next argument.
<code>arg_pv4v6portMapArray</code>	Pointer to an array of structures containing v4 to v6 address with port number mappings. Pointer is not valid after the callback returns.

Output/Returns

<code>arg_Result</code>	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded.
-------------------------	--

18.3.13 `ix_cc_natpt_async_get_active_sessions()`

Sends a message of type `IX_CC_NATPT_GET_ACTIVE_SESSIONS` to the translation core component.

C Syntax

```
ix_error ix_cc_natpt_async_get_active_sessions (
    ix_cc_natpt_cb_get_active_sessions arg_Callback,
    void * arg_pClientContext);
```

Input

<code>arg_Callback</code>	Pointer to the calling application-supplied callback function. See ix_cc_natpt_cb_get_active_sessions .
<code>arg_pClientContext</code>	Pointer to the calling application-defined context passed to the callback function.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • <code>IX_CC_NATPT_ERROR_MSG_LIBRARY</code>—operation failed, failure from message support library • <code>IX_CC_ERROR_NULL</code>—operation failed, invalid pointer • <code>IX_CC_ERROR_OOM</code>—operation failed, memory allocation failure
--------------	--

18.3.13.1 `ix_cc_natpt_cb_get_active_sessions`

This is the function prototype for the callback function used to return message-specific data (active NAT-PT sessions) to the calling application.

C Syntax

```
ix_error (*ix_cc_natpt_cb_get_active_sessions) (
    ix_error arg_Result,
    void * arg_pClientContext,
    ix_uint32 arg_NumItems,
    ix_cc_natpt_session * arg_pSessionArray);
```

Input

<code>arg_pClientContext</code>	Context pointer supplied by the calling application in the original message API call.
<code>arg_NumItems</code>	Number of mappings in the array pointed to by the next argument.
<code>arg_pSessionArray</code>	Pointer to an array of structures containing current NAT-PT sessions. Pointer is not valid after the callback returns.

Output/Returns

<code>arg_Result</code>	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded.
-------------------------	--

18.4 Library API

The library API consists of synchronous operations for the translation core component and is used by the asynchronous message support mechanism. [Table 18-6](#) lists the library API in the translation core component.

Table 18-6. Library API in the Translation Core Component

Message Helper API	Description
<code>ix_cc_natpt_set_config_parameters()</code>	Sets global translation core component configuration parameters
<code>ix_cc_natpt_get_config_parameters()</code>	Retrieves current global configuration parameters
<code>ix_cc_natpt_add_static_mapping()</code>	Adds a static IPv6 address to IPv4 address mapping
<code>ix_cc_natpt_remove_static_mapping()</code>	Removes an existing static IPv6 address to IPv4 address mapping
<code>ix_cc_natpt_get_static_mapping()</code>	Retrieves existing static IPv6 address to IPv4 address mappings
<code>ix_cc_natpt_add_v4addr_pool()</code>	Adds one or more IPv4 addresses used to map IPv6 addresses dynamically
<code>ix_cc_natpt_remove_v4addr_pool()</code>	Removes one or more existing IPv4 addresses from the IPv4 address pool
<code>ix_cc_natpt_get_v4addr_pool()</code>	Retrieves existing IPv4 addresses in the IPv4 address pool
<code>ix_cc_natpt_set_napt_mode()</code>	Turns ON/OFF NAPT mode of operation
<code>ix_cc_natpt_add_v4v6port_mapping()</code>	Adds an IPv4 address, IPv6 address and Protocol/Port Number static mapping
<code>ix_cc_natpt_remove_v4v6port_mapping()</code>	Removes an existing static mapping of IPv4 address, IPv6 address and Port Number
<code>ix_cc_natpt_get_v4v6port_mapping()</code>	Retrieves an existing static IPv6 address, IPv4 address and Port Number mappings
<code>ix_cc_natpt_get_active_sessions()</code>	Retrieves the list of currently active NATPT sessions

Note: The following functions are relevant to the NAPT-PT mode return `IX_CC_NATPT_ERROR_NOT_IMPLEMENTED`.

- `ix_cc_natpt_set_napt_mode()`
- `ix_cc_natpt_add_v4v6port_mapping()`
- `ix_cc_natpt_remove_v4v6port_mapping()`
- `ix_cc_natpt_get_v4v6port_mapping()`

18.4.1 `ix_cc_natpt_set_config_parameters()`

This function sets global translation core component configuration parameters.

C Syntax

```
ix_error ix_cc_natpt_set_config_parameters(
    ix_cc_natpt_config_params * arg_pConfig,
```

```
void * arg_pContext);
```

Input

<code>arg_pConfig</code>	Pointer to configuration parameters structure.
<code>arg_pContext</code>	Pointer to translation context.

Output/Returns

Return Value	<p>Returns a valid <code>ix_error</code>.</p> <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • <code>IX_CC_ERROR_NULL</code>—operation failed, invalid pointer • <code>IX_CC_NATPT_ERROR_INVALID_CONTEXT</code>—operation failed, the information pointed to by <code>arg_pContext</code> is invalid.
--------------	---

18.4.2 ix_cc_natpt_get_config_parameters()

This function retrieves current global configuration parameters. A pointer to an allocated `ix_cc_natpt_config_params` structure must be passed.

C Syntax

```
ix_error ix_cc_natpt_get_config_parameters(
    ix_cc_natpt_config_params * arg_pConfig,
    void * arg_pContext);
```

Input

`arg_pContext` Pointer to translation context.

Output
Arguments:

Output/Returns

`arg_pConfig` Pointer to configuration parameters structure that is filled.

Return Value Returns a valid `ix_error`.

- `IX_SUCCESS`—the operation succeeded.
- `IX_CC_ERROR_NULL`—operation failed, invalid pointer
- `IX_CC_NATPT_ERROR_INVALID_CONTEXT`—operation failed, the information pointed to by `arg_pContext` is invalid.

18.4.3 ix_cc_natpt_add_static_mapping()

This function adds a static IPv6 address to IPv4 address mapping. It is essential to add a static mapping for the IPv6 DNS server residing in the IPv6 realm. However, more static mappings may be added as necessary by invoking this function multiple times. The IPv4 addresses used in the static mappings cannot be used for dynamic mapping (given via `ix_cc_natpt_get_v4addr_pool()`).

C Syntax

```
ix_error ix_cc_natpt_add_static_mapping(
    ix_cc_natpt_static_v6v4map * arg_pv6v4Map,
    void * arg_pContext);
```

<code>arg_pv6v4Map</code>	Pointer to a v6 v4 address mapping structure.
<code>arg_pContext</code>	Pointer to translation context.

Output/Returns

Return Value	<p>Returns a valid <code>ix_error</code>.</p> <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • <code>IX_CC_ERROR_NULL</code>—operation failed, invalid pointer • <code>IX_CC_ERROR_OOM</code>—operation failed, memory allocation failure • <code>IX_CC_NATPT_ERROR_INVALID_CONTEXT</code>—operation failed, the information pointed to by <code>arg_pContext</code> is invalid. • <code>IX_CC_NATPT_ERROR_INVALID_INPUT_PARAM</code>—operation failed, the information pointed to by <code>arg_pv6v4Map</code> is invalid.
--------------	---

18.4.4 `ix_cc_natpt_remove_static_mapping()`

This function removes an existing static IPv6 address to IPv4 address mapping. Multiple static mappings may be removed by invoking this function multiple times.

To remove all static mapping entries at once, fill in the `arg_pv6v4Map` members as follows:

```
v6addr=ffff:ffff:ffff:ffff:ffff:ffff:ffff:ffff and v4addr=255.255.255.255
```

C Syntax

```
ix_error ix_cc_natpt_remove_static_mapping(
    ix_cc_natpt_static_v6v4map * arg_pv6v4Map,
    void * arg_pContext);
```

Input

<code>arg_pv6v4Map</code>	Pointer to a v6 v4 address mapping structure.
<code>arg_pContext</code>	Pointer to translation context.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • <code>IX_CC_ERROR_NULL</code>—operation failed, invalid pointer • <code>IX_CC_ERROR_OOM</code>—operation failed, memory allocation failure • <code>IX_CC_NATPT_ERROR_INVALID_CONTEXT</code>—operation failed, the information pointed to by <code>arg_pContext</code> is invalid. • <code>IX_CC_NATPT_ERROR_INVALID_INPUT_PARAM</code>—operation failed, the information pointed to by <code>arg_pv6v4Map</code> is invalid.
--------------	---

18.4.5 `ix_cc_natpt_get_static_mapping()`

This function retrieves existing static IPv6 address to IPv4 address mappings. The information is filled into the pre-allocated buffer passed by the caller.

If the error `IX_CC_NATPT_ERROR_BUFFER_TOO_SMALL` is returned, the calling application can allocate a larger buffer and call this function again.

C Syntax

```
ix_error ix_cc_natpt_get_static_mapping(
    ix_uint32 * arg_NumItems,
    ix_cc_natpt_static_v6v4map * arg_pv6v4MapArray,
    void * arg_pContext);
```

Input

<code>arg_pContext</code>	Pointer to the translation context.
---------------------------	-------------------------------------

Input/Output

<code>arg_NumItems</code>	<ul style="list-style-type: none"> • On input—the number of entries that the buffer pointed to by <code>arg_pv6v4MapArray</code> can accommodate. • On output—the actual number of entries written to the buffer. If <code>IX_CC_NATPT_ERROR_BUFFER_TOO_SMALL</code> is returned, then this number is the total number of entries that exist.
---------------------------	---

Output Arguments:

Output/Returns

<code>arg_pv6v4MapArray</code>	Pointer to the buffer into which the array of v6-v4 address mappings are returned.
Return Value	<p>Returns a valid <code>ix_error</code>.</p> <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. <code>IX_CC_ERROR_NULL</code>—operation failed, invalid pointer <code>IX_CC_ERROR_OOM</code>—operation failed, memory allocation failure <code>IX_CC_NATPT_ERROR_INVALID_CONTEXT</code>—operation failed, the information pointed to by <code>arg_pContext</code> is invalid. <code>IX_CC_NATPT_ERROR_INVALID_INPUT_PARAM</code>—operation failed, <code>arg_NumItems</code> is zero. <code>IX_CC_NATPT_ERROR_BUFFER_TOO_SMALL</code>—operation failed, more items exist than could fit into the supplied buffer (non-fatal error).

18.4.6 `ix_cc_natpt_add_v4addr_pool()`

This function adds one or more IPv4 addresses used to map IPv6 addresses dynamically. The IPv4 addresses given via this function cannot be used for static IPv4 to IPv6 mapping (added via `ix_cc_natpt_add_static_mapping()` function). When this function is called multiple times, the entire list of IPv4 addresses are replaced.

C Syntax

```
ix_error ix_cc_natpt_add_v4addr_pool(
    ix_unit32 arg_NumItems,
    ix_cc_natpt_v4addr * arg_pv4addrArray,
    void * arg_pContext);
```

:

Input

<code>arg_NumItems</code>	Number of v4 addresses in the array pointed by the next argument.
<code>arg_pv4addrArray</code>	Pointer to a v4 address array.
<code>arg_pContext</code>	Pointer to translation context.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • <code>IX_CC_ERROR_NULL</code>—operation failed, invalid pointer • <code>IX_CC_ERROR_OOM</code>—operation failed, memory allocation failure • <code>IX_CC_NATPT_ERROR_INVALID_CONTEXT</code>—operation failed, the information pointed to by <code>arg_pContext</code> is invalid. • <code>IX_CC_NATPT_ERROR_INVALID_INPUT_PARAM</code>—operation failed, the information pointed to by <code>arg_pv6v4Map</code> is invalid.
--------------	---

18.4.7 `ix_cc_natpt_remove_v4addr_pool()`

This function removes one or more existing IPv4 addresses from the IPv4 address pool.

C Syntax

```
ix_error ix_cc_natpt_remove_v4addr_pool(
    ix_unit32 arg_NumItems,
    ix_cc_natpt_v4addr * arg_pv4addrArray,
    void * arg_pContext);
:
```

Input

<code>arg_NumItems</code>	Number of v4 addresses in the array pointed by the next argument.
<code>arg_pv4addrArray</code>	Pointer to a v4 address array.
<code>arg_pContext</code>	Pointer to translation context.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • <code>IX_CC_ERROR_NULL</code>—operation failed, invalid pointer • <code>IX_CC_ERROR_OOM</code>—operation failed, memory allocation failure • <code>IX_CC_NATPT_ERROR_INVALID_CONTEXT</code>—operation failed, the information pointed to by <code>arg_pContext</code> is invalid. • <code>IX_CC_NATPT_ERROR_INVALID_INPUT_PARAM</code>—operation failed, the information pointed to by <code>arg_pv4addrArray</code> is invalid.
--------------	---

18.4.8 ix_cc_natpt_get_v4addr_pool()

This function retrieves existing IPv4 addresses in the IPv4 address pool. The information is filled into the pre-allocated buffer passed by the caller. If the error `IX_CC_NATPT_ERROR_BUFFER_TOO_SMALL` is returned, the calling application can allocate a larger buffer and call this function again.

C Syntax

```
ix_error ix_cc_natpt_get_v4addr_pool(
    ix_uint32 * arg_NumItems,
    ix_cc_natpt_v4addr * arg_pv4addrArray);
void * arg_pContext);
```

Input

`arg_pContext` Pointer to translation context.

Input/Output Arguments:

Input/Output

`arg_NumItems`

- On input—the number of entries that the buffer pointed to by `arg_pv4addrArray` can accommodate.
- On output—the actual number of entries written to the buffer. If `IX_CC_NATPT_ERROR_BUFFER_TOO_SMALL` is returned, then this number is the total number of entries that exist.

Output/Returns

`arg_pv4addrArray` Pointer to buffer into which the array of IPv4 addresses is returned.

Return Value Returns a valid `ix_error`.

- `IX_SUCCESS`—the operation succeeded.
- `IX_CC_ERROR_NULL`—operation failed, invalid pointer
- `IX_CC_ERROR_OOM`—operation failed, memory allocation failure
- `IX_CC_NATPT_ERROR_INVALID_CONTEXT`—operation failed, the information pointed to by `arg_pContext` is invalid.
- `IX_CC_NATPT_ERROR_INVALID_INPUT_PARAM`—operation failed, `arg_NumItems` is zero.
- `IX_CC_NATPT_ERROR_BUFFER_TOO_SMALL`—operation failed, more items exist than could fit into the supplied buffer (non-fatal error).

18.4.9 ix_cc_natpt_set_napt_mode()

This function turns ON/OFF NAPT mode of operation. When NAPT mode is OFF, the IPv4 addresses in the pool are dynamically selected for mapping IPv6 addresses until all the IPv4 addresses are in use. When NAPT mode is ON, only one IPv4 address from the pool is selected and TCP/UDP port numbers are dynamically used to map sessions. Technically, more than 60,000 sessions each for TCP and UDP can be multiplexed with a single IPv4 address.

C Syntax

```
ix_error ix_cc_natpt_set_natpt_mode(
    ix_cc_natpt_natptmode arg_NaptMode,
    void * arg_pContext);
```

:

Input

<code>arg_NaptMode</code>	NAPT mode flag.
<code>arg_pContext</code>	Pointer to translation context.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> .
--------------	---

- `IX_SUCCESS`—the operation succeeded.
- `IX_CC_ERROR_NULL`—operation failed, invalid pointer.

18.4.10 `ix_cc_natpt_add_v4v6port_mapping()`

This function adds an IPv4 address, IPv6 address and Protocol/Port Number static mapping. The IPv4 address is not specifically given in the structure, since it would have been already automatically chosen when NAPT mode is turned ON. This is typically used for assigning a permanent IPv4 address for the IPv6 servers. For example, an entry for web server could be “web server IPv6 address, permanent IPv4 address, and port number 80”. Multiple static mappings may be added as necessary by invoking this function multiple times.

NAPT mode must be turned ON prior to adding this kind of mapping.

C Syntax

```
ix_error ix_cc_natpt_add_v4v6port_mapping(
    ix_cc_natpt_v4v6portmap * arg_pv4v6portMap,
    void * arg_pContext);
```

Input

<code>arg_pv4v6portMap</code>	Pointer to a v4 to v6 address with port number mapping structure.
<code>arg_pContext</code>	Pointer to translation context.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • <code>IX_CC_ERROR_NULL</code>—operation failed, invalid pointer • <code>IX_CC_ERROR_OOM</code>—operation failed, memory allocation failure • <code>IX_CC_NATPT_ERROR_INVALID_CONTEXT</code>—operation failed, the information pointed to by <code>arg_pContext</code> is invalid. • <code>IX_CC_NATPT_ERROR_INVALID_INPUT_PARAM</code>—operation failed, the information pointed to by <code>arg_pv4v6portMap</code> is invalid.
--------------	---

18.4.11 `ix_cc_natpt_remove_v4v6port_mapping()`

This function removes an existing static mapping of IPv4 address, IPv6 address and Port Number. Multiple mappings may be removed by invoking this function multiple times.

To remove all mapping entries at once, fill in the `arg_pv4v6portMap` members as follows:

- `v6addr=ffff:ffff:ffff:ffff:ffff:ffff:ffff:ffff`
- `v4addr=255.255.255.255`
- `portNumber=65535`

C Syntax

```
ix_error ix_cc_natpt_remove_v4v6port_mapping(
    ix_cc_natpt_v4v6portmap * arg_pv4v6portMap,
    void * arg_pContext);
:
```

Input

<code>arg_pv4v6portMap</code>	Pointer to a v4 to v6 address with port number mapping structure.
<code>arg_pContext</code>	Pointer to translation context.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • <code>IX_CC_ERROR_NULL</code>—operation failed, invalid pointer • <code>IX_CC_ERROR_OOM</code>—operation failed, memory allocation failure • <code>IX_CC_NATPT_ERROR_INVALID_CONTEXT</code>—operation failed, the information pointed to by <code>arg_pContext</code> is invalid. • <code>IX_CC_NATPT_ERROR_INVALID_INPUT_PARAM</code>—operation failed, the information pointed to by <code>arg_pv4v6portMap</code> is invalid.
--------------	---

18.4.12 ix_cc_natpt_get_v4v6port_mapping()

This function retrieves existing static IPv6 address, IPv4 address and Port Number mappings. The information is filled into the pre-allocated buffer passed by the caller.

If the error `IX_CC_NATPT_ERROR_BUFFER_TOO_SMALL` is returned, the calling application can allocate a larger buffer and call this function again.

C Syntax

```
ix_error ix_cc_natpt_get_v4v6port_mapping(
    ix_uint32 * arg_NumItems,
    ix_cc_natpt_v4v6portmap * arg_pv4v6portMapArray,
    void * arg_pContext);
```

Input

`arg_pContext` Pointer to translation context.

Input/Output

`arg_NumItems`

- On input—the number of entries that the buffer pointed to by `arg_pv4v6portMapArray` can accommodate.
- On output—the actual number of entries written to the buffer. If `IX_CC_NATPT_ERROR_BUFFER_TOO_SMALL` is returned, then this number is the total number of the existing entries.

Output/Returns

`arg_pv4v6portMapArray` Pointer to an array of structures containing v4 to v6 address with port number mappings.

Return Value Returns a valid `ix_error`.

- `IX_SUCCESS`—the operation succeeded.
- `IX_CC_ERROR_NULL`—operation failed, invalid pointer
- `IX_CC_ERROR_OOM`—operation failed, memory allocation failure
- `IX_CC_NATPT_ERROR_INVALID_CONTEXT`—operation failed, the information pointed to by `arg_pContext` is invalid.
- `IX_CC_NATPT_ERROR_INVALID_INPUT_PARAM`—operation failed, `arg_NumItems` is zero.
- `IX_CC_NATPT_ERROR_BUFFER_TOO_SMALL`—operation failed, more items exist than could fit into the supplied buffer (non-fatal error).

18.4.13 ix_cc_natpt_get_active_sessions()

This function retrieves the list of currently active NATPT sessions. The information is filled into the pre-allocated buffer passed by the caller.

If the error `IX_CC_NATPT_ERROR_BUFFER_TOO_SMALL` is returned, the calling application can allocate a larger buffer and call this function again.

C Syntax

```
ix_error ix_cc_natpt_get_active_sessions (
    ix_uint32 * arg_NumItems,
    ix_cc_natpt_session * arg_pSessionArray,
    void * arg_pContext);
```

Input

`arg_pContext` Pointer to translation context.

Input/Output

`arg_NumItems`

- On input—the number of entries that the buffer pointed to by `arg_pSessionArray` can accommodate.
- On output—the actual number of entries written to the buffer. If `IX_CC_NATPT_ERROR_BUFFER_TOO_SMALL` is returned, then this number is the total number of entries that exist.

Output Arguments:

Output/Returns

`arg_pSessionArray` Pointer to an array of structures containing current NATPT sessions.

Return Value Returns a valid `ix_error`.

- `IX_SUCCESS`—the operation succeeded.
- `IX_CC_ERROR_NULL`—operation failed, invalid pointer
- `IX_CC_ERROR_OOM`—operation failed, memory allocation failure
- `IX_CC_NATPT_ERROR_INVALID_CONTEXT`—operation failed, the information pointed to by `arg_pContext` is invalid.
- `IX_CC_NATPT_ERROR_INVALID_INPUT_PARAM`—operation failed, `arg_NumItems` is zero.
- `IX_CC_NATPT_ERROR_BUFFER_TOO_SMALL`—operation failed, more items exist than could fit into the supplied buffer (non-fatal error).

DiffServ Components

The following chapters are included in this section:

- [Chapter 19, “Six-Tuple Exact Match Classifier”](#)
- [Chapter 20, “Three Color Meter”](#)
- [Chapter 21, “Weighted Random Early Detection”](#)
- [Chapter 22, “DSCP Classifier”](#)

The core component provides the following functionality:

- Initializes and configures the 6-tuple classifier microblock by patching symbols.
- Provides an API interface to add and remove exact-match rules. The rules are stored in a hash table shared between the core component and the microblock.
- Implements exact-matching capability. There is no range matching classification in a core component (slow path). Range matching is available with TCAM support.
- Implements the basic classification function of IP packets without header options. The core component classifies local IP packets generated by the Stack Driver.
- Implements an extended classification function for exception packets thrown by the 6-tuple microblock. The exception packets are the packets with IP header options.

For complete details see [Chapter 56, “Six-Tuple Classifier Core Component”](#) of the *Intel® Internet Exchange Architecture Software Building Blocks Developer’s Manual*.

19.1 Data Structures

The data structures are used in calls to functional APIs—Message Helper API and Library API. [Table 19-1](#) lists the three data structures for configuring exact-match rules defined by the classifier core component.

Table 19-1. Data Structures for Configuring Exact-match Rules

Data Structure	Description
<code>ix_s_cc_classifier_6t_pattern</code>	A bit string defining the exact-match classification rule.
<code>ix_s_cc_classifier_6t_qos_result</code>	A QoS classification result associated with the specified rule.
<code>ix_s_cc_classifier_6t_fwd_result</code>	A forwarding classification result associated with the specified rule.
<code>ix_cc_classifier_6t_result</code>	A classification result (either forwarding or QoS). The structure includes a flag used to control statistics gathering for the specified rule.
<code>ix_cc_classifier_6t_statistics</code>	64-bit statistics associated with a rule

19.1.1 Classification Pattern Data Type

This data type defines an exact-match rule classification pattern. This is a packed structure, passed on as a byte string in a call to add/update/search/remove entry messages. The pattern is defined in the following section.

19.1.1.1 ix_s_cc_classifier_6t_pattern

A pattern comprising of source and destination address equal to zero denotes a default rule. In this case, all the other field values are ignored.

C Syntax

```
typedef structure ix_s_cc_classifier_6t_pattern
{
    ix_uint32 ipv4SourceAddress: 32;
    ix_uint32 ipv4DestinationAddress: 32;
    ix_uint32 inputPort: 16;
    ix_uint32padding: 2;
    ix_uint32 dscp: 6;
    ix_uint32 protocol: 8;
    ix_uint32 l4SourcePort: 16;
    ix_uint32 l4DestinationPort: 16;
} ix_cc_classifier_6t_pattern;
```

19.1.2 Classification Result Data Type

A classification pattern can be associated with two lookup results—QoS and forwarding information. Both QoS and forwarding rules can be configured independently, typically from different applications. To facilitate configuration, the core component provides an abstraction of two logical tables.

19.1.2.1 ix_cc_classifier_6t_table_type Enumeration

C Syntax

```
typedef enum {
    IX_CC_CLASSIFIER_6T_QOS, /* QoS rule */
    IX_CC_CLASSIFIER_6T_FWD, /* forwarding rule */
} ix_cc_classifier_6t_table_type;
```

19.1.2.2 ix_cc_classifier_6t_qos_output Enumeration

When the block sets QoS parameters, it can direct traffic either to DSCP marker or TC meter. The two outputs are specified using the following enumeration:

C Syntax

```
typedef enum {
    IX_CC_CLASSIFIER_6T_tc_meter_QOS_OUTPUT,
    IX_CC_CLASSIFIER_6T_DSCP_MARKER_QOS_OUTPUT
} ix_cc_classifier_6t_qos_output;
```

19.1.2.3 ix_s_cc_classifier_6t_qos_result

The QoS lookup result is defined as follows:

C Syntax

```
typedef structure ix_s_cc_classifier_6t_qos_result
{
    ix_uint32 flowId; /* QoS flow id used to police traffic */
    ix_uint16 classId; /* relative identifier of an output queue */
    ix_uint8 colorId; /* packet drop precedence level */
    ix_uint8 nextBlockId; /* identifier of a QoS microblock */
} ix_cc_classifier_6t_result;
```

The QoS structure with flowId value equal to 0xFFFFFFFF is invalid. The block internally uses it to mark the hash entries that do not have valid QoS information. The core component must not allow adding QoS entries with the value.

19.1.2.4 ix_s_cc_classifier_6t_fwd_result

This structure defines a forwarding result associated with an exact-match rule.

C Syntax

```
typedef struct ix_s_cc_classifier_6t_fwd_result
{
    /* forwarding lookup identifier */
    ix_uint16 NextHopId:16;

    /* type of forwarding lookup identifier */
    ix_uint8 NextHopIdType:4, Reserved:4;
} ix_cc_classifier_6t_fwd_result;
```

The forwarding structure with nextHopId value equal to 0xFFFF is invalid. The block internally uses the value to mark the hash entries that do not have valid QoS information. The core component must not allow adding forwarding entries with the value.

19.1.2.5 ix_cc_classifier_6t_result

This structure provides the API procedures and is used to configure forwarding or QoS classification result associated with a pattern. It is also used to enable or disable statistics gathering for traffic matching the pattern. The core component considers the value of stat_enable field only when the microblock supports statistics.

C Syntax

```
typedef structure ix_s_cc_classifier_6t_result
{
    ix_cc_classifier_6t_table_type type;
    union {
        ix_cc_classifier_6t_qos_result qos;
        ix_cc_classifier_6t_fwd_result fwd;
    };
};
```

```

    } result;
} ix_cc_classifier_6t_result;

```

19.1.3 Statistics Data Type

Statistics associated with 6-tuple classification rules consist of two 64-bit counters: packet counter and byte counter. The current counter values are returned using the structure:

19.1.3.1 ix_cc_classifier_6t_statistics

C Syntax

```

typedef struct ix_s_cc_classifier_6t_statistics
{
    /* number of packets matching the rule */
    ix_uint64 packets;
    /* number of bytes matching the rule */
    ix_uint64 bytes;
}ix_cc_classifier_6t_statistics;

```

19.2 Core Component Infrastructure API

Table 19-2 lists the Core Component Infrastructure API supported by 6-tuple classifier core components.

Table 19-2. 6-tuple Classifier Core Components Infrastructure API

API Function	Description
<code>ix_cc_classifier_6t_init()</code>	Initializes the core component
<code>ix_cc_classifier_6t_fini()</code>	Terminates the core component
<code>ix_cc_classifier_6t_pkt_handler()</code>	Message handler for processing rule add/remove requests
<code>ix_cc_classifier_6t_msg_handler()</code>	Packet handler for processing exception packets

19.2.1 ix_cc_classifier_6t_init()

The function initializes the core component. It should be called and returned successfully before any other function in the core component can be called. The function performs the following functions:

- Allocates DRAM/SRAM memory for the hash table and initializes it with empty entries (all zeros). The function calculates the amount of allocated memory using the formula:


```
size = CLASSIFIER_6T_HASH_ENTRY_SRAM_SIZE * 2 CLASSIFIER_6T_HASH_KEY_MASK
```
- Patches 6-tuple microcode with imported variables. The function gets the variable values from the system repository properties.
- Initializes a hardware hash unit with the 128-bit multiplicand.
- Registers a packet handler `ix_cc_classifier_6t_pkt_handler()` for inputs:
 - `IX_CC_CLASSIFIER_6T_EXCEPTION_PKT_INPUT` (single source mode)
 - `IX_CC_CLASSIFIER_6T_LOCAL_PKT_INPUT` (single source mode)
- Registers a message handler `ix_cc_classifier_6t_msg_handler()` for inputs:
 - `IX_CC_CLASSIFIER_6T_MSG_INPUT` (multiple sources mode)
- Allocates a control block, and returns a pointer to this block in `arg_ppContext`.

C Syntax

```
ix_error ix_cc_classifier_6t_init (
    ix_cc_handle arg_CcHandle,
    void **arg_ppContext);
```

Input

`arg_CcHandle` Handle to the core component.

Output/Returns

`arg_ppContext` Location where the pointer to the control block allocated by the core component is stored. The control block is internal to the core component and contains variables and internal data structures. This pointer is used later, and passed on to the `ix_cc_classifier_6t_fini()` function by core component infrastructure to free memory when the core component is terminated.

Output/Returns (Continued)

Return Value	<p>Returns a valid <code>ix_error</code>.</p> <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded, initialization completed successfully • <code>IX_CC_ERROR_NULL</code>—operation failed, <code>arg_ppContext</code> or <code>arg_CcHandle</code> argument is NULL • <code>IX_CC_ERROR_OOM</code>—operation failed, system memory allocation failure • <code>IX_CC_ERROR_OOM_SRAM</code>—operation failed, SRAM memory allocation failure • <code>IX_CC_ERROR_CLASSIFIER_6T_NOT_NULL</code>—operation failed, <code>*arg_ppContext</code> is not NULL • <code>IX_CC_CLASSIFIER_6T_ERROR_REPOSITORY_OPEN</code>—operation failed, the function failed to open the system repository • <code>IX_CC_CLASSIFIER_6T_ERROR_REPOSITORY_READ</code>—operation failed, the function failed to read from the system repository • <code>IX_CC_CLASSIFIER_6T_ERROR_HASH_TABLE_EXISTS</code>—operation failed, hash table already exists • <code>IX_CC_CLASSIFIER_6T_ERROR_HASH_IDX_SIZE</code>—operation failed, hash index size is 0 • <code>IX_CC_CLASSIFIER_6T_ERROR_SET_UCODE_FILE</code>—operation failed, the function failed to set the microcode image • <code>IX_CC_CLASSIFIER_6T_PATCH_SYMBOLS</code>—operation failed, the function failed to patch symbols • <code>IX_OSSL_ERROR_MUTEX_INIT_FAILED</code>—operation failed, the function failed to allocate a mutex
--------------	---

19.2.2 `ix_cc_classifier_6t_fini()`

The function terminates the 6-tuple classifier core component. It is executed when the execution engine running the core component is being shut down. This function performs the following tasks:

- Frees DRAM/SRAM memory allocated by the `ix_cc_classifier_6t_init()`.
- Removes a packet handler `ix_cc_classifier_6t_pkt_handler()`.
- Removes a message handler `ix_cc_classifier_6t_msg_handler()`.
- Frees an internal control block.

C Syntax

```
ix_error ix_cc_classifier_6t_fini (
    ix_cc_handle arg_CcHandle,
    void *arg_pContext);
```

Input

<code>arg_CcHandle</code>	Handle to the core component.
<code>arg_pContext</code>	Pointer to the control block allocated in <code>ix_cc_classifier_6t_init()</code> .

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded, shutdown completed successfully <code>IX_CC_ERROR_NULL</code>—operation failed, <code>arg_pContext</code> is NULL <code>IX_CC_CLASSIFIER_6T_ERROR_INVALID_CC_HANDLE</code>—operation failed, invalid data in <code>arg_CcHandle</code> <code>IX_CC_CLASSIFIER_6T_ERROR_INV_HASH_TABLE</code>—operation failed, hash table has not been created <code>IX_CC_CLASSIFIER_6T_ERROR_FREE_MEM</code>—operation failed, the function failed to free hash table memory
--------------	---

19.2.3 ix_cc_classifier_6t_pkt_handler()

The core component uses one packet handler for processing packets from two sources:

- Local IPv4 packets generated by Stack Driver
- Exception IPv4 packets with header options, thrown from the 6-tuple exact-match classifier microblock.

Packets other than IPv4 are dropped, and the handler reports an error.

For IPv4 packets, the function retrieves 6 fields from the IP and UDP/TCP headers. Next, it calculates a hash value and lookups the hash table. The lookup algorithm is identical as in the 6-tuple Exact Match Classifier Microblock described in *Intel® Internet Exchange Architecture Software Building Blocks Developer's Manual*.

The packet handler updates metadata variables according to the lookup result. Next, it uses the value in `nextBlockId` (or in `CLASSIFIER_6T_DEFAULT_NEXT_BLOCK` on lookup failure) to select one of the core component outputs. Table 19-3 shows the mapping rules applied when the next block identifier points to a microblock.:

Table 19-3. Microblock and Core Component Mapping Rules

Next Microblock ID	Core Component Output ID
CLASSIFIER_6T_METER	IX_CC_CLASSIFIER_6T_METER_OUTPUT
CLASSIFIER_6T_MARK	IX_CC_CLASSIFIER_6T_MARK_OUTPUT
CLASSIFIER_6T_FORWARD	IX_CC_CLASSIFIER_6T_FORWARD_OUTPUT
CLASSIFIER_6T_INV_IP	IX_CC_CLASSIFIER_6T_DEFAULT_OUTPUT

C Syntax

```
ix_error ix_cc_classifier_6t_pkt_handler (
    ix_buffer_handle arg_Pkt,
    ix_uint32 arg_ExceptionCode,
    void *arg_pContext)
```

Input

<code>arg_Pkt</code>	Buffer handle to the IP packet.
<code>arg_ExceptionCode</code>	Exception code setup by a 6-tuple classifier microblock. It equals to zero when Stack Driver originates the packet.
<code>arg_pContext</code>	Pointer to a control block allocated in <code>ix_cc_classifier_6t_init()</code> , that is passed to the core component when a packet arrives.

Output/Returns

Return Value	<p>Returns a valid <code>ix_error</code>.</p> <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded, packet processed successfully <code>IX_CC_ERROR_NULL</code>—operation failed, <code>arg_pContext</code> or <code>arg_Pkt</code> are NULL or the hash table has not been created <code>IX_CC_ERROR_INTERNAL</code>—operation failed, the function failed to get the packet meta data or the packet payload <code>IX_CC_ERROR_SEND_FAIL</code>—operation failed, reply message sending failure <code>IX_CC_CLASSIFIER_6T_ERROR_INVALID_BUF_HANDLE</code>—operation failed, buffer handle is NULL <code>IX_CC_CLASSIFIER_6T_ERROR_HASH_TIME_OUT</code>—operation failed, hash operation timed out
--------------	--

19.2.4 `ix_cc_classifier_6t_msg_handler()`

The core component uses one message handler for processing all types of messages. The 6-tuple core component receives messages from the Lookup Library, which serves as an entry point to higher-level components, such as DiffServ or MPLS/TE applications.

The suite of supported messages directly reflects a subset of Lookup Library APIs relevant to the exact-match classification. The handler function parses message payloads and calls the appropriate 6-tuple core component Library API functions to process the message.

C Syntax

```
ix_error ix_cc_classifier_6t_msg_handler(
    ix_buffer_handle arg_hDataToken,
    ix_uint32 arg_UserData,
    void *arg_pContext);
```

Input

<code>arg_hDataToken</code>	Buffer handle embedding information for the message passed in <code>arg_UserData</code> .
<code>arg_UserData</code>	Message type. Table 19-4 lists the supported messages.

Input (Continued)

`arg_pContext` Pointer to a control block allocated in `ix_cc_classifier_6t_init()` that is passed to the core component when a message arrives.

Table 19-4. 6-tuple Core Component Supported Message Types

Message type	Description
<code>IX_CC_CLASSIFIER_6T_MSG_ADD_ENTRY</code>	Adds a new entry to the 6-tuple hash table
<code>IX_CC_CLASSIFIER_6T_MSG_REMOVE_ENTRY</code>	Deletes an entry from the 6-tuple hash table
<code>IX_CC_CLASSIFIER_6T_MSG_UPDATE_ENTRY</code>	Updates an entry in the 6-tuple hash table.
<code>IX_CC_CLASSIFIER_6T_MSG_SEARCH_ENTRY</code>	Returns lookup results associated with a 6-tuple pattern.

Output/Returns

Return Value Returns a valid `ix_error`.

- `IX_SUCCESS`—the operation succeeded, message processed successfully
- `IX_CC_ERROR_NULL`—operation failed, `arg_pContext` is NULL or the hash table has not been created
- `IX_CC_ERROR_SEND_FAIL`—operation failed, reply message sending failure
- `IX_CC_ERROR_UNDEFINED_MSG`—operation failed, unknown message type received
- `IX_CC_CLASSIFIER_6T_ERROR_BUFF_FREE`—operation failed, buffer memory deallocation failure
- `IX_CC_CLASSIFIER_6T_ERROR_INVALID_BUF_HANDLE`—operation failed, buffer handle is NULL

19.3 Message Helper API

The Message Helper is a wrapper library that facilitates sending messages to the 6-tuple core component. There is one-to-one mapping between Message Helper APIs and message types supported by a core component. All APIs are asynchronous—a core component reports operation status in a callback routine. [Table 19-5](#) lists the 6-tuple core component message helper APIs.

Table 19-5. 6-tuple Core Component Message Helper API

API Function	Description
<code>ix_cc_classifier_6t_async_add_entry()</code>	Adds a new classification pattern and the associated lookup results to the 6-tuple hash table
<code>ix_cc_classifier_6t_async_remove_entry()</code>	Removes a classification pattern and lookup result from the 6-tuple hash table
<code>ix_cc_classifier_6t_async_update_entry()</code>	Updates selected fields of the lookup result associated with a specified classification pattern.
<code>ix_cc_classifier_6t_async_search_entry()</code>	Returns a lookup result associated with a specified classification pattern.
<code>ix_cc_classifier_6t_async_get_statistics()</code>	Returns statistics for a rule
<code>ix_cc_classifier_6t_async_get_statistics_def()</code>	Returns statistics for the default rule

19.3.1 `ix_cc_classifier_6t_async_add_entry()`

This message helper function builds and sends an entry add message to the classifier core component. The core component uses the classification pattern, `arg_pKey`, to prepare a hash input, employs a hardware hash unit to generate a hash table index and performs the following tasks:

- reports an error (via callback) and does not update lookup results associated with the existing rule, if the hash table already contains a specified classification pattern for a given table type.
- adds a new entry to the hash table, if a hash lookup fails. The lookup result, `arg_pLookupResult`, is stored in the newly added entry.

C Syntax

```
ix_error ix_cc_classifier_6t_async_add_entry (
    ix_cc_classifier_6t_pattern *arg_pKey,
    ix_cc_classifier_6t_result *arg_pLookupResult,
    ix_cc_classifier_6t_cb_add_entry arg_Callback,
    void* arg_pUserContext);
```

Input

`arg_pKey` Pointer to the classification pattern structure, which defines an exact-match rule. The structure is defined in [Section 19.1.1, “Classification Pattern Data Type”](#) on page 387.

Input (Continued)

<code>arg_pLookupResult</code>	Pointer to the structure of lookup results associated with the specified classification pattern. The type field identifies a logical lookup table; it should be <code>IX_CC_CLASSIFIER_6T_QOS</code> or <code>IX_CC_CLASSIFIER_6T_FWD</code> . The remaining fields specify lookup result, as defined in Section 19.1.2, “Classification Result Data Type” on page 388.
<code>arg_Callback</code>	A calling application-provided callback function. The core component invokes this function to report an operation status. See ix_cc_classifier_6t_cb_add_entry .
<code>arg_pUserContext</code>	Pointer to a calling application-defined context. This value is opaque to the core component. The context is returned in a callback function.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded, message was successfully built and sent <code>IX_CC_ERROR_OOM_SYSTEM</code>—operation failed, not enough memory to send a message <code>IX_CC_ERROR_NULL</code>—operation failed, NULL pointer in input parameters <code>IX_CC_ERROR_SEND_FAIL</code>—operation failed, failure at sending a message
--------------	--

19.3.1.1 ix_cc_classifier_6t_cb_add_entry

The function prototype for message-handler callback function provided by the calling application to return an operation status.

C Syntax

```
ix_error (*ix_cc_classifier_6t_cb_add_entry) (
    ix_error arg_Result,
    void* arg_pContext);
```

Input

<code>arg_pContext</code>	Pointer to client-defined context, provided as a parameter in the API call. Used by the caller to match asynchronous request with a response
---------------------------	--

Output/Returns

<code>arg_Result</code>	Indicates error conditions for the call with the following returns:
-------------------------	---

Output/Returns

Return Value	<ul style="list-style-type: none"> IX_SUCCESS—the operation succeeded, entry successfully added IX_CC_ERROR_DUPLICATE_ENTRY—operation failed, an entry for the key already configured IX_CC_CLASSIFIER_6T_ERROR_TIME_OUT—operation failed, timeout in communication with the hash unit IX_CC_CLASSIFIER_6T_ERROR_PARAM—operation failed, invalid parameter value specified
--------------	--

19.3.2 ix_cc_classifier_6t_async_remove_entry()

This message helper function builds and sends an entry remove message to the classifier core component. The core component uses the classification pattern, `arg_pKey`, to prepare a hash input, then employs a hardware hash unit to generate a hash table index and performs the following tasks:

- removes an entry, if the hash table already contains a specified classification pattern for a given table.
- reports an error and no entry is removed from the hash table, if a hash lookup fails.

C Syntax

```
ix_error ix_cc_classifier_6t_async_remove_entry(
    ix_cc_classifier_6t_pattern *arg_pKey,
    ix_cc_classifier_6t_table_type arg_Type;
    ix_cc_classifier_6t_cb_remove_entry arg_Callback,
    void* arg_pContext);
```

Input

<code>arg_pKey</code>	Pointer to the classification pattern structure, which defines an exact-match rule. The structure is defined in Section 19.1.1, “Classification Pattern Data Type” on page 387.
<code>arg_Type</code>	Identifier of a logical lookup table; it should be <code>IX_CC_CLASSIFIER_6T_QOS</code> or <code>IX_CC_CLASSIFIER.T_FWD</code>
<code>arg_Callback</code>	A calling application-provided callback function. The core component invokes this function to report an operation status. See ix_cc_classifier_6t_cb_remove_entry .
<code>arg_pUserContext</code>	Pointer to a calling application-defined context. This value is opaque to the core component. The context is returned in a callback function.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded, message was successfully built and sent <code>IX_CC_ERROR_OOM_SYSTEM</code>—operation failed, not enough memory to send a message <code>IX_CC_ERROR_NULL</code>—operation failed, NULL pointer in input parameters <code>IX_CC_ERROR_SEND_FAIL</code>—operation failed, failure at sending a message
--------------	--

19.3.2.1 `ix_cc_classifier_6t_cb_remove_entry`

The function prototype for message-handler callback function provided by the calling application to return an operation status.

C Syntax

```
ix_error (*ix_cc_classifier_6t_cb_remove_entry) (
    ix_error arg_Result,
    void* arg_pContext);
```

Input

<code>arg_pContext</code>	Pointer to client-defined context, provided as a parameter in the API call. Used by the calling application to match asynchronous request with a response.
---------------------------	--

Output/Returns

<code>arg_Result</code>	Indicates error conditions for the call with the following returns:
Return Value	<ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded, entry successfully removed <code>IX_CC_ERROR_ENTRY_NOT_FOUND</code>—operation failed, a classification pattern does not exist

19.3.3 `ix_cc_classifier_6t_async_update_entry()`

This message helper function builds and sends an entry update message to the classifier core component. The core component uses the classification pattern, `arg_pKey`, to prepare a hash input, then employs a hardware hash unit to generate a hash table index and performs the following tasks:

- reports an error, if the hash lookup fails (classification pattern is not configured).
- updates result fields in the hash table entry, on successful lookup.

C Syntax

```
ix_error ix_cc_classifier_6t_async_update_entry (
```

```
ix_cc_classifier_6t_pattern *arg_pKey,
ix_cc_classifier_6t_result *arg_pLookupResult,
ix_cc_classifier_6t_cb_update_entry arg_Callback,
void* arg_pUserContext);
```

Input

<code>arg_pKey</code>	Pointer to the classification pattern structure, which defines an exact-match rule. The structure is defined in Section 19.1.1, “Classification Pattern Data Type” on page 387.
<code>arg_pLookupResult</code>	Pointer to a placeholder for the lookup results associated with the specified classification pattern. The calling application sets type field to <code>IX_CC_CLASSIFIER_6T_QOS</code> or <code>IX_CC_CLASSIFIER_6T_FWD</code> . The core component fills in the remaining fields with lookup result as defined in Section 19.1.2, “Classification Result Data Type” on page 388.
<code>arg_Callback</code>	A calling application-provided callback function. The core component invokes this function to report an operation status. See ix_cc_classifier_6t_cb_update_entry
<code>arg_pUserContext</code>	Pointer to a calling application-defined context. This value is opaque to the core component. The context is returned in a callback function.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded, message was successfully built and sent <code>IX_CC_ERROR_OOM_SYSTEM</code>—operation failed, not enough memory to send a message <code>IX_CC_ERROR_NULL</code>—operation failed, NULL pointer in input parameters <code>IX_CC_ERROR_SEND_FAIL</code>—operation failed, failure at sending a message
--------------	--

19.3.3.1 ix_cc_classifier_6t_cb_update_entry

The function prototype for message-handler callback function provided by the calling application to return an operation status.

C Syntax

```
ix_error (*ix_cc_classifier_6t_cb_update_entry) (
    ix_error arg_Result,
    void* arg_pContext);
```

Input

<code>arg_pContext</code>	Pointer to client-defined context, provided as a parameter in the API call. Used by the calling application to match asynchronous request with a response.
---------------------------	--

Output/Returns

<code>arg_Result</code>	Indicates error conditions for the call with the following returns:
Return Value	<ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded, entry successfully updated • <code>IX_CC_ERROR_ENTRY_NOT_FOUND</code>—operation failed, an entry for the lookup pattern not configured • <code>IX_CC_CLASSIFIER_6T_ERROR_TIME_OUT</code>—operation failed, timeout in communication with the hash unit • <code>IX_CC_CLASSIFIER_6T_ERROR_PARAM</code>—operation failed, invalid parameter value specified

19.3.4 `ix_cc_classifier_6t_async_search_entry()`

This message helper function builds and sends an entry search message to the classifier core component. The core component uses the classification pattern, `arg_pKey`, to prepare a hash input, then employs a hardware hash unit to generate a hash table index, and performs the following tasks:

- a lookup result associated with the classification pattern is returned, if the requested classification pattern is found.
- returns error code and does not change the `arg_pLookupResult` parameter, if a hash table does not contain the classification pattern.

C Syntax

```
ix_error ix_cc_classifier_6t_async_search_entry (
    ix_cc_classifier_6t_pattern *arg_pKey,
    ix_cc_classifier_6t_result *arg_pLookupResult,
    ix_cc_classifier_6t_cb_search_entry arg_Callback,
    void* arg_pUserContext);
```

Input

<code>arg_pKey</code>	Pointer to the classification pattern structure, which defines an exact-match rule. The structure is defined in Section 19.1.1, “Classification Pattern Data Type” on page 387.
<code>arg_pLookupResult</code>	Pointer to a placeholder for the lookup results associated with the specified classification pattern. The calling application sets type field to <code>IX_CC_CLASSIFIER_6T_QOS</code> or <code>IX_CC_CLASSIFIER_6T_FWD</code> . The core component fills in the remaining fields with lookup result as defined in Section 19.1.2, “Classification Result Data Type” on page 388.
<code>arg_Callback</code>	A calling application-provided callback function. The core component invokes this function to report an operation status. See ix_cc_classifier_6t_cb_search_entry
<code>arg_pUserContext</code>	Pointer to a calling application-defined context. This value is opaque to the core component. The context is returned in a callback function.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded, message was successfully built and sent <code>IX_CC_ERROR_OOM_SYSTEM</code>—operation failed, not enough memory to send a message <code>IX_CC_ERROR_NULL</code>—operation failed, NULL pointer in input parameters <code>IX_CC_ERROR_SEND_FAIL</code>—operation failed, failure at sending a message
--------------	--

19.3.4.1 `ix_cc_classifier_6t_cb_search_entry`

The function prototype for message-handler callback function provided by the calling application to return an operation status.

C Syntax

```
ix_error (*ix_cc_classifier_6t_cb_search_entry)(
    ix_error arg_Result,
    ix_cc_classifier_6t_result *arg_pLookupResult,
    void* arg_pContext);
```

Input

<code>arg_pContext</code>	Pointer to client-defined context, provided as a parameter in the API call. Used by the calling application to match asynchronous request with a response.
<code>arg_pLookupResult</code>	The results structure is defined in section Section 19.1.2, “Classification Result Data Type” on page 388 . If the error code is <code>IX_SUCCESS</code> , the argument contains lookup results associated with the requested classification pattern. Otherwise, the structure is not changed.

Output/Returns

<code>arg_Result</code>	Indicates error conditions for the call with the following returns:
Return Value	<ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded, entry found <code>IX_CC_ERROR_ENTRY_NOT_FOUND</code>—operation failed, a classification pattern is not configured

19.3.5 ix_cc_classifier_6t_async_get_statistics()

This helper function builds and sends an get statistics message to the classifier core component. The core component uses the classification pattern, `arg_pKey`, to prepare a hash input. Next, it employs a hardware hash unit to generate a hash table index.

If the requested classification pattern is found and the microblock supports statistics gathering, statistics associated with this pattern are returned. Otherwise, the core component returns error code.

C Syntax

```
ix_error ix_cc_classifier_6t_async_get_statistics(
    ix_cc_classifier_6t_pattern *arg_pKey,
    ix_cc_classifier_6t_statistics *arg_pStatistics,
    ix_cc_classifier_6t_cb_get_statistics arg_Callback,
    void* arg_pUserContext);
```

Input

<code>arg_pKey</code>	Pointer to the classification pattern structure, which defines an exact-match rule. The structure is defined in section 6.3.5.2.1.
<code>arg_pStatistics</code>	Pointer to a placeholder for the statistics values associated with the specified classification pattern.
<code>arg_Callback</code>	Pointer to the calling application-provided callback function. See ix_cc_classifier_6t_cb_get_statistics . The core component invokes this function to report an operation status.
<code>arg_pUserContext</code>	Pointer to the calling application-defined context. This value is opaque to the core component. The context is returned in a callback function.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded, message was successfully built and sent <code>IX_CC_ERROR_OOM_SYSTEM</code>—operation failed, not enough memory to send a message <code>IX_CC_ERROR_NULL</code>—operation failed, NULL pointer in input parameters <code>IX_CC_ERROR_SEND_FAIL</code>—operation failed, failure at sending a message
--------------	--

19.3.5.1 ix_cc_classifier_6t_cb_get_statistics

The function prototype for message-handler callback function provided by the calling application to return an operation status.

C Syntax

```
ix_error (*ix_cc_classifier_6t_cb_get_statistics) (
    ix_error arg_Result,
    ix_cc_classifier_6t_statistics *arg_pStatistics,

    void* arg_pContext);
```

Input

arg_pStatistics	If the error code is IX_SUCCESS, the argument contains statistics associated with the requested classification pattern. Otherwise, the structure is not changed. The statistics structure is defined in Section 19.1.2, “Classification Result Data Type” on page 388 .
arg_pContext	Pointer to client-defined context, provided as a parameter in the API call. Used by the caller to match asynchronous request with a response.

Output/Returns

arg_Result	Indicates error conditions for the call with the following returns:
Return Value	<ul style="list-style-type: none"> IX_SUCCESS—the operation succeeded, entry found IX_CC_ERROR_ENTRY_NOT_FOUND—operation failed, a classification pattern is not configured IX_CC_CLASSIFIER_6T_ERROR_STATS_NOT_SUPPORTED—operation failed, the block does not support statistics

19.3.6 ix_cc_classifier_6t_async_get_statistics_def()

This helper function builds and sends an get statistics def message to the classifier core component. If the microblock supports statistics gathering, statistics associated with default pattern are returned. Otherwise, the core component returns error code.

C Syntax

```
ix_error ix_cc_classifier_6t_async_get_statistics_def (
    ix_cc_classifier_6t_statistics *arg_pStatistics,
    ix_cc_classifier_6t_cb_get_statistics_def arg_Callback,
    void* arg_pUserContext);
```

Input

arg_pStatistics	Pointer to a placeholder for the statistics values associated with the default rule.
arg_Callback	A calling application-provided callback function. See ix_cc_classifier_6t_cb_get_statistics_def .
arg_pUserContext	Pointer to calling application-defined context. This value is opaque to the core component. The context is returned in a callback function.

Output/Returns

Return Value	Returns a valid ix_error. <ul style="list-style-type: none"> IX_SUCCESS—the operation succeeded, message was successfully built and sent IX_CC_ERROR_OOM_SYSTEM—operation failed, not enough memory to send a message IX_CC_ERROR_NULL—operation failed, NULL pointer in input parameters IX_CC_ERROR_SEND_FAIL—operation failed, failure sending a message
--------------	---

19.3.6.1 ix_cc_classifier_6t_cb_get_statistics_def

The function prototype for message-handler callback function provided by the calling application to return an operation status.

C Syntax

```
ix_error (*ix_cc_classifier_6t_cb_get_statistics_def) (
    ix_error arg_Result,
    ix_cc_classifier_6t_statistics *arg_pStatistics,
    void* arg_pContext);
```

Input

<code>arg_pStatistics</code>	If the error code is <code>IX_SUCCESS</code> , the argument contains statistics associated with the default rule. Otherwise, the structure is not changed. The statistics structure is defined in Section 19.1.2, “Classification Result Data Type” on page 388.
<code>arg_pContext</code>	Pointer to calling application-defined context, provided as a parameter in the API call. Used by the calling application to match asynchronous request with a response.

Output/Returns

<code>arg_Result</code>	Indicates error conditions for the call with the following returns:
Return Value	<ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded, entry found <code>IX_CC_CLASSIFIER_6T_ERROR_STATS_NOT_SUPPORTED</code>—operation failed, the block does not support statistics <code>6T_ERROR_STATS_NOT_SUPPORTED</code>—operation failed, the block does not support statistics

19.4 Library API

The Library API provides direct C-style calls to the classifier core component. [Table 19-6](#) lists the 6-tuple core component library functions.

Table 19-6. 6-tuple Core Component Library API

API Function	Description
<code>ix_cc_classifier_6t_add_entry()</code>	Adds a new classification pattern and the associated lookup results to the 6-tuple hash table
<code>ix_cc_classifier_6t_remove_entry()</code>	Removes a classification pattern and lookup result from the 6-tuple hash table
<code>ix_cc_classifier_6t_update_entry()</code>	Updates selected fields of the lookup result associated with a specified classification pattern.
<code>ix_cc_classifier_6t_search_entry()</code>	Returns a lookup result associated with a specified classification pattern.
<code>ix_cc_classifier_6t_get_statistics()</code>	Returns statistics associated with a specified classification pattern.
<code>ix_cc_classifier_6t_get_statistics_def()</code>	Returns statistics associated with the default rule.

The library APIs correspond one-to-one with message helper APIs. The difference is that the library functions are executed in a caller context. In addition, the Library API is synchronous and there are no callback functions to report an operation result.

19.4.1 ix_cc_classifier_6t_add_entry()

This function adds a new entry in a hash table. The core component uses the classification pattern, `arg_pKey`, to prepare a hash input, then employs a hardware hash unit to generate a hash table index and performs the following tasks:

- reports an error and does not update lookup results associated with the existing rule, if the hash table already contains a specified classification pattern for a given table type.
- adds a new entry to the auxiliary hash table and appends the new entry to the chain associated with the hash value, if the hash table contains an entry for the hash value, but the entry pattern does match the pattern specified in `arg_pKey`.
- adds a new entry to the hash table, if a hash lookup fails. The lookup result, `arg_pLookupResult`, is stored in the newly added entry.

If the result type is `IX_CC_CLASSIFIER_6T_FWD` and qos information for the entry is not defined, the core component sets `flow_id` field to `0xFFFFFFFF` to indicate that the QoS information in the entry is invalid. The core must also set the `next_block_id` field to `CLASSIFIER_6T_FORWARD`.

If the result type is `IX_CC_CLASSIFIER_6T_QOS` and forwarding information for the entry is not defined, the core component sets `next_hop_id` field to `0xFFFF` to indicate that the forwarding information in the entry is invalid.

C Syntax

```
ix_error ix_cc_classifier_6t_add_entry (
    ix_cc_classifier_6t_pattern *arg_pKey,
    ix_cc_classifier_6t_result *arg_pLookupResult,
    void* arg_pContext);
```

Input

<code>arg_pKey</code>	Pointer to the classification pattern structure, which defines an exact-match rule. The structure is defined in section 5.3.5.2.1.
<code>arg_pLookupResult</code>	Pointer to the structure of lookup results associated with the specified classification pattern. The type field identifies a logical lookup table; it should be <code>IX_CC_CLASSIFIER_6T_QOS</code> or <code>IX_CC_CLASSIFIER_6T_FWD</code> . The remaining fields specify lookup result, as defined in Section 19.1.2 , “Classification Result Data Type” on page 388.
<code>arg_pContext</code>	Pointer to the control block allocated in <code>ix_cc_classifier_6t_init()</code> .

Output/Returns

Return Value	<p>Returns a valid <code>ix_error</code>.</p> <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded, entry successfully added • <code>IX_CC_ERROR_NULL</code>—operation failed, <code>arg_pKey</code>, <code>arg_pLookupResult</code> or <code>arg_pContext</code> argument is NULL • <code>IX_CC_ERROR_DUPLICATE_ENTRY</code>—operation failed, an entry for the key already configured • <code>IX_CC_CLASSIFIER_6T_ERROR_TIME_OUT</code>—operation failed, timeout in communication with the hash unit • <code>IX_CC_CLASSIFIER_6T_ERROR_PARAM</code>—operation failed, invalid parameter value specified
--------------	---

19.4.2 `ix_cc_classifier_6t_remove_entry()`

This function removes an entry from the hash table. The core component uses the classification pattern, `arg_pKey`, to prepare a hash input then employs a hardware hash unit to generate a hash table index and performs the following tasks:

- removes an entry, if the hash table already contains a specified classification pattern.
- reports an error and does not remove any entry from the hash table, if a hash lookup fails.

If QoS information is removed, the function sets `flow_id` field in the hash entry to `0xFFFFFFFF`. It must also set the `next_block_id` field to `CLASSIFIER_6T_FORWARD` to skip the metering and marking stage. Note that for performance reasons the classifier microblock does not check the `flow_id` field against the special value. It behaves as if the entry contained valid QoS data. Setting the `next_block_id` field to `CLASSIFIER_6T_FORWARD` ensures that the invalid data are not used.

If forwarding information is removed, the function sets `next_hop_id` field in the hash entry to `0xFFFF`. Note that for performance reasons the classifier microblock does not check the `next_hop_id` field against the special value. It behaves as if the entry contained valid forwarding data. It is assumed that the IPv4 forwarder performs the check.

If the entry lacks both pieces of information, the function must remove the whole entry. In this case, there are two possibilities, either the entry is in a collision chain or it is located directly in the hash table. If the entry is located in a collision chain, the function removes the entry from the chain. If the entry is located in the hash table, the function sets `IPv4_src_address` and `IPv4_dst_address` fields to `0.0.0.0` value. This ensures that the packets whose hash value index points to the entry does not match the entry and they are classified to the default traffic class.

C Syntax

```
ix_error ix_cc_classifier_6t_remove_entry (
    ix_cc_classifier_6t_pattern *arg_pKey,
    ix_cc_classifier_6t_table_type arg_Type,
    void* arg_pContext);
```


:

Input

<code>arg_pKey</code>	Pointer to the classification pattern structure, which defines an exact-match rule. The structure is defined in Section 19.1.1, “Classification Pattern Data Type” on page 387.
<code>arg_Type</code>	Identifier of a logical lookup table; it should be <code>IX_CC_CLASSIFIER_6T_QOS</code> or <code>IX_CC_CLASSIFIER_6T_FWD</code>
<code>arg_pContext</code>	Pointer to the control block allocated in <code>ix_cc_classifier_6t_init()</code> .

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded, entry successfully removed <code>IX_CC_ERROR_ENTRY_NOT_FOUND</code>—operation failed, a classification pattern does not exist <code>IX_CC_ERROR_NULL</code>—operation failed, <code>arg_pKey</code> or <code>arg_pContext</code> is <code>NULL</code>
--------------	--

19.4.3 `ix_cc_classifier_6t_update_entry()`

This function updates lookup results in a hash table entry. The core component uses the classification pattern, `arg_pKey`, to prepare a hash input, then employs a hardware hash unit to generate a hash table index, and performs the following tasks:

- reports an error, if the hash lookup fails (classification pattern is not configured).
- updates result fields in the hash table entry, on successful lookup.

C Syntax

```
ix_error ix_cc_classifier_6t_update_entry (
    ix_cc_classifier_6t_pattern *arg_pKey,
    ix_cc_classifier_6t_result *arg_pLookupResult,
    void* arg_pContext);
```

Input

<code>arg_pKey</code>	Pointer to the classification pattern structure, which defines an exact-match rule. The structure is defined in Section 19.1.1, “Classification Pattern Data Type” on page 387.
<code>arg_pLookupResult</code>	Pointer to the structure of lookup results associated with the specified classification pattern. The type field identifies a logical lookup table; it should be <code>IX_CC_CLASSIFIER_6T_QOS</code> or <code>IX_CC_CLASSIFIER_6T_FWD</code> . The remaining fields specify lookup result, as defined in Section 19.1.2, “Classification Result Data Type” on page 388.
<code>arg_pContext</code>	Pointer to the control block allocated in <code>ix_cc_classifier_6t_init()</code> .

Output/Returns

Return Value	<p>Returns a valid <code>ix_error</code>.</p> <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded, entry successfully updated • <code>IX_CC_ERROR_ENTRY_NOT_FOUND</code>—operation failed, a classification pattern does not exist • <code>IX_CC_ERROR_NULL</code>—operation failed, <code>arg_pKey</code>, <code>arg_pLookupResult</code> or <code>arg_pContext</code> argument is NULL • <code>IX_CC_CLASSIFIER_6T_ERROR_TIME_OUT</code>—operation failed, timeout in communication with the hash unit • <code>IX_CC_CLASSIFIER_6T_ERROR_PARAM</code>—operation failed, invalid parameter value specified
--------------	---

19.4.4 `ix_cc_classifier_6t_search_entry()`

The function does a lookup in the hash table. If the requested classification pattern is found, a lookup result associated with this pattern is returned. Otherwise, if a hash table does not contain the classification pattern, the core component returns `IX_CC_ERROR_ENTRY_NOT_FOUND` error code. In this case, the `arg_pLookupResult` argument is left unaltered.

C Syntax

```
ix_error ix_cc_classifier_6t_search_entry (
    ix_cc_classifier_6t_pattern *arg_pKey,
    ix_cc_classifier_6t_result *arg_pLookupResult,
    void* arg_pContext);
```

Input

<code>arg_pKey</code>	Pointer to the classification pattern structure, which defines an exact-match rule. The structure is defined in Section 19.1.1, “Classification Pattern Data Type” on page 387.
<code>arg_pContext</code>	Pointer to the control block allocated in <code>ix_cc_classifier_6t_init()</code> .

Input/Output

<code>arg_pLookupResult</code>	Pointer to a placeholder for the lookup results associated with the specified classification pattern. The caller sets type field to <code>IX_CC_CLASSIFIER_6T_QOS</code> or <code>IX_CC_CLASSIFIER_6T_FWD</code> . The core component fills in the remaining fields with lookup result as defined in Section 19.1.2, “Classification Result Data Type” on page 388.
--------------------------------	---

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded, lookup successful <code>IX_CC_ERROR_ENTRY_NOT_FOUND</code>—operation failed, no entry found for the classification pattern <code>IX_CC_ERROR_NULL</code>—operation failed, <code>arg_pKey</code>, <code>arg_pLookupResult</code> or <code>arg_pContext</code> argument is NULL
--------------	---

19.4.5 `ix_cc_classifier_6t_get_statistics()`

The function does a lookup in the hash table. If the microblock supports statistics and the requested classification pattern is found, statistics associated with this pattern are returned. Otherwise, the core component returns an error code and the `arg_pStatistics` argument is left unaltered.

The function uses SRAM Read operations to get 64-bit counter values, because the operation does not change the statistics and does not have to be synchronized with statistics updates. This means that the core component can retrieve a counter value in the middle of a read-modify-write atomic operation performed by the microblock. In such case, the SRAM controller returns the counter value before the atomic operation.

Syntax C

```
ix_error ix_cc_classifier_6t_get_statistics (
    ix_cc_classifier_6t_pattern *arg_pKey,
    ix_cc_classifier_6t_statistics *arg_pStatistics,
    void* arg_pContext);
```

Input

<code>arg_pKey</code>	Pointer to the classification pattern structure, which defines an exact-match rule. The structure is defined in Section 19.1.1, “Classification Pattern Data Type” on page 387.
<code>arg_pContext</code>	Pointer to the control block allocated in <code>ix_cc_classifier_6t_init()</code> .

Input/Output

<code>arg_pStatistics</code>	Pointer to a placeholder for the statistics associated with the specified classification pattern. The structure is defined in Section 19.1.2, “Classification Result Data Type” on page 388.
------------------------------	--

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded, lookup successful <code>IX_CC_ERROR_ENTRY_NOT_FOUND</code>—operation failed, no entry found for the classification pattern <code>IX_CC_ERROR_NULL</code>—operation failed, <code>arg_pKey</code>, <code>arg_pStatistics</code> or <code>arg_pContext</code> argument is NULL
--------------	---

19.4.6 `ix_cc_classifier_6t_get_statistics_def()`

If the microblock supports statistics, the function returns statistics associated with the default rule. Otherwise, the core component returns an error code and the `arg_pStatistics` argument is left unaltered.

The function uses SRAM Read operation to retrieve the statistics from the SRAM statistics table.

C Syntax

```
ix_error ix_cc_classifier_6t_get_statistics_def (
    ix_cc_classifier_6t_statistics *arg_pStatistics,
    void* arg_pContext);
```

Input

<code>arg_pContext</code>	Pointer to the control block allocated in <code>ix_cc_classifier_6t_init()</code> .
---------------------------	---

Input/Output

<code>arg_pStatistics</code>	Pointer to a placeholder for the statistics associated with the specified classification pattern. The structure is defined in section 6.3.5.2.2.
------------------------------	--

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded, lookup successful <code>IX_CC_ERROR_NULL</code>—operation failed, <code>arg_pStatistics</code> or <code>arg_pContext</code> argument is NULL <code>IX_CC_CLASSIFIER_6T_ERROR_STATS_NOT_SUPPORTED</code>—operation failed, the microblock does not support statistics gathering
--------------	---

The core component provides the following functionality:

- Initializes and configures the Rate Three Color Meter (SRTCM) microblock by patching symbols.
- Provides an API interface to add, remove and update TCM instances. The instance parameters are stored in a TCM meter table shared between the core component and the microblock.

For complete details see [Chapter 57, “Three Color Meter Core Component”](#) of the *Intel® Internet Exchange Architecture Software Building Blocks Developer’s Manual*.

20.1 Data Structures

This section describes data structures used in calls to the SRTCM functional APIs—Message Helper API and Library API. [Table 20-1](#) lists the two TCM core component data structures.

Table 20-1. SRTCM Core Component Data Structures

Data Structure	Description
ix_cc_tc_meter_parameters	A set of configuration parameters for a meter instance.
ix_cc_tc_meter_statistics	A collection of statistics counters associated with a meter instance.

20.1.1 TCM Parameters Data Type

This data type defines outputs from the microblock

20.1.1.1 [ix_cc_tc_meter_output](#) Enumerration

C Syntax

```
typedef enum {  
    IX_CC_TC_METER_NEXT1, /* TCM_NEXT1 output */  
    IX_CC_TC_METER_NEXT2, /* TCM_NEXT2 output */  
    IX_CC_TC_METER_NEXT3, /* TCM_NEXT3 output */  
} ix_cc_tc_meter_output;
```

20.1.1.2 ix_cc_tc_meter_algo_type Enumeration

This data type defines metering algorithm for a meter instance.

C Syntax

```
typedef enum {
    IX_CC_TC_METER_SRTCM, /* Single Rate Three Color Meter */
    IX_CC_TC_METER_TRTCM /* Two Rate Three Color Meter */
} ix_cc_tc_meter_algo_type;
```

20.1.1.3 ix_cc_tc_meter_parameters

This data type defines a structure of externally visible parameters, configurable for a meter instance:

C Syntax

```
typedef structure ix_s_cc_tc_meter_parameters
{
    ix_cc_tc_meter_algo_type algorithm; /* type of metering algorithm */
    ix_uint32 cir; /* Committed Information Rate in kbits/sec */
    ix_uint32 pir; /* Peak Information Rate in kbits/sec
                   (used only if the algorithm is TRTCM) */
    ix_uint32 cbs; /* Committed Burst Size in bytes */
    ix_uint32 ebs; /* Excess Burst Size or Peak Burst Size in bytes */
    ix_uint8 greenDscp; /* DSCP value to mark for green packets */
    ix_cc_tc_meter_output greenOutput; /* next block for green packets */
    ix_uint8 redDscp; /* DSCP value to mark for yellow packets */
    ix_cc_tc_meter_output yellowOutput; /* next block for yellow packets */
    ix_uint8 redDscp; /* DSCP value to mark for green packets */
    ix_cc_tc_meter_output redOutput; /* next block for red packets */
    ix_uint8 statsEnabled; /* if true, statistics are gathered
                           (boolean value) */
} ix_cc_tc_meter_parameters;
```

Note: The parameters are invalid if one or more of the conditions are detected:

- CIR field is zero
- PIR < CIR and algorithm = IX_CC_TC_METER_TRTCM

20.1.2 TCM Statistics Data Type

This data type defines a structure of statistic counters associated with a meter instance.

20.1.2.1 ix_cc_tc_meter_statistics

C Syntax

```
typedef structure ix_s_cc_tc_meter_statistics
{
    ix_uint64 greenPacketCount; /* number of green packets metered */
    ix_uint64 greenByteCount; /* number of green bytes metered */
    ix_uint64 yellowPacketCount; /* number of yellow packets metered */
    ix_uint64 yellowByteCount; /* number of yellow bytes metered */
    ix_uint64 redPacketCount; /* number of red packets metered */
    ix_uint64 redByteCount; /* number of red bytes metered */
} ix_cc_tc_meter_statistics;
```

20.2 Core Component Infrastructure API

Table 20-2 lists the Core Component Infrastructure APIs supported by the SRTCM core component.

Table 20-2. SRTCM Core Component Infrastructure APIs

API Function	Description
<code>ix_cc_tc_meter_init()</code>	Initializes the core component
<code>ix_cc_tc_meter_fini()</code>	Terminates the core component
<code>ix_cc_tc_meter_pkt_handler()</code>	Messages handler for processing add/update/remove requests
<code>ix_cc_tc_meter_msg_handler()</code>	Packets handler for processing received packets

20.2.1 ix_cc_tc_meter_init()

The function initializes the core component. It should be called and returned successfully before any other function in the core component can be called. The function:

- Allocates SRAM memory for the SRTCM meter table, and marks all the entries as empty. The function calculates the amount of allocated memory using system repository properties. The function must clear all the entries to mark them empty.
- Allocates SRAM memory for the 64-bit statistics table, and initializes it with all zeros. The function calculates the amount of allocated memory using system repository properties. The function must clear all the entries.
- Patches SRTCM microblock with imported variables. The variables that are static configuration data are read from the system repository properties.
- Registers a packet handler `ix_cc_tc_meter_pkt_handler()`.
- Registers a message handler `ix_cc_tc_meter_msg_handler()`.

- Allocates a control block, and returns a pointer to this block in `arg_ppContext`.

C Syntax

```
ix_error ix_cc_tc_meter_init(
ix_cc_handle arg_CcHandle,
void **arg_ppContext);
```

Input

`arg_CcHandle` Handle to the core component.

Output/Returns

`arg_ppContext` A placeholder where the pointer to the control block allocated by the core component is be stored. The core component is a passive library, invoked by a Core Component Infrastructure. The core component uses a control block to store internal variables, such as SRAM base addresses, used in library calls. The Core Component Infrastructure passes this pointer when it invokes message/packet handlers. The pointer is also passed to [ix_cc_tc_meter_fini\(\)](#) function to free memory when the core component is terminated.

Return Value Returns a valid `ix_error`.

- `IX_SUCCESS`—the operation succeeded, initialization completed successfully
- `IX_CC_ERROR_NULL`—operation failed, NULL parameter used (either `arg_CcHandle` or `arg_ppContext`)
- `IX_CC_ERROR_OOM_SYSTEM`—operation failed, local memory allocation failure
- `IX_CC_ERROR_INTERNAL`—operation failed, error in reading static configuration
- `IX_CC_ERROR_OOM_SRAM`—operation failed, SRAM memory allocation failure
- `IX_CC_ERROR_INTERNAL`—operation failed, packet or message handler registration failure
- `IX_CC_TC_METER_ERROR_FAILED_PATCHING`—operation failed, symbol patching procedure failure
- `IX_CC_TC_METER_ERROR_CCI`—operation failed, internal Core Component Infrastructure error (only if level 1 used)

20.2.2 [ix_cc_tc_meter_fini\(\)](#)

The function terminates the SRTCM core component. It is executed when the execution engine running the core component is being shut down. This function:

- Frees SRAM memory allocated by the [ix_cc_tc_meter_init\(\)](#).
- Removes a packet handler [ix_cc_tc_meter_pkt_handler\(\)](#).
- Removes a message handler [ix_cc_tc_meter_msg_handler\(\)](#).
- Frees an internal control block.

C Syntax

```
ix_error ix_cc_tc_meter_fini (
    ix_cc_handle arg_CcHandle,
    void *arg_pContext);
```

Input

<code>arg_CcHandle</code>	Handle to the core component.
<code>arg_pContext</code>	Pointer to the control block that was allocated in <code>ix_cc_tc_meter_init()</code> .

Output/Returns

Return Value	<p>Returns a valid <code>ix_error</code>.</p> <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded, shutdown completed successfully • <code>IX_CC_ERROR_NULL</code>—operation failed, NULL parameter used (either <code>arg_CcHandle</code> or <code>arg_ppContext</code>) • <code>IX_CC_ERROR_INTERNAL</code>—operation failed, message or packet handler unregistration failure • <code>IX_CC_ERROR_OOM_SRAM</code>—operation failed, de-allocation of SRAM memory failed • <code>IX_CC_TC_METER_ERROR_CCI</code>—operation failed, message or packet handler unregistration failure • <code>IX_CC_TC_METER_ERROR_INVALID_HANDLE</code>—operation failed, the core component's handle (<code>arg_CcHandle</code>) does not match the control block (<code>arg_pContext</code>)
--------------	--

20.2.3 `ix_cc_tc_meter_pkt_handler()`

The core component uses one packet handler for processing all packets. When invoked, the handler redirects a packet to the SRTCM microblock. The core component does not execute the metering algorithm in order to avoid synchronization with the microblock.

C Syntax

```
ix_error ix_cc_tc_meter_pkt_handler (
    ix_buffer_handle arg_Pkt,
    ix_uint32 arg_ExceptionCode,
    void *arg_pContext);
```

Input

<code>arg_Pk</code>	Buffer handle to the received packet.
<code>arg_ExceptionCode</code>	An exception code thrown by a microblock; not used by the SRTCM core component.
<code>arg_pContext</code>	Pointer to the control block that was allocated in <code>ix_cc_tc_meter_init()</code> .

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded, packet processed successfully <code>IX_CCI_ERR_SEND_FAIL</code>—operation failed, failure to send the packet to the microblock
--------------	--

20.2.4 `ix_cc_tc_meter_msg_handler()`

The core component uses one message handler for processing all types of messages. The handler receives messages from a single source—DiffServ application.

A set of supported messages covers managing SRTCM instances and configuring their parameters as specified in [RFC2697]. Internally, the handler function parses message payloads and calls appropriate Library API functions to perform the requested operation.

C Syntax

```
ix_error ix_cc_tc_meter_msg_handler (
    ix_buffer_handle arg_hDataToken,
    ix_uint32 arg_UserData,
    void *arg_pContext);
```

Input

<code>arg_hDataToken</code>	Buffer handle embedding information for the message passed in <code>arg_UserData</code> .
<code>arg_UserData</code>	Message type. Table 20-3 lists the supported messages:
<code>arg_pContext</code>	Pointer to a control block that was allocated in <code>ix_cc_tc_meter_init()</code> .

Table 20-3. SRTCM Message Types supported

Message type	Description
IX_CC_TC_METER_MSG_ADD_ENTRY	Add a new entry (instance) to the SRTCM meter table.
IX_CC_TC_METER_MSG_REMOVE_ENTRY	Delete an entry (instance) from the SRTCM meter table.
IX_CC_TC_METER_MSG_UPDATE_ENTRY	Update an entry (instance) in the SRTCM meter table.
IX_CC_TC_METER_MSG_GET_STATISTICS	Return 64-bit statistics associated with the selected meter instance.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> IX_SUCCESS—the operation succeeded, message processed successfully IX_CC_ERROR_NULL—operation failed, <code>arg_pContext</code> or buffer handle is NULL IX_CC_ERROR_UNDEFINED_MSG—operation failed, invalid message type received IX_CC_ERROR_INTERNAL—operation failed, function was unable to free the buffer IX_CC_ERROR_SEND_FAIL—operation failed, reply message sending error
--------------	--

20.3 Message Helper API

The Message Helper is a wrapper library that facilitates sending messages to the SRTCM core component. There is one-to-one mapping between Message Helper APIs and message types supported by a core component. All message helper APIs are asynchronous—a core component reports an operation status in a callback routine. Table 20-4 lists the SRTCM message helper APIs.

Table 20-4. SRTCM Message Helper APIs

API Function	Description
<code>ix_cc_tc_meter_async_add_entry()</code>	Adds a new meter instance to the TCM table
<code>ix_cc_tc_meter_async_remove_entry()</code>	Removes a meter instance from the TCM table
<code>ix_cc_tc_meter_async_update_entry()</code>	Updates meter instance parameters in the TCM table
<code>ix_cc_tc_meter_async_get_statistics()</code>	Returns statistics associated with a meter instance

20.3.1 `ix_cc_tc_meter_async_add_entry()`

This message helper function builds and sends an entry add message to the SRTCM core component. The core component uses `arg_Instance` to select a row in the meter table and performs the following tasks:

- reports an error (via callback) and does not override instance parameters stored in the table, if the row is already configured.

- copies data from `arg_pInstanceParameters` to the selected entry, if the row is not configured and an entry is empty.

C Syntax

```
ix_error ix_cc_tc_meter_async_add_entry (
    ix_uint32 arg_Instance,
    ix_cc_tc_meter_parameters *arg_pInstanceParameters,
    ix_cc_tc_meter_cb_add_entry arg_Callback,
    void* arg_pUserContext);
:
```

Input

<code>arg_Instance</code>	Index of a meter table entry that should be configured with new parameters. The value is equal to <code>flowId</code> set by the classifier block for the packets processed by the meter instance.
<code>arg_pInstanceParameters</code>	Pointer to the structure of parameters defining a meter instance. The structure layout is defined in section Section 20.1.1, “TCM Parameters Data Type” on page 413.
<code>arg_Callback</code>	A calling application-provided callback function. The core component invokes this function to report an operation status. See ix_cc_tc_meter_cb_add_entry .
<code>arg_pUserContext</code>	Pointer to a calling application-defined context. This value is opaque to the core component. The context is returned in a callback function.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded, message was successfully built and sent • <code>IX_CC_ERROR_NULL</code>—operation failed, callback function or <code>arg_pInstanceParameters</code> argument is <code>NULL</code> • <code>IX_CC_ERROR_OOM_SYSTEM</code>—operation failed, the function cannot allocate memory for message context • <code>IX_CC_ERROR_TC_METER_MSG_LIBRARY</code>—operation failed, failure on sending a message
--------------	--

20.3.1.1 ix_cc_tc_meter_cb_add_entry

The function prototype for message-handler callback function provided by the calling application to return an operation status.

C Syntax

```
ix_error (*ix_cc_tc_meter_cb_add_entry) (
    ix_error arg_Result,
```

```
void* arg_pContext);
```

Input

<code>arg_pContext</code>	Pointer to client-defined context, provided as a parameter in the API call. Used by the calling application to match asynchronous request with a response.
---------------------------	--

Output/Returns

<code>arg_Result</code>	Indicates error conditions for the call with the following returns:
Return Value	<ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded, meter instance successfully configured • <code>IX_CC_ERROR_CONFLICTING_ENTRY</code>—operation failed, meter instance already configured • <code>IX_CC_ERROR_RANGE</code>—operation failed, meter instance out of range; the instance identifier is higher than the maximum number of instances that can be configured in the block • <code>IX_CC_TC_METER_ERROR_INVALID_PARAM</code>—operation failed, <code>arg_pParams</code> structure contains invalid values (for example, CIR equals 0)

20.3.2 `ix_cc_tc_meter_async_remove_entry()`

This message helper function builds and sends an entry remove message to the SRTCM core component. The core component first checks if an entry, pointed by `arg_Instance`, is valid and performs the following tasks:

- marks the entry as empty and changes the statistics value to zero, if an entry in the SRTCM table is valid
- reports an error in a callback function., if the referred entry is already marked empty.

C Syntax

```
ix_error ix_cc_tc_meter_async_remove_entry (
    ix_uint32 arg_Instance,
    ix_cc_tc_meter_cb_remove_entry arg_Callback,
    void* arg_pUserContext);
:
```

Input

<code>arg_Instance</code>	Index of a meter table entry that should be removed
<code>arg_Callback</code>	A calling application-provided callback function. The core component invokes this function to report an operation status. See ix_cc_tc_meter_cb_remove_entry .
<code>arg_pUserContext</code> <code>t</code>	Pointer to a calling application-defined context. This value is opaque to the core component. The context is returned in a callback function.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded, message was successfully built and sent • <code>IX_CC_ERROR_NULL</code>—operation failed, NULL callback argument • <code>IX_CC_ERROR_SEND_FAIL</code>—operation failed, failure on sending a message
--------------	---

20.3.2.1 `ix_cc_tc_meter_cb_remove_entry`

The function prototype for message-handler callback function provided by the calling application to return an operation status.

C Syntax

```
ix_error (*ix_cc_tc_meter_cb_remove_entry) (
    ix_error arg_Result,
    void* arg_pContext);
```

Input

<code>arg_pContext</code>	Pointer to client-defined context, provided as a parameter in the API call. Used by the calling application to match asynchronous request with a response.
---------------------------	--

Output/Returns

<code>arg_Result</code>	Indicates error conditions for the call with the following returns:
Return Value	<ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded, meter instance successfully removed • <code>IX_CC_ERROR_ENTRY_NOT_FOUND</code>—operation failed, meter instance does not exist • <code>IX_CC_ERROR_RANGE</code>—operation failed, meter instance out of range

20.3.3 `ix_cc_tc_meter_async_update_entry()`

This helper function builds and sends an entry update message to the SRTCM core component. The core component checks if an SRTCM entry, pointed by `arg_Instance`, is valid and performs the following tasks:

- updates a meter entry with values provided in `arg_pInstanceParameters`, if an SRTCM entry is valid
- reports an error and does not modify the meter parameters, If the referred entry is empty. The table entry remains empty.

C Syntax

```
ix_error ix_cc_tc_meter_async_update_entry (
    ix_uint32 arg_Instance,
    ix_cc_tc_meter_parameters *arg_pInstanceParameters,
    ix_cc_tc_meter_cb_update_entry arg_Callback,
    void* arg_pUserContext);
```

Input

<code>arg_Instance</code>	Index of a meter table entry that should be updated
<code>arg_pInstanceParameters</code>	Pointer to the structure of updated parameters for a meter instance. The structure layout is defined in Section 20.1.1, “TCM Parameters Data Type” on page 413.
<code>arg_Callback</code>	A calling application-provided callback function. The core component invokes this function to report an operation status. See ix_cc_tc_meter_cb_update_entry .
<code>arg_pUserContext</code>	Pointer to a calling application-defined context. This value is opaque to the core component. The context is returned in a callback function.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded, message was successfully built and sent <code>IX_CC_ERROR_SEND_FAIL</code>—operation failed, failure on sending a message <code>IX_CC_ERROR_NULL</code>—operation failed, callback function or <code>arg_pInstanceParameters</code> argument is NULL <code>IX_CC_ERROR_OOM_SYSTEM</code>—operation failed, the function cannot allocate memory for message context
--------------	--

20.3.3.1 `ix_cc_tc_meter_cb_update_entry`

The function prototype for message-handler callback function provided by the calling application to return an operation status.

C Syntax

```
ix_error (*ix_cc_tc_meter_cb_update_entry) (
    ix_error arg_Result,
    void* arg_pContext);
```

Input

<code>arg_pContext</code>	Pointer to client-defined context, provided as a parameter in the API call. Used by the calling application to match asynchronous request with a response.
---------------------------	--

Output/Returns

<code>arg_Result</code>	Indicates error conditions for the call with the following returns:
Return Value	<ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded, meter instance successfully updated • <code>IX_CC_ERROR_ENTRY_NOT_FOUND</code>—operation failed, meter instance is empty • <code>IX_CC_TC_METER_ERROR_INVALID_INPUT_PARAM</code>—operation failed, the meter parameters are invalid (for example, the CIR is zero) • <code>IX_CC_ERROR_RANGE</code>—operation failed, entry index is out of range; the instance identifier is higher than the maximum number of instances that can be configured in the block

20.3.4 `ix_cc_tc_meter_async_get_statistics()`

This message helper function builds and sends a statistics read message to the SRTCM core component. The core component checks if the meter entry, pointed by `arg_Instance`, is valid and performs the following tasks:

- reports an error (via a callback routine), if the referred table entry is empty.
- retrieves 64-bit statistic counters associated with this entry and returns them in a callback routine, if the referred table entry is not empty.

C Syntax

```
ix_error ix_cc_tc_meter_async_get_statistics(
    ix_uint32 arg_Instance,
    ix_cc_tc_meter_statistics * arg_pInstanceStatistics,
    ix_cc_tc_meter_cb_get_statistics arg_Callback,
    void* arg_pUserContext);
```


Input

<code>arg_Instance</code>	Index of a meter table entry that is queried for statistics
<code>arg_pInstanceStatistics</code>	Pointer to the statistics placeholder that should be filled with data by the core component. The structure layout is defined in Section 20.1.2, “TCM Statistics Data Type” on page 415.
<code>arg_Callback</code>	A calling application-provided callback function. The core component invokes this function to report an operation status and statistic counters. See ix_cc_tc_meter_cb_get_statistics .
<code>arg_pUserContext</code>	Pointer to the client-defined context, provided as a parameter in the API call. Used by the calling application to match asynchronous request with a response.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded, message was successfully built and sent • <code>IX_CC_ERROR_SEND_FAIL</code>—operation failed, failure on sending a message • <code>IX_CC_ERROR_NULL</code>—operation failed, callback function argument is NULL • <code>IX_CC_ERROR_OOM_SYSTEM</code>—operation failed, the function cannot allocate memory for message context
--------------	--

20.3.4.1 `ix_cc_tc_meter_cb_get_statistics`

The function prototype for message-handler callback function provided by the calling application to return an operation status.

C Syntax

```
ix_error (*ix_cc_tc_meter_cb_get_statistics) (
    ix_error arg_Result;
    ix_cc_tc_meter_statistics *arg_pInstanceStatistics,
    void* arg_pContext);
```

Input

<code>arg_pInstanceStatistics</code>	Pointer to the statistics structure filled with valid data if a meter entry is not empty. The structure layout is defined in Section 20.1.2, “TCM Statistics Data Type” on page 415.
<code>arg_pContext</code>	Pointer to client-defined context, provided as a parameter in the API call. Used by the calling application to match asynchronous request with a response.

Output/Returns

<code>arg_Result</code>	Indicates error conditions for the call with the following returns:
Return Value	<ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded, <code>arg_pInstanceStatistics</code> contains valid statistics <code>IX_CC_ERROR_ENTRY_NOT_FOUND</code>—operation failed, entry is empty (not configured yet) <code>IX_CC_ERROR_RANGE</code>—operation failed, entry index is out of range; the instance identifier is higher than the maximum number of instances that can be configured in the block

20.4 Library API

The Library API provides direct C-style calls to the classifier core component. [Table 20-5](#) lists the SRTCM library functions.

Table 20-5. SRTCM Library API

API Function	Description
<code>ix_cc_tc_meter_add_entry()</code>	Add a new meter instance to the TCM table
<code>ix_cc_tc_meter_remove_entry()</code>	Remove a meter instance from the TCM table
<code>ix_cc_tc_meter_update_entry()</code>	Update meter instance parameters in the TCM table
<code>ix_cc_tc_meter_get_statistics()</code>	Return statistics associated with a meter instance

The library API functions correspond one-to-one with message helper APIs. The difference is that library functions are executed in a calling application context. In addition, the Library API is synchronous and there are no callback functions to report an operation result.

20.4.1 `ix_cc_tc_meter_add_entry()`

This function adds a new entry in the SRTCM meter table. The core component uses `arg_Instance` to select a row in the meter table depending on whether the row is configured or not and performs the following tasks:

- reports an error, if the row is already configured, and does not override instance parameters stored in the table.
- copies meter instance parameters from `arg_pInstanceParameters` to the selected entry, if an entry is empty.

[Table 20-6](#) shows the values assigned by the function to the meter instance parameters.

C Syntax

```
ix_error ix_cc_tc_meter_add_entry (
    ix_uint32 arg_Instance,
    ix_cc_tc_meter_parameters *arg_pInstanceParameters,
    void* arg_pContext);
```

Input

<code>arg_Instance</code>	Index of a meter table entry that should be configured with new parameters. The argument must not be higher than the value defined in <code>\\SRTCM\TABLE_SRAM_SIZE</code> system property.
<code>arg_pInstanceParameters</code>	Pointer to the structure of parameters defining a meter instance. The structure layout is defined in Section 20.1.1, “TCM Parameters Data Type” on page 413.
<code>arg_pContext</code>	Pointer to the control block allocated in <code>ix_cc_tc_meter_init()</code> .

Output/Returns

Return Value	<p>Returns a valid <code>ix_error</code>.</p> <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded, new meter instance successfully added and configured <code>IX_CC_ERROR_CONFLICTING_ENTRY</code>—operation failed, meter instance already configured <code>IX_CC_ERROR_RANGE</code>—operation failed, meter instance out of range; the instance identifier is higher than the maximum number of instances that can be configured in the block <code>IX_CC_ERROR_NULL</code>—operation failed, <code>arg_pInstanceParameters</code> argument is NULL <code>IX_CC_TC_METER_ERROR_INVALID_PARAM</code>—operation failed, <code>arg_pInstanceParameters</code> structure contains invalid meter configuration (for example CIR equals 0)
--------------	--

Table 20-6. SRAM Entry Field Values Set by the `ix_cc_tc_meter_add` API

SRAM entry field	Value set by the core
timestamp	Zero
TC	<code>arg_pInstanceParameters->cbs</code>
TE	<code>arg_pInstanceParameters->ebs</code>
CIR	<code>arg_pInstanceParameters->cir*2^(cir_to_pir_rate_scale + pir_rate_scale)/TCM_UNIT_CHANGE_FACTOR</code>
PIR	<code>arg_pInstanceParameters->pir*2^pir_rate_scale/TCM_UNIT_CHANGE_FACTOR</code>
CBS	<code>arg_pInstanceParameters->cbs</code>
EBS / PBS	<code>arg_pInstanceParameters->ebs</code>
green_dscp (G) - bits [7..2]	<code>arg_pInstanceParameters->greenDscp[5..0]</code>

Table 20-6. SRAM Entry Field Values Set by the ix_cc_tc_meter_add API

SRAM entry field	Value set by the core
green_dscp(G) - bits [1..0]	01b, if arg_pInstanceParameters->greenOutput equals to IX_CC_TC_METER_NEXT1; 10b, if arg_pInstanceParameters->greenOutput equals to IX_CC_TC_METER_NEXT2; 11b, if arg_pInstanceParameters->greenOutput equals to IX_CC_TC_METER_NEXT3;
yellow_dscp(Y) - bits [7..2]	arg_pInstanceParameters->yellowDscp[5..0]
yellow_dscp(G) - bits [1..0]	01b, if arg_pInstanceParameters->greenOutput equals to IX_CC_TC_METER_NEXT1; 10b, if arg_pInstanceParameters->yellowOutput equals to IX_CC_TC_METER_NEXT2; 11b, if arg_pInstanceParameters->yellowOutput equals to IX_CC_TC_METER_NEXT3;
red_dscp(R) - bits [7..2]	arg_pInstanceParameters->redDscp[5..0]
red_dscp(G) - bits [1..0]	01b, if arg_pInstanceParameters->greenOutput equals to IX_CC_TC_METER_NEXT1; 10b, if arg_pInstanceParameters->yellowOutput equals to IX_CC_TC_METER_NEXT2; 11b, if arg_pInstanceParameters->yellowOutput equals to IX_CC_TC_METER_NEXT3;
meter_type_flag (T)	if (arg_pInstanceParameters->algorithm == IX_CC_TC_METER_SRTCM)meter_type_flag = 0; else meter_type_flag = 1;
statistics_flag (S)	if (arg_pInstanceParameters->statsEnabled == 0)statistics_flag = 0;elsestatistics_flag = 1;
pir_rate_scale	if(arg_pInstanceParameters->algorithm == IX_CC_TC_METER_SRTCM)pir_rate_scale = 0; else pir_rate_scale = calc_rate_scale(arg_pInstanceParameters->pir);The calc_rate_scale() procedure is specified in Listing 8 1.
cir_to_pir_rate_scale	cir_rate_scale = calc_rate_scale(arg_pInstanceParameters->pir) - pir_rate_scale; The calc_rate_scale() procedure is specified in Example 20-1 .
green_pkt_count	Zero
green_byte_count	Zero
yellow_pkt_count	Zero
yellow_byte_count	Zero
red_pkt_count	Zero
red_byte_count	Zero

Note: The function zeroes the entry in the 64-bit statistics table.

Example 20-1. Procedure for calculating rate_scale value

```
int calc_rate_scale(int rate)
{
    int rate_scale = 0;
    float scaled_rate = rate/TCM_UNIT_CHANGE_FACTOR;
    while ( (scaled_rate - floor(scaled_rate)) > 0,01*scaled_rate )
    {
        scaled_rate = scaled_rate*2;
        rate_scale++;
    }
    return rate_scale;
}
```

The constant SRTCM_UNIT_CHANGE_FACTOR is equal to 292968.75.

$$\frac{6000000000 \text{Cycles/second} * 8 \text{bits/byte}}{16 \text{Cycles/timestamp} * 1024}$$

It is the factor used to change the rate in kbits/s into rate expressed in bytes/timestamp.

20.4.2 ix_cc_tc_meter_remove_entry()

This function removes an entry from the SRTCM meter table. The core component first checks if an entry, pointed by `arg_Instance`, is valid and performs the following tasks:

- changes the value of the CIR field and the statistics to zero to mark the entry as empty, and leaves the rest of the fields unchanged, if an entry in the SRTCM meter table is valid.
- Returns `IX_CC_ERROR_ENTRY_NOT_FOUND`, if an entry in the SRTCM meter table is not valid, that is, the referred entry is already marked empty.

The core component uses zero CIR value to mark the empty entries in the meter instance table. However, when the microblock processes a packet, it checks whether a packet matches an empty entry in the meter table and it does not check whether the CIR field is zero. It behaves as if the entry was not empty.

It is assumed that the application that removes a meter instance is responsible for changing the configuration of the classifier block in such way that no packets match the empty meter entries.

C Syntax

```
ix_error ix_cc_tc_meter_remove_entry (
    ix_uint32 arg_Instance,
    void* arg_pContext);
```

Input

<code>arg_Instance</code>	Index of a meter table entry that should be removed. The argument must not be higher than the value defined in <code>\\SRTCM\TABLE_SRAM_SIZE</code> system property.
<code>arg_pContext</code>	Pointer to the control block allocated in <code>ix_cc_tc_meter_init()</code> .

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded, meter instance successfully removed <code>IX_CC_ERROR_ENTRY_NOT_FOUND</code>—operation failed, meter instance does not exist <code>IX_CC_ERROR_RANGE</code>—operation failed, meter instance out of range
--------------	--

20.4.3 `ix_cc_tc_meter_update_entry()`

This function checks if a meter entry, pointed by `arg_Instance`, is valid. Depending on whether meter entry is valid and performs the following tasks:

- updates a meter entry with values provided in `arg_pInstanceParameters`, if meter entry is valid.
- reports an error if the referred entry is empty or invalid. In such case, meter parameters are not modified and the table entry remains empty.

Table 20-7 shows the values assigned by the function to the meter instance parameters.

C Syntax

```
ix_error ix_cc_tc_meter_update_entry (
    ix_uint32 arg_Instance,
    ix_cc_tc_meter_parameters *arg_pInstanceParameters,
    void* arg_pContext);
```

Input

<code>arg_Instance</code>	Index of a meter table entry that should be updated. The argument must not be higher than the value defined in <code>\\SRTCM\TABLE_SRAM_SIZE</code> system property.
<code>arg_pInstanceParameters</code>	Pointer to the structure of updated parameters for a meter instance. The structure layout is defined in Section 20.1.1, “TCM Parameters Data Type” on page 413.
<code>arg_pContext</code>	Pointer to the control block allocated in <code>ix_cc_tc_meter_init()</code>

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded, meter instance successfully updated <code>IX_CC_ERROR_ENTRY_NOT_FOUND</code>—operation failed, meter instance does not exist <code>IX_CC_ERROR_RANGE</code>—operation failed, meter instance out of range; the instance identifier is higher than the maximum number of instances that can be configured in the block <code>IX_CC_TC_METER_ERROR_INVALID_PARAM</code>—operation failed, <code>arg_pInstanceParameters</code> structure contains invalid SRTCM configuration (for example CIR equals 0) <code>IX_CC_ERROR_NULL</code>—operation failed, <code>arg_pParams</code> is NULL
--------------	--

Table 20-7. SRAM Entry Field Values Set by the `ix_cc_tc_meter_update` API

SRAM entry field	Value set by the core
timestamp	Not changed
TC	<code>arg_pInstanceParameters->cbs</code>
TE	<code>arg_pInstanceParameters->ebs</code>
CIR	<code>arg_pInstanceParameters->cir*2^(cir_to_pir_rate_scale + pir_rate_scale)/TCM_UNIT_CHANGE_FACTOR</code>
PIR	<code>arg_pInstanceParameters->pir*2^pir_rate_scale/TCM_UNIT_CHANGE_FACTOR</code>
CBS	<code>arg_pInstanceParameters->cbs</code>
EBS	<code>arg_pInstanceParameters->ebs</code>
green_dscp (G) - bits [7..2]	<code>arg_pInstanceParameters->greenDscp[5..0]</code>
green_dscp(G) - bits [1..0]	01b, if <code>arg_pInstanceParameters->greenOutput</code> equals to <code>IX_CC_TC_METER_NEXT1</code> ; 10b, if <code>arg_pInstanceParameters->greenOutput</code> equals to <code>IX_CC_TC_METER_NEXT2</code> ; 11b, if <code>arg_pInstanceParameters->greenOutput</code> equals to <code>IX_CC_TC_METER_NEXT3</code> ;
yellow_dscp (Y) - bits [7..2]	<code>arg_pInstanceParameters->yellowDscp[5..0]</code>
yellow_dscp(G) - bits [1..0]	01b, if <code>arg_pInstanceParameters->greenOutput</code> equals to <code>IX_CC_TC_METER_NEXT1</code> ; 10b, if <code>arg_pInstanceParameters->yellowOutput</code> equals to <code>IX_CC_TC_METER_NEXT2</code> ; 11b, if <code>arg_pInstanceParameters->yellowOutput</code> equals to <code>IX_CC_TC_METER_NEXT3</code> ;
red_dscp (R) - bits [7..2]	<code>arg_pInstanceParameters->redDscp[5..0]</code>
red_dscp(G) - bits [1..0]	01b, if <code>arg_pInstanceParameters->greenOutput</code> equals to <code>IX_CC_TC_METER_NEXT1</code> ; 10b, if <code>arg_pInstanceParameters->yellowOutput</code> equals to <code>IX_CC_TC_METER_NEXT2</code> ; 11b, if <code>arg_pInstanceParameters->yellowOutput</code> equals to <code>IX_CC_TC_METER_NEXT3</code> ;

Table 20-7. SRAM Entry Field Values Set by the ix_cc_tc_meter_update API

SRAM entry field	Value set by the core
meter_type_flag (T)	if (arg_pInstanceParameters->algorithm == IX_CC_TC_METER_SRTCM) meter_type_flag = 0; else meter_type_flag = 1;
statistics_flag (S)	if (arg_pInstanceParameters->statsEnabled == 0) statistics_flag = 0; else statistics_flag = 1;
pir_rate_scale	if(arg_pInstanceParameters->algorithm == IX_CC_TC_METER_SRTCM) pir_rate_scale = 0; else pir_rate_scale = calc_rate_scale(arg_pInstanceParameters->pir); The calc_rate_scale() procedure is specified in Example 20-1
cir_to_pir_rate_scale	cir_rate_scale = calc_rate_scale(arg_pInstanceParameters->pir) - pir_rate_scale; The calc_rate_scale() procedure is specified in Example 20-1 .
green_pkt_count (including the entry in 64-bit statistics table)	Zero if statistics_flag changes from 0 to 1, otherwise unchangedThe operation must be performed before the statistics_flag is written to SRAM.
green_byte_count (including the entry in 64-bit statistics table)	Zero if statistics_flag changes from 0 to 1, otherwise unchangedThe operation must be performed before the statistics_flag is written to SRAM.
yellow_pkt_count (including the counterpart in 64-bit statistics table)	Zero if statistics_flag changes from 0 to 1, otherwise unchangedThe operation must be performed before the statistics_flag is written to SRAM.
yellow_byte_count (including the counterpart in 64-bit statistics table)	Zero if statistics_flag changes from 0 to 1, otherwise unchangedThe operation must be performed before the statistics_flag is written to SRAM.
red_pkt_count (including the counterpart in 64-bit statistics table)	Zero if statistics_flag changes from 0 to 1, otherwise unchangedThe operation must be performed before the statistics_flag is written to SRAM.
red_byte_count (including the counterpart in 64-bit statistics table)	Zero if statistics_flag changes from 0 to 1, otherwise unchangedThe operation must be performed before the statistics_flag is written to SRAM.

20.4.4 ix_cc_tc_meter_get_statistics()

This function returns statistic counters associated with a meter instance. First, the core component checks if the meter entry, pointed by arg_Instance, is valid and performs the following tasks:

- reports an error (via a callback routine), if the referred table entry is empty.
- retrieves 64-bit statistic counters associated with this entry and fills in arg_pInstanceStatistics with valid data, if the referred table entry is not empty.

C Syntax

```
ix_error ix_cc_tc_meter_get_statistics (
    ix_uint32 arg_Instance,
```



```
ix_cc_tc_meter_statistics *arg_pInstanceStatistics,  
void* arg_pContext);
```

Input

<code>arg_Instance</code>	Index of a meter table entry that should be updated
<code>arg_pContext</code>	Pointer to the control block allocated in <code>ix_cc_tc_meter_init()</code>

Output/Returns

<code>arg_pInstanceStatistics</code>	Pointer to the statistics structure that should be filled with data by the core component. The structure layout is defined in Section 20.1.2 , “TCM Statistics Data Type” on page 415.
--------------------------------------	--

Return Value	<p>Returns a valid <code>ix_error</code>.</p> <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded, statistics successfully stored in a callback structure • <code>IX_CC_ERROR_ENTRY_NOT_FOUND</code>—operation failed, entry does not exist • <code>IX_CC_ERROR_RANGE</code>—operation failed, meter instance out of range; the instance identifier is higher than the maximum number of instances that can be configured in the block • <code>IX_CC_TC_METER_ERROR_INVALID_PARAM</code>—operation failed, <code>arg_pParams</code> structure contains invalid values (for example CIR equals 0) • <code>IX_CC_ERROR_NULL</code>—operation failed, <code>arg_pParams</code> is NULL
--------------	---

Weighted Random Early Detection 21

The core component provides the following functionality:

- Initializes and configures the Weighted Random Early Detection (WRED) microblock by patching symbols.
- Provides an API interface to add, remove and update WRED instances. The instance parameters are shared between the core component and the microblock.
- Provides an API interface to read statistics associated with a selected WRED instance.

For complete details see [Chapter 58, “Weighted Random Early Detection \(WRED\) Core Component”](#) of the *Intel® Internet Exchange Architecture Software Building Blocks Developer’s Manual*.

21.1 Data Structures

This section describes data structures used by the Message Helper API and Library API. The WRED core component defines three data structures.

Table 21-1. WRED Core Component Data Structures for Message Helper and Library APIs

Data Structure	Description
ix_s_cc_red_instance	A set of configuration parameters that are color-specific.
ix_s_cc_wred_parameters	A set of configuration parameters for a WRED-protected queue.
WRED Statistics Data Type	A collection of statistics counters associated with a WRED instance.

21.1.1 WRED Parameters Data Types

The two data types are used to describe WRED parameters.

21.1.1.1 [ix_s_cc_red_instance](#)

Defines a set of configuration parameters that are color-specific

C Syntax

```
typedef structure ix_s_cc_red_instance
{
    ix_uint8 min_th; /* minimum queue threshold in packets (0-255)*/
    /* no packets are dropped, if the average */
    /* queue length is below this threshold */
    ix_uint8 max_th; /* maximum queue threshold in packets (0-255)*/
    /* packets are always dropped, if the average*/
    /* queue length exceeds this threshold */
    ix_uint16 max_prob; /* dropping probability at max_th, expressed */
}
```

```

        /* as a fraction with denominator of 65535 */
    } ix_cc_red_instance;

```

21.1.1.2 ix_s_cc_wred_parameters

A WRED instance comprises three copies of RED parameters, as well as color-independent parameters. It defines a set of configuration parameters for a WRED-protected queue.

C Syntax

```

typedef structure ix_s_cc_wred_parameters
{
    ix_uint32 service_rate; /* Queue service rate in packets/sec */
    ix_uint8 weight; /* EWMA factor, negative power of 2 */
    ix_cc_red_instance green; /* RED params for green packets */
    ix_cc_red_instance yellow; /* RED params for yellow packets */
    ix_cc_red_instance red; /* RED params for red packets */
    ix_uint8 statsEnabled;
        /* boolean value; if true, statistics are gathered */
} ix_cc_wred_parameters;

```

Note: The weight field must be in range from 0 to 15 and it is forbidden to specify zero service_rate.

21.1.2 WRED Statistics Data Type

This data type defines a structure of statistic counters associated with a WRED instance.

21.1.2.1 ix_cc_wred_statistics

C Syntax

```

typedef structure ix_s_cc_wred_statistics
{
    ix_uint64 greenPacketCount; /* number of green packets dropped */
    ix_uint64 greenByteCount; /* number of green bytes dropped */
    ix_uint64 yellowPacketCount; /* number of yellow packets dropped */
    ix_uint64 yellowByteCount; /* number of yellow bytes dropped */
    ix_uint64 redPacketCount; /* number of red packets dropped */
    ix_uint64 redByteCount; /* number of red bytes dropped */
} ix_cc_wred_statistics;

```

21.2 Core Component Infrastructure API

Table 21-2 shows the WRED core component Infrastructure APIs.

Table 21-2. WRED Core Component Infrastructure API

API Function	Description
<code>ix_cc_wred_init()</code>	Initializes the core component
<code>ix_cc_wred_fini()</code>	Terminates the core component
<code>ix_cc_wred_pkt_handler()</code>	Message Handler for processing add/update/remove requests
<code>ix_cc_wred_msg_handler()</code>	Packet handler for processing received packets
<code>ix_cc_wred_timer_event_handler()</code>	Periodically calculates the dropping probability for WRED instances.

21.2.1 `ix_cc_wred_init()`

The function initializes the core component. It is called and returned successfully before any other function in the core component can be called. The function:

- Allocates SRAM memory for the WRED table, and marks all the entries as empty. The function calculates the amount of allocated memory using the system repository properties.
- Marks all the entries as empty. Table 21-3 shows the empty entry pattern.
- Allocates SRAM memory for the 64-bit statistics table, and initializes it with all zeros. The function calculates the amount of allocated memory using the system repository properties.
- Retrieves from a system registry the SRAM base address of Queue Descriptor table.
- Patches WRED microblock with imported variables. The values are retrieved from the system repository entries.
- Registers a packet handler `ix_cc_wred_pkt_handler()`.
- Registers a message handler `ix_cc_wred_msg_handler()`.
- Allocates a control block, and returns a pointer to this block in `arg_ppContext`

C Syntax

```
ix_error ix_cc_wred_init (
    ix_cc_handle arg_CcHandle,
    void **arg_ppContext);
```

Input

`arg_CcHandle` Handle to the core component.

Output/Returns

<code>arg_ppContext</code>	A placeholder where the pointer to the control block allocated by the core component is be stored. The core component is a passive library, invoked by a Core Component Infrastructure. The core component uses a control block to store internal variables, such as SRAM base addresses, used in library calls. The Core Component Infrastructure passes this pointer when it invokes message/packet handlers. The pointer is also passed to <code>ix_cc_wred_fini()</code> function for memory freeing when the core component is terminated.
Return Value	<p>Returns a valid <code>ix_error</code>.</p> <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. <code>IX_CC_ERROR_NULL</code>—operation failed, NULL parameter used (either <code>arg_CcHandle</code> or <code>arg_ppContext</code>) <code>IX_CC_ERROR_OOM_SYSTEM</code>—operation failed, local memory allocation failure <code>IX_CC_ERROR_INTERNAL</code>—operation failed, error in reading static configuration <code>IX_CC_ERROR_OOM_SRAM</code>—operation failed, SRAM memory allocation failure <code>IX_CC_ERROR_INTERNAL</code>—operation failed, packet or message handler registration failure <code>IX_CC_WRED_ERROR_FAILED_PATCHING</code>—operation failed, symbol patching procedure failure <code>IX_CC_WRED_ERROR_CCI</code>—operation failed, internal CCI error (only if level 1 used)

Table 21-3. WRED Empty Entry Pattern

Field name	Value in empty entry
<code>avg_len</code>	Zero
<code>green_count</code> , <code>yellow_count</code> , <code>red_count</code>	Zero
<code>green_packets</code> , <code>yellow_packets</code> , <code>red_packets</code>	Zero
<code>green_bytes</code> , <code>yellow_bytes</code> , <code>red_bytes</code>	Zero
<code>service_time</code>	0xFF
statistics (S)	Zero
<code>weight</code>	Zero
<code>green_const</code> , <code>yellow_const</code> , <code>red_const</code>	Zero
<code>green_min_th</code> , <code>yellow_min_th</code> , <code>red_min_th</code>	0xFFFF
<code>green_max_th</code> , <code>yellow_max_th</code> , <code>red_max_th</code>	0xFFFF
<code>green_drop_prob</code> , <code>yellow_drop_prob</code> , <code>red_drop_prob</code>	0

21.2.2 `ix_cc_wred_fini()`

The function terminates the WRED core component. It is executed when the execution engine running the core component is being shut down. This function:

- Frees SRAM memory allocated by the `ix_cc_wred_init()`.
- Removes a packet handler `ix_cc_wred_pkt_handler()`.
- Removes a message handler `ix_cc_wred_msg_handler()`.
- Frees an internal control block.

C Syntax

```
ix_error ix_cc_wred_fini (
    ix_cc_handle arg_CcHandle,
    void *arg_pContext);
```

Input

<code>arg_CcHandle</code>	Handle to the core component.
<code>arg_pContext</code>	Pointer to the control block that was allocated in <code>ix_cc_wred_init()</code> .

Output/Returns

Return Value	<p>Returns a valid <code>ix_error</code>.</p> <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • <code>IX_CC_ERROR_NULL</code>—operation failed, NULL parameter used (either <code>arg_CcHandle</code> or <code>arg_ppContext</code>) • <code>IX_CC_ERROR_INTERNAL</code>—operation failed, message or packet handler unregistration failure (level 0) • <code>IX_CC_ERROR_OOM_SRAM</code>—operation failed, de-allocation of SRAM memory failed • <code>IX_CC_WRED_ERROR_CCI</code>—operation failed, message or packet handler unregistration failure (level 1) • <code>IX_CC_WRED_ERROR_INVALID_HANDLE</code>—operation failed, the core component's handle (<code>arg_CcHandle</code>) does not match the control block (<code>arg_pContext</code>)
--------------	--

21.2.3 ix_cc_wred_pkt_handler()

The core component uses one packet handler for processing all packets. When invoked, the handler simply redirects a packet to the WRED microblock. The core component does not execute the WRED algorithm.

C Syntax

```
ix_error ix_cc_wred_pkt_handler (  
    ix_buffer_handle arg_Pkt,  
    ix_uint32 arg_ExceptionCode,  
    void *arg_pContext)
```

Input

arg_Pkt	Buffer handle to the received packet.
arg_ExceptionCode	An exception code thrown by a microblock; not used by the WRED core component.
arg_pContext	Pointer to the control block that was allocated in ix_cc_wred_init() .

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none">• <code>IX_SUCCESS</code>—operation succeeded, packet processed successfully• <code>IX_CCI_ERR_SEND_FAIL</code>—operation failed due to a failure to send the packet to the microblock
--------------	---

21.2.4 ix_cc_wred_msg_handler()

The core component uses one message handler for processing all types of messages. The handler receives messages from a single source—DiffServ application.

A set of supported messages manage WRED instances and configures their parameters. Internally, the handler function parses message payloads and calls appropriate Library API functions to perform the requested operation.

C Syntax

```
ix_error ix_cc_wred_msg_handler (  
    ix_buffer_handle arg_hDataToken,  
    ix_uint32 arg_UserData,  
    void *arg_pContext);
```


Input

arg_hDataToken Buffer handle embedding information for the message passed in arg_UserData.

arg_UserData Message type. Table 21-4 shows the supported messages.

arg_pContext Pointer to a control block that was allocated in ix_cc_wred_init()

:

Table 21-4. Supported Messages (ix_cc_wred_msg_handler)

Message type	Description
IX_CC_WRED_MSG_ADD_ENTRY	Add a new entry (instance) to the WRED table.
IX_CC_WRED_MSG_REMOVE_ENTRY	Delete an entry (instance) from the WRED table.
IX_CC_WRED_MSG_UPDATE_ENTRY	Update an entry (instance) in the WRED table.
IX_CC_WRED_MSG_GET_STATISTICS	Return 64-bit statistics associated with the selected WRED instance.

Output/Returns

Return Value Returns a valid ix_error.

- IX_SUCCESS—operation succeeded, message processed successfully
- IX_CC_ERROR_NULL—operation failed, arg_pContext or buffer handle is NULL
- IX_CC_ERROR_UNDEFINED_MSG—operation failed, invalid message type received
- IX_CC_ERROR_INTERNAL—operation failed, function was unable to free the buffer
- IX_CC_ERROR_SEND_FAIL—operation failed, reply message sending error

21.3 Message Helper API

The Message Helper is a wrapper library that facilitates sending messages to the WRED core component. There is one-to-one mapping between Message Helper API functions and message types supported by a core component. All the functions are asynchronous—a core component reports operation status in a callback routine. [Table 21-5](#) lists the WRED message helper functions.

Table 21-5. WRED Message Helper API

API Function	Description
ix_cc_wred_async_add_entry()	Adds a new WRED instance to the parameters table
ix_cc_wred_async_remove_entry()	Removes a WRED instance from the parameters table
ix_cc_wred_async_update_entry()	Updates a WRED instance parameters in the parameters table
ix_cc_wred_async_get_statistics()	Returns statistics associated with a WRED instance

21.3.1 ix_cc_wred_async_add_entry()

This message helper function builds and sends an entry add message to the WRED core component. The core component uses `arg_Instance` to select a row in the WRED table and performs the following tasks:

- reports an error (via callback) and does not override instance parameters stored in the table, if the row is already configured
- copies data from `arg_pInstanceParameters` to the selected entry, if an entry is empty.

C Syntax

```
ix_error ix_cc_wred_async_add_entry (
    ix_uint32 arg_Instance,
    ix_cc_wred_parameters *arg_pInstanceParameters,
    ix_cc_wred_cb_add_entry arg_Callback,
    void* arg_pUserContext);
```

Input

<code>arg_Instance</code>	Index of a WRED table entry that should be configured with new parameters.
<code>arg_pInstanceParameters</code>	Pointer to the structure of parameters defining a WRED instance. The structure layout is defined in Section 21.1.1 , “WRED Parameters Data Types” on page 435.
<code>arg_Callback</code>	A calling application-provided callback function. The core component invokes this function to report an operation status. See ix_cc_wred_cb_add_entry .
<code>arg_pUserContext</code>	Pointer to a calling application-defined context. This value is opaque to the core component. The context is returned in a callback function.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—operation succeeded, message was successfully built and sent <code>IX_CC_ERROR_NULL</code>—operation failed, <code>arg_pInstanceParameters</code> or <code>arg_Callback</code> argument is NULL <code>IX_CC_ERROR_OOM_SYSTEM</code>—operation failed, the function cannot allocate memory for message context <code>IX_CC_ERROR_SEND_FAIL</code>—operation failed, failure on sending a message
--------------	--

21.3.1.1 `ix_cc_wred_cb_add_entry`

The function prototype for message-handler callback function provided by the calling application to return an operation status.

C Syntax

```
ix_error (*ix_cc_wred_cb_add_entry) (
    ix_error arg_Result,
    void* arg_pContext);
```

Input

<code>arg_pContext</code>	Pointer to the calling application-defined context, provided as a parameter in the API call. Used by the calling application to match asynchronous request with a response.
---------------------------	---

Output/Returns

<code>arg_Result</code>	Error conditions for the call with the following returns:
Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—operation succeeded, WRED instance successfully configured <code>IX_CC_ERROR_CONFLICTING_ENTRY</code>—operation failed, WRED instance already configured <code>IX_CC_ERROR_RANGE</code>—operation failed, WRED instance out of range; the instance identifier is higher than the maximum number of instances that can be configured in the block <code>IX_CC_WRED_ERROR_INVALID_PARAM</code>—operation failed, <code>arg_pInstanceParameters</code> structure contains invalid values.

21.3.2 ix_cc_wred_async_remove_entry()

This message helper function builds and sends an entry remove message to the WRED core component. The core component first checks if an entry, pointed by `arg_Instance`, is valid and performs the following tasks:

- marks the entry as empty and changes the statistics value to zero, if WRED table entry is valid
- reports an error in a callback function, if the referred WRED table entry is already marked empty.

C Syntax

```
ix_error ix_cc_wred_async_remove_entry (
    ix_uint32 arg_Instance,
    ix_cc_wred_cb_remove_entry arg_Callback,
    void* arg_pUserContext);
```

Input

<code>arg_Instance</code>	Index of a WRED table entry that should be removed
<code>arg_Callback</code>	A calling application-provided callback function. The core component invokes this function to report an operation status. See ix_cc_wred_cb_remove_entry .
<code>arg_pUserContext</code>	Pointer to the calling application-defined context. This value is opaque to the core component. The context is returned in a callback function.

Output/Returns

Return Value	<p>Returns a valid <code>ix_error</code>.</p> <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—operation succeeded, message was successfully built and sent • <code>IX_CC_ERROR_NULL</code>—operation failed, <code>arg_pInstanceParameters</code> or <code>arg_Callback</code> argument is NULL • <code>IX_CC_ERROR_OOM_SYSTEM</code>—operation failed, the function cannot allocate memory for message context • <code>IX_CC_ERROR_SEND_FAIL</code>—operation failed, failure on sending a message
--------------	--

21.3.2.1 ix_cc_wred_cb_remove_entry

The function prototype for message-handler callback function provided by the calling application to return an operation status.

C Syntax

```
ix_error (*ix_cc_wred_cb_remove_entry) (
    ix_error arg_Result,
    void* arg_pContext);
```

Input

arg_pContext	Pointer to the calling application-defined context, provided as a parameter in the API call. Used by the calling application to match asynchronous request with a response.
--------------	---

Output/Returns

arg_Result	Error conditions for the call with the following return value:
Return Value	<p>Returns a valid ix_error.</p> <ul style="list-style-type: none"> • IX_SUCCESS—operation succeeded, WRED instance successfully removed • IX_CC_ERROR_ENTRY_NOT_FOUND—operation failed, WRED instance does not exist • IX_CC_ERROR_RANGE—operation failed, WRED instance out of range

21.3.3 ix_cc_wred_async_update_entry()

This message helper function builds and sends an entry update message to the WRED core component. The core component checks if a WRED entry, pointed by arg_Instance, is valid, and performs the following tasks

- updates a WRED entry with values provided in arg_pInstanceParameters, if a WRED entry is valid.
- reports an error and does not modify the WRED parameters, If the referred entry is empty. The table entry remains empty.

C Syntax

```
ix_error ix_cc_wred_async_update_entry (
    ix_uint32 arg_Instance,
    ix_cc_wred_parameters *arg_pInstanceParameters,
    ix_cc_wred_cb_update_entry arg_Callback,
    void* arg_pUserContext);
```

Input

<code>arg_Instance</code>	Index of a WRED table entry that should be updated
<code>arg_pInstanceParameters</code>	Pointer to the structure of updated parameters for a WRED instance. The structure layout is defined in Section 21.1.1, “WRED Parameters Data Types” on page 435.
<code>arg_Callback</code>	A calling application-provided callback function. The core component invokes this function to report an operation status. See ix_cc_wred_cb_update_entry .
<code>arg_pUserContext</code>	Pointer to the calling application-defined context. This value is opaque to the core component. The context is returned in a callback function.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—operation succeeded, message was successfully built and sent <code>IX_CC_ERROR_NULL</code>—operation failed, <code>arg_pInstanceParameters</code> or <code>arg_Callback</code> argument is NULL <code>IX_CC_ERROR_OOM_SYSTEM</code>—operation failed, the function cannot allocate memory for message context <code>IX_CC_ERROR_SEND_FAIL</code>—operation failed, failure on sending a message
--------------	--

21.3.3.1 [ix_cc_wred_cb_update_entry](#)

The function prototype for message-handler callback function provided by the calling application to return an operation status.

C Syntax

```
ix_error (*ix_cc_wred_cb_update_entry) (
    ix_error arg_Result,
    void* arg_pContext);
```

Input

<code>arg_pContext</code>	Pointer to the calling application-defined context, provided as a parameter in the API call. Used by the calling application to match asynchronous request with a response.
---------------------------	---

Output/Returns

<code>arg_Result</code>	Error conditions for the call with the following returns:
Return Value	<ul style="list-style-type: none"> <code>IX_SUCCESS</code>—operation succeeded, WRED instance successfully reconfigured <code>IX_CC_ERROR_ENTRY_NOT_FOUND</code>—operation failed, WRED instance does not exist <code>IX_CC_ERROR_RANGE</code>—operation failed, WRED instance out of range; the instance identifier is higher than the maximum number of instances that can be configured in the block <code>IX_CC_WRED_ERROR_INVALID_PARAM</code>—operation failed, <code>arg_pInstanceParameters</code> structure contains invalid values

21.3.4 `ix_cc_wred_async_get_statistics()`

This message helper function builds and sends a statistics read message to the WRED core component. The core component checks if the WRED entry, pointed by `arg_Instance`, is valid, and performs the following tasks:

- retrieves 64-bit statistical counters associated with this entry and returns them in a callback routine, if the WRED entry is valid.
- reports an error (via a callback routine), if the referred table entry is empty.

C Syntax

```
ix_error ix_cc_wred_async_get_statistics (
    ix_uint32 arg_Instance,
    ix_cc_wred_statistics *arg_pInstanceStatistics,
    ix_cc_wred_cb_get_statistics arg_Callback,
    void* arg_pUserContext);
```

Input

<code>arg_Instance</code>	Index of a WRED table entry that is queried for statistics
<code>arg_pInstanceStatistics</code>	Pointer to the statistics placeholder that should be filled with data by the core component. The structure layout is defined in Section 21.1.2, “WRED Statistics Data Type” on page 436.
<code>arg_Callback</code>	A calling application-provided callback function. The core component invokes this function to report an operation status and statistic counters. See ix_cc_wred_cb_get_statistics .

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—operation succeeded, message was successfully built and sent <code>IX_CC_ERROR_NULL</code>—operation failed, <code>arg_pInstanceParameters</code> or <code>arg_Callback</code> argument is NULL <code>IX_CC_ERROR_OOM_SYSTEM</code>—operation failed, the function cannot allocate memory for message context <code>IX_CC_ERROR_SEND_FAIL</code>—operation failed, failure on sending a message
--------------	--

21.3.4.1 `ix_cc_wred_cb_get_statistics`

The function prototype for message-handler callback function provided by the calling application to return an operation status.

C Syntax

```
ix_error (*ix_cc_wred_cb_get_statistics) (
    ix_error arg_Result;
    ix_cc_wred_statistics *arg_pInstanceStatistics,
    void* arg_pContext);
```

Input

<code>arg_pInstanceStatistics</code>	Pointer to the statistics structure filled with valid data if a WRED entry is not empty. The structure layout is defined in Section 21.1.2, “WRED Statistics Data Type” on page 436.
<code>arg_pContext</code>	Pointer to the calling application-defined context, provided as a parameter in the API call. Used by the calling application to match asynchronous request with a response.
<code>arg_pUserContext</code>	Pointer to the calling application-defined context. This value is opaque to the core component. The context is returned in a callback function.

Output/Returns

<code>arg_Result</code>	Error conditions for the call with the following returns:
Return Values	<ul style="list-style-type: none"> <code>IX_SUCCESS</code>—operation succeeded, <code>arg_pInstanceStatistics</code> contains valid statistics <code>IX_CC_ERROR_ENTRY_NOT_FOUND</code>—operation failed, WRED instance does not exist <code>IX_CC_ERROR_RANGE</code>—operation failed, WRED instance number out of range

21.4 Library API

The Library API provides direct C-style calls to the classifier core component. [Table 21-6](#) lists the WRED library functions.

Table 21-6. WRED Library API

API Function	Description
ix_cc_wred_add_entry()	Adds a new WRED instance to the parameters table
ix_cc_wred_remove_entry()	Removes a WRED instance from the parameters table
ix_cc_wred_update_entry()	Updates WRED instance parameters in the parameters table
ix_cc_wred_get_statistics()	Returns statistics associated with a WRED instance

The library API functions correspond one-to-one with message helper APIs. The difference is that library functions are executed in a calling application context. In addition, the Library API is synchronous. There are no callback functions to report operation results.

21.4.1 [ix_cc_wred_add_entry\(\)](#)

This function adds a new entry in the WRED table. The core component uses `arg_Instance` to select a row in the WRED table and performs the following tasks:

- reports an error and does not override instance parameters stored in the table, if the row is already configured.
- copies WRED instance parameters from `arg_pInstanceParameters` to the selected entry, if an entry is empty.

[Table 21-7](#) shows how the function sets field values in the new entry.

C Syntax

```
ix_error ix_cc_wred_add_entry(
    ix_uint32 arg_Instance,
    ix_cc_wred_parameters *arg_pInstanceParameters,
    void* arg_pContext);
```

Input

<code>arg_Instance</code>	Index of a WRED table entry that should be configured with new parameters. The value must not be higher than <code>\\WRED\\TABLE_SRAM_SIZE</code> system property.
<code>arg_pInstanceParameter s</code>	Pointer to the structure of parameters defining a WRED instance. The structure layout is defined in Section 21.1.1, “WRED Parameters Data Types” on page 435
<code>arg_pContext</code>	Pointer to the control block allocated in <code>ix_cc_wred_init()</code> .

Output/Returns

Return Value	<p>Returns a valid <code>ix_error</code>.</p> <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—operation succeed, new WRED instance successfully added and configured • <code>IX_CC_ERROR_CONFLICTING_ENTRY</code>—operation failed, WRED instance already configured • <code>IX_CC_ERROR_RANGE</code>—operation failed, WRED instance out of range; the instance identifier is higher than the maximum number of instances that can be configured in the block • <code>IX_CC_ERROR_NULL</code>—operation failed, <code>arg_pInstanceParameters</code> argument is NULL • <code>IX_CC_WRED_ERROR_INVALID_PARAM</code>—operation failed, <code>arg_pInstanceParameters</code> structure contains invalid WRED configuration (for example, the weight is too high)
--------------	---

Table 21-7. WRED Table Entry Fields Set by `ix_cc_wred_add_entry`

Field name	Value set by the function
<code>avg_len</code>	Zero
<code>green_count</code>	Zero
<code>yellow_count</code>	Zero
<code>red_count</code>	Zero
<code>green_packets</code>	Zero
<code>green_bytes</code>	Zero
<code>yellow_packets</code>	Zero
<code>yellow_bytes</code>	Zero
<code>red_packets</code>	Zero
<code>red_bytes</code>	Zero
<code>service_time</code>	<code>round(log2(WRED_UNIT_CHANGE_FACTOR/arg_pInstanceParameters->service_rate))</code>
<code>statistics (S)</code>	<pre>if (arg_pInstanceParameters->statsEnabled) statistics = 1; else statistics = 0;</pre>
<code>weight</code>	<code>arg_pInstanceParameters->weight</code>
<code>green_const</code>	<pre>if (arg_pInstanceParameters->green->max_th == arg_pInstanceParameters->green->min_th) green_const = 0xFFFF else green_const = arg_pInstanceParameters->green->max_prob / (arg_pInstanceParameters->green->max_th - arg_pInstanceParameters->green->min_th)</pre>

Table 21-7. WRED Table Entry Fields Set by ix_cc_wred_add_entry (Continued)

Field name	Value set by the function
yellow_const	<pre> if (arg_pInstanceParameters->yellow->max_th == arg_pInstanceParameters->yellow->min_th) yellow_const = 0xFFFF else yellow_const = arg_pInstanceParameters->yellow->max_prob / (arg_pInstanceParameters->yellow->max_th - arg_pInstanceParameters->yellow->min_th) </pre>
red_const	<pre> if (arg_pInstanceParameters->red->max_th == arg_pInstanceParameters->red->min_th) red_const = 0xFFFF else red_const = arg_pInstanceParameters->red->max_prob / (arg_pInstanceParameters->red->max_th - arg_pInstanceParameters->red->min_th) </pre>
green_min_th	<code>arg_pInstanceParameters->green->min_th << 8</code>
yellow_min_th	<code>arg_pInstanceParameters->yellow->min_th << 8</code>
red_min_th	<code>arg_pInstanceParameters->red->min_th << 8</code>
green_max_th	<code>arg_pInstanceParameters->green->max_th << 8</code>
yellow_max_th	<code>arg_pInstanceParameters->yellow->max_th << 8</code>
red_max_th	<code>arg_pInstanceParameters->red->max_th << 8</code>
green_drop_prob	Zero
yellow_drop_prob	Zero
red_drop_prob	Zero
sampling_time (in local config table)	Zero

The constant WRED_UNIT_CHANGE_FACTOR is equal to 37500000 ().

$6000000000 \text{Cycles/second}$

$16 \text{Cycles/timestamp}$

It is the factor used to change the rate in packets/s into rate expressed in packets/timestamp.

21.4.2 ix_cc_wred_remove_entry()

This function removes an entry from the WRED table. The core component first checks if an entry, pointed by `arg_Instance`, is valid, and performs the following tasks:

- marks the entry as empty and changes the statistics to zero, if an entry in the WRED table is valid.
- reports an error in a callback function, if the referred entry is already marked empty.

The empty entries are marked with special value of `service_time` equal to 0xFF. Note that the WRED microblock does not check the field against the special value. It behaves as if the entry was not empty. For this reason the empty entries must have the `max_th` and `min_th` fields for all colors equal to maximal 16-bit value (0xFFFF). This ensures that the microblock does not drop packets for packets directed to the queue. It also zeroes all the statistic fields. Additionally the core component sets the `xxx_const` fields for all the packet colors to zero.

C Syntax

```
ix_error ix_cc_wred_remove_entry (
    ix_uint32 arg_Instance,
    void* arg_pContext);
```

Input

<code>arg_Instance</code>	Index of a WRED table entry that should be removed. The value must not be higher than <code>\\WRED\\TABLE_SRAM_SIZE</code> system property.
<code>arg_pContext</code>	Pointer to the control block allocated in <code>ix_cc_wred_init()</code>

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—operation succeeded, entry successfully removed • <code>IX_CC_ERROR_ENTRY_NOT_FOUND</code>—operation failed, entry does not exist • <code>IX_CC_ERROR_RANGE</code>—operation failed, WRED instance out of range
--------------	---

21.4.3 ix_cc_wred_update_entry()

This function checks if a WRED entry, pointed by `arg_Instance`, is valid, and performs the following tasks:

- updates a WRED entry with values provided in `arg_pInstanceParameters`, if the referred WRED entry is valid.
- reports an error and does not modify the WRED parameters, if the referred entry is empty. The table entry remains empty.

Table 21-8 shows how the function sets field values in the updated entry.

C Syntax

```
ix_error ix_cc_wred_update_entry (
    ix_uint32 arg_Instance,
    ix_cc_wred_parameters *arg_pInstanceParameters,
    void* arg_pContext);
```

Input

arg_pContext	Pointer to the control block allocated in <code>ix_cc_wred_init()</code> .
arg_Instance	Index of a WRED table entry that should be updated. The value must not be higher than <code>\\WRED\\TABLE_SRAM_SIZE</code> system property.
arg_pInstanceParameters	Pointer to the structure of updated parameters for a WRED instance. The structure layout is defined in Section 21.1.1 , “WRED Parameters Data Types” on page 435.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> IX_SUCCESS—operation succeeded, entry successfully updated IX_CC_ERROR_ENTRY_NOT_FOUND—operation failed, entry does not exist IX_CC_ERROR_RANGE—operation failed, WRED instance out of range IX_CC_WRED_ERROR_INVALID_PARAM—operation failed, <code>arg_pInstanceParameters</code> structure contains invalid WRED configuration (for example the weight is too high)
--------------	--

Table 21-8. WRED Table Entry Fields Set by `ix_cc_wred_update_entry`

Field name	Value set by the function
avg_len	Not changed
green_count	Zero if statistics field changes from 0 to 1, otherwise not changed. The operation must be performed before the statistics field is written to SRAM.
yellow_count	Zero if statistics field changes from 0 to 1, otherwise not changed. The operation must be performed before the statistics field is written to SRAM.
red_count	Zero if statistics field changes from 0 to 1, otherwise not changed. The operation must be performed before the statistics field is written to SRAM.
green_packets	Zero if statistics field changes from 0 to 1, otherwise not changed. The operation must be performed before the statistics field is written to SRAM.
green_bytes	Zero if statistics field changes from 0 to 1, otherwise not changed. The operation must be performed before the statistics field is written to SRAM.
yellow_packets	Zero if statistics field changes from 0 to 1, otherwise not changed. The operation must be performed before the statistics field is written to SRAM.
yellow_bytes	Zero if statistics field changes from 0 to 1, otherwise not changed. The operation must be performed before the statistics field is written to SRAM.
red_packets	Zero if statistics field changes from 0 to 1, otherwise not changed. The operation must be performed before the statistics field is written to SRAM.

Table 21-8. WRED Table Entry Fields Set by ix_cc_wred_update_entry (Continued)

Field name	Value set by the function
red_bytes	Zero if statistics field changes from 0 to 1, otherwise not changed. The operation must be performed before the statistics field is written to SRAM.
service_time	<code>round(log2(WRED_UNIT_CHANGE_FACTOR/arg_pInstanceParameters->service_rate))</code>
statistics (S)	<pre>if(arg_pInstanceParameters->statsEnabled) statistics = 1; else statistics = 0;</pre>
weight	<code>arg_pInstanceParameters->weight</code>
green_const	<pre>if (arg_pInstanceParameters->green->max_th == arg_pInstanceParameters->green->min_th) green_const = 0xFFFF else green_const = arg_pInstanceParameters->green->max_prob / (arg_pInstanceParameters->green->max_th - arg_pInstanceParameters->green->min_th)</pre>
yellow_const	<pre>if (arg_pInstanceParameters->yellow->max_th == arg_pInstanceParameters->yellow->min_th) yellow_const = 0xFFFF else yellow_const = arg_pInstanceParameters->yellow->max_prob / arg_pInstanceParameters->yellow->max_th - arg_pInstanceParameters->yellow->min_th)</pre>
red_const	<pre>if (arg_pInstanceParameters->red->max_th == arg_pInstanceParameters->red->min_th) red_const = 0xFFFF else red_const = arg_pInstanceParameters->red->max_prob / (arg_pInstanceParameters->red->max_th - arg_pInstanceParameters->red->min_th)</pre>
green_min_th	<code>arg_pInstanceParameters->green->min_th << 8</code>
yellow_min_th	<code>arg_pInstanceParameters->yellow->min_th << 8</code>
red_min_th	<code>arg_pInstanceParameters->red->min_th << 8</code>
green_max_th	<code>arg_pInstanceParameters->green->max_th << 8</code>
yellow_max_th	<code>arg_pInstanceParameters->yellow->max_th << 8</code>
red_max_th	<code>arg_pInstanceParameters->red->max_th << 8</code>
green_drop_prob	Not changed
yellow_drop_prob	Not changed
red_drop_prob	Not changed
sampling_time (in local config table)	Not changed

The constant WRED_UNIT_CHANGE_FACTOR is defined in the [Section 21.4.1](#), “`ix_cc_wred_add_entry()`” on page 449.

21.4.4 `ix_cc_wred_get_statistics()`

This function returns statistic counters associated with a WRED instance. The core component checks if the WRED entry, pointed by `arg_Instance`, is valid, and performs the following tasks:

- reports an error (via a callback routine), if the referred table entry is empty.
- retrieves 64-bit statistic counters associated with the referred entry and fills in `arg_pInstanceStatistics` with valid data, if the referred WRED entry is valid.

C Syntax

```
ix_error ix_cc_wred_async_get_statistics (
    ix_uint32 arg_Instance,
    ix_cc_wred_statistics *arg_pInstanceStatistics,
    void* arg_pContext);
```

Input

<code>arg_Instance</code>	Index of a WRED table entry that should be updated
<code>arg_pContext</code>	Pointer to the control block allocated in <code>ix_cc_wred_init()</code> .

Output/Returns

<code>arg_pInstanceStatistics</code>	Pointer to the statistics structure that should be filled with data by the core component. The structure layout is defined in Section 21.1.2, “WRED Statistics Data Type” on page 436.
--------------------------------------	--

Return Value	<p>Returns a valid <code>ix_error</code>.</p> <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—operation succeeded, statistics successfully stored in a callback structure • <code>IX_CC_ERROR_ENTRY_NOT_FOUND</code>—operation failed, entry does not exist • <code>IX_CC_ERROR_RANGE</code>—operation failed, WRED instance out of range
--------------	---

The core component provides the following functionality:

- Initializes and configures the DSCP classifier microblock by patching symbols.
- Provides an API functions to set classification rules for interfaces and read the current classification rules for interfaces. The rules are stored in a configuration table shared between the core component and the microblock.
- Provides API functions to read per-rule statistics.
- Implements DSCP classification for slow path traffic received from other components (for example local packet from the Stack Driver). If the configuration entry requires remarking DSCP value, the core component changes the DSCP value in the packet.

For complete details see [Chapter 59, “DSCP Classifier Core Component”](#) of *Intel® Internet Exchange Architecture Software Building Blocks Developer’s Manual*.

22.1 Data Structures in Functional APIs

The data structures are used in calls to functional APIs—Message Helper API and Library API. [Table 22-1](#) lists the data structures defined by the DSCP classifier core component for configuring exact-match rules.

Table 22-1. Data Structures Defined by the DSCP Classifier Core Component

Data Structure	Description
<code>ix_s_cc_classifier_dscp_result</code>	A QoS classification results for a single <input port, DSCP value> pair.
<code>ix_s_cc_classifier_dscp_statistics</code>	64-bit statistics associated with <input port, DSCP value> pair.

22.1.1 Classification Result Data Type

Classification results for <input port, DSCP value> pair are specified using the structure

22.1.1.1 ix_s_cc_classifier_dscp_result

C Syntax

```
typedef struct ix_s_cc_classifier_dscp_result
{
    ix_uint32 flowId; /* Flow ID*/
    ix_uint16 classId; /* Class ID */
    ix_uint8 colorId; /* Color ID */
    ix_uint8 statistics; /* statistics flag—indicates whether
                        statistics is gathered for the rule*/
    ix_uint8 TC; /* Traffic Conditioning flag—indicates whether
                metering is the next processing step*/
    ix_uint8 remark; /* Remark flag—indicates whether
                    the DSCP value should be remarked*/
} ix_s_cc_classifier_dscp_result;
```

22.1.2 Statistics Data Type

Statistics associated with a <input port, DSCP value> pair consist of two 64-bit counters—packet counter and byte counter. The current counter values are returned using the structure.

22.1.2.1 ix_s_cc_classifier_dscp_statistics

C Syntax

```
typedef struct ix_s_cc_classifier_dscp_statistics
{
    ix_uint64 packets; /* number of packets for the pair */
    ix_uint64 bytes; /* number of bytes for the pair */
} ix_cc_classifier_dscp_statistics;
```

22.2 Core Component Infrastructure API

The DSCP classifier core component supports the following Core Component Infrastructure APIs.

Table 22-2. DSCP Classifier Core Component Infrastructure API

API Function	Description
<code>ix_cc_classifier_dscp_init()</code>	Initializes the core component

Table 22-2. DSCP Classifier Core Component Infrastructure API

API Function	Description
<code>ix_cc_classifier_dscp_fini()</code>	Terminates the core component
<code>ix_cc_classifier_dscp_pkt_handler()</code>	Message handler for classification rule change requests
<code>ix_cc_classifier_dscp_msg_handler()</code>	Packet handler for processing local packets from the Stack Driver

22.2.1 `ix_cc_classifier_dscp_init()`

The function initializes the core component. It should be called and returned before any other function in the core component can be called. The function performs the following:

- Allocates SRAM memory for the configuration table and populates the entries with the default configuration values read from the system registry. It sets the Input Port Configured flag in the entries to zero. The function calculates the amount of allocated memory using the formula

$$\text{size} = \text{CLASSIFIER_DSCP_IN_PORTS} * 64 \\ * \text{CLASSIFIER_DSCP_CONFIG_ENTRY_SRAM_SIZE}$$

- If the microblock supports statistics (`CLASSIFIER_DSCP_PACKET_COUNTER_FEATURE` flag is asserted), the function allocates SRAM memory for 64-bit statistics table and initializes it with zero values. The size of the table is calculated using the formula

$$\text{size} = \text{CLASSIFIER_DSCP_IN_PORTS} * 64 * \\ \text{CLASSIFIER_DSCP_STATS_ENTRY_SRAM_SIZE}$$

- Patches DSCP classifier microcode with imported variables, as specified in [Section 33.2.4.3, “Imported Variables”](#) on page 568 in [Chapter 33, “DSCP Classifier Microblock”](#) of *Intel® Internet Exchange Architecture Software Building Blocks Developer’s Manual*.
- Registers a packet handler `ix_cc_classifier_dscp_pkt_handler()` for inputs: `IX_CC_CLASSIFIER_DSCP_LOCAL_PKT_INPUT` (single source mode)
- Registers a message handler `ix_cc_classifier_dscp_msg_handler()` for inputs: `IX_CC_CLASSIFIER_DSCP_MSG_INPUT` (multiple sources mode)
- Allocates and initializes a bitmap to store information which input ports have its own interface-specific set of classification rules.
- Allocates a control block, and returns a pointer to this block in `arg_ppContext`.

C Syntax

```
ix_error ix_cc_classifier_dscp_init (
    ix_cc_handle arg_CcHandle,
    void **arg_ppContext);
```

Input

`arg_CcHandle` A handle to the core component.

Output/Returns

<code>arg_ppContext</code>	Location where the pointer to the control block allocated by the core component is stored. The control block is internal to the core component and contains variables and internal data structures. This pointer is used later, and passed to the <code>ix_cc_classifier_dscp_fini()</code> function by the Core Component Infrastructure to free memory when the core component is terminated.
Return value	<p>Returns a valid <code>ix_error</code>:</p> <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—operation succeeded, initialization completed. • <code>IX_CC_ERROR_INTERNAL</code>—operation failed, an error occurred during message or packet handlers registration • <code>IX_CC_ERROR_NULL</code>—operation failed, <code>arg_ppContext</code> or <code>arg_CcHandle</code> argument is NULL • <code>IX_CC_ERROR_OOM_SYSTEM</code>—operation failed, system memory allocation failure • <code>IX_CC_ERROR_OOM_SRAM</code>—operation failed, SRAM memory allocation failure • <code>IX_CC_CLASSIFIER_DSCP_ERROR_REGISTRY</code>—operation failed, the function failed to open or read the system repository • <code>IX_CC_CLASSIFIER_DSCP_PATCH_SYMBOLS</code>—operation failed, the function failed to patch symbols • <code>IX_OSSL_ERROR_MUTEX_INIT_FAILED</code>—operation failed, the function failed to allocate a mutex

22.2.2 `ix_cc_classifier_dscp_fini()`

The function terminates the DSCP classifier core component. It is executed when the execution engine running the core component is being shut down. The function performs the following:

- Frees SRAM memory allocated by the `ix_cc_classifier_dscp_init()`.
- Removes a packet handler `ix_cc_classifier_dscp_pkt_handler()`.
- Removes a message handler `ix_cc_classifier_dscp_msg_handler()`.
- Frees an internal control block.

C Syntax

```
ix_error ix_cc_classifier_dscp_fini (
    ix_cc_handle arg_CcHandle,
    void *arg_pContext);
```

Input

<code>arg_CcHandle</code>	Handle to the core component.
<code>arg_pContext</code>	Pointer to the control block allocated in <code>ix_cc_classifier_dscp_init()</code> .

Output/Returns

Return Value	Returns a valid <code>ix_error</code> :
	<ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—operation succeeded, shutdown completed. • <code>IX_CC_ERROR_INTERNAL</code>—operation failed, an error occurred during message or packet handlers unregistration • <code>IX_CC_ERROR_NULL</code>—operation failed, <code>arg_pContext</code> is NULL • <code>IX_CC_CLASSIFIER_DSCP_ERROR_INVALID_CC_HANDLE</code>—operation failed, invalid data in <code>arg_CcHandle</code>

22.2.3 `ix_cc_classifier_dscp_pkt_handler()`

The core component uses the packet handler for processing local IPv6 and IPv4 packets generated by Stack Driver.

Packets other than IPv6 or IPv4 are sent to `IX_CC_CLASSIFIER_DSCP_INV_IP_OUTPUT`, and the handler reports an error.

For IPv6 or IPv4 packets, the function retrieves DSCP value from the IP header and looks up configuration entry in the SRAM configuration table. Next the packet handler updates QoS related metadata variables according to the lookup result.

If `CLASSIFIER_DSCP_PACKET_COUNTER_FEATURE` global flag is asserted and statistics for the rule are enabled, the packet handler updates per-rule statistics. The core component implements the same statistics update algorithm as the microblock. This ensures synchronization in access to the statistics variables. The core component uses `ix_rm_atomic_sram_test_and_add()` and `ix_rm_atomic_sram_add()` functions implemented by the Resource Manager.

If the IPC flag in the configuration entry is asserted, the function sends IPv4 packets to `IX_CC_CLASSIFIER_DSCP_IP4_DEFAULT_OUTPUT` and IPv6 packets to `IX_CC_CLASSIFIER_DSCP_IP6_DEFAULT_OUTPUT`.

If the TC flag is set in the configuration entry, the function sends the packet to the `IX_CC_CLASSIFIER_DSCP_METER` output.

If the remark flag is set in the configuration entry, the function changes the DSCP value carried in the packet. Next, it sends the packet to the forwarder core component (either `IX_CC_CLASSIFIER_DSCP_IP4_FWD_OUTPUT` or `IP6_FWD_OUTPUT`).

C Syntax

```
ix_error ix_cc_classifier_dscp_pkt_handler (
    ix_buffer_handle arg_Pkt,
    ix_uint32 arg_ExceptionCode,
    void *arg_pContext)
```

Input

<code>arg_Pkt</code>	Buffer handle to the IP packet.
<code>arg_ExceptionCode</code>	Equals to zero when Stack Driver originates the packet.
<code>arg_pContext</code>	Pointer to a control block (allocated in ix_cc_classifier_dscp_init()) that is passed to the core component when a packet arrives.

Output/Returns

Return Value	<p>Returns a valid <code>ix_error</code>:</p> <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—operation succeeded, packet processed. • <code>IX_CC_ERROR_NULL</code>—operation failed, <code>arg_pContext</code> or <code>arg_Pkt</code> are NULL or the hash table has not been created • <code>IX_CC_ERROR_INTERNAL</code>—operation failed, the function failed to get the packet meta data or the packet payload • <code>IX_CC_ERROR_SEND_FAIL</code>—operation failed, reply message sending failure • <code>IX_CC_CLASSIFIER_DSCP_ERROR_INVALID_BUF_HANDLE</code>—operation failed, buffer handle is NULL
--------------	---

22.2.4 `ix_cc_classifier_dscp_msg_handler()`

The core component uses one message handler for processing all types of messages. The DSCP classifier core components receives messages from the DiffServ Application.

The suite of supported messages directly reflects a subset of Lookup Library API primitives pertinent to DSCP classification. The handler function parses message payloads and calls appropriate DSCP classification core components Library API functions to process the message.

C Syntax

```
ix_error ix_cc_classifier_dscp_msg_handler (
    ix_buffer_handle arg_hDataToken,
    ix_uint32 arg_UserData,
    void *arg_pContext);
```

Input

<code>arg_hDataToken</code>	Buffer handle embedding information for the message passed in <code>arg_UserData</code> .
<code>Arg_UserData</code>	Message type. Table 22-3 lists the messages supported.
<code>arg_pContext</code>	Pointer to a control block allocated in <code>ix_cc_classifier_dscp_init()</code> passed to the core component when a message arrives.

Table 22-3. Messages Supported by DSCP Classifier

Message type	Description
<code>IX_CC_CLASSIFIER_DSCP_MSG_ADD_IF_CONFIG</code>	Add an interface-specific set of classification rules.
<code>IX_CC_CLASSIFIER_DSCP_MSG_REMOVE_IF_CONFIG</code>	Delete an interface-specific set of classification rules.
<code>IX_CC_CLASSIFIER_DSCP_MSG_UPDATE_IF_CONFIG</code>	Update an interface-specific set of classification rules.
<code>IX_CC_CLASSIFIER_DSCP_MSG_GET_IF_CONFIG</code>	Return an interface-specific set of classification rules.
<code>IX_CC_CLASSIFIER_DSCP_MSG_SET_DEF_RULES</code>	Change default classification rules.
<code>IX_CC_CLASSIFIER_DSCP_MSG_GET_DEF_RULES</code>	Return default classification rules
<code>IX_CC_CLASSIFIER_DSCP_MSG_GET_STATS</code>	Return statistics associated with a <input port, DSCP value> pair.

Output/Returns

Return Value	<p>Returns a valid <code>ix_error</code>:</p> <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—operation succeeded, message processed successfully <code>IX_CC_ERROR_INTERNAL</code>—operation failed, internal error occurred <code>IX_CC_ERROR_NULL</code>—operation failed, <code>arg_pContext</code> is NULL or the hash table has not been created <code>IX_CC_ERROR_SEND_FAIL</code>—operation failed, reply message sending failure <code>IX_CC_ERROR_UNDEFINED_MSG</code>—operation failed, unknown message type received <code>IX_CC_CLASSIFIER_DSCP_ERROR_FREE_BUFFER</code>—operation failed, buffer memory deallocation failure <code>IX_CC_CLASSIFIER_DSCP_ERROR_INVALID_HANDLE</code>—operation failed, buffer handle is NULL
--------------	--

22.3 Message Helper API

The Message Helper is a wrapper library that facilitates sending messages to the DSCP classifier core component. There is one-to-one mapping between Message Helper API primitives and message types supported by a core component. All API functions are asynchronous—a core component reports operation status in a callback routine.

Table 22-4. DSCP Classifier Message Helper API

API Function	Description
<code>ix_cc_classifier_dscp_async_add_if_config()</code>	Adds an interface-specific set of classification rules.
<code>ix_cc_classifier_dscp_async_remove_if_config()</code>	Deletes an interface-specific set of classification rules
<code>ix_cc_classifier_dscp_async_update_if_config()</code>	Updates selected fields of the lookup result associated with a specified classification pattern.
<code>ix_cc_classifier_dscp_async_get_if_config()</code>	Returns an interface-specific set of classification rules.
<code>ix_cc_classifier_dscp_async_set_def_rules()</code>	Changes default classification rules.
<code>ix_cc_classifier_dscp_async_get_def_rules()</code>	Returns default classification rules
<code>ix_cc_classifier_dscp_async_get_statistics()</code>	Returns statistics associated with a <input port, DSCP value> pair.

22.3.1 `ix_cc_classifier_dscp_async_add_if_config()`

This helper function builds and sends an add interface configuration message to the classifier core components. If an interface-specific configuration already exists for the input port, the core component returns an error code. Otherwise it uses the set of classification results, `arg_pClassifResults`, to fill in all the configuration entries for the interface.

C Syntax

```
ix_error ix_cc_classifier_dscp_async_add_if_config (
    ix_uint16 arg_inputPort,
    ix_cc_classifier_dscp_result *arg_pClassifResults,
    ix_cc_classifier_dscp_cb_add_if_config arg_Callback,
    void* arg_pUserContext);
```

Input

<code>arg_inputPort</code>	Interface for which the classification results are added.
<code>arg_pClassifResults</code>	64-element table indexed with DSCP value. Each entry of this table contains QoS parameters assigned to packets received from the input port and carrying the DSCP value. The element structure is defined in Section 22.1.1, “Classification Result Data Type” on page 458 .
<code>arg_Callback</code>	A calling application-provided callback function. See ix_cc_classifier_dscp_cb_add_if_config .

Input (Continued)

<code>arg_pUserContext</code>	Pointer to calling application-defined context. This value is opaque to the core component. The context is returned in the callback function ix_cc_classifier_dscp_cb_add_if_config .
-------------------------------	---

Output/Returns

Return Value	<p>Returns a valid <code>ix_error</code>:</p> <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—operation succeeded, message was successfully built and sent • <code>IX_CC_ERROR_OOM_SYSTEM</code>—operation failed, not enough memory to send a message • <code>IX_CC_ERROR_NULL</code>—operation failed, NULL pointer in input parameters • <code>IX_CC_ERROR_SEND_FAIL</code>—operation failed, failure at sending a message
--------------	--

22.3.1.1 [ix_cc_classifier_dscp_cb_add_if_config](#)

The function prototype for message-handler callback functions provided by the calling application to report an operation status.

C Syntax

```
ix_error (*ix_cc_classifier_dscp_cb_add_if_config) (
    ix_error arg_Result,
    void* arg_pContext);
```

Input

<code>arg_pContext</code>	Pointer to calling application-defined context, provided as a parameter in the ix_cc_classifier_dscp_async_add_if_config() call. Used by the calling application to match asynchronous request with a response.
---------------------------	---

Output/Returns

<code>arg_Result</code>	<p>Indicates error conditions for the call—returns a valid <code>ix_error</code>:</p> <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—operation succeed, entry added • <code>IX_ERROR_RANGE</code>—operation failed, the input port number is out of range • <code>IX_CC_ERROR_DUPLICATE_ENTRY</code>—operation failed, the input port already configured
-------------------------	---

22.3.2 `ix_cc_classifier_dscp_async_remove_if_config()`

This helper function builds and sends a remove interface configuration message to the classifier core components. If the input port has interface-specific configuration, the core component deletes interface-specific set of classification rules for the input port and applies the default settings. Otherwise it reports an error.

C Syntax

```
ix_error ix_cc_classifier_dscp_async_remove_if_config (
    ix_uint16 arg_inputPort,
    ix_cc_classifier_dscp_cb_remove_if_config arg_Callback,
    void* arg_pContext);
```

Input

<code>arg_inputPort</code>	Identifies input port whose configuration is deleted
<code>arg_Callback</code>	A calling application-provided callback function. See ix_cc_classifier_7t_cb_remove_if_config .
<code>arg_pUserContext</code>	Pointer to calling application-defined context. This value is opaque to the core component. The context is returned in the callback function ix_cc_classifier_7t_cb_remove_if_config .

Output/Returns

Return Value	Returns a valid <code>ix_error</code> : <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—operation succeeded, message was successfully built and sent <code>IX_CC_ERROR_OOM_SYSTEM</code>—operation failed, not enough memory to send a message <code>IX_CC_ERROR_NULL</code>—operation failed, NULL pointer in input parameters <code>IX_CC_ERROR_SEND_FAIL</code>—operation failed, failure at sending a message
--------------	--

22.3.2.1 `ix_cc_classifier_7t_cb_remove_if_config`

The function prototype for message-handler callback functions provided by the calling application to report an operation status.

C Syntax

```
ix_error (*ix_cc_classifier_7t_cb_remove_if_config) (
    ix_error arg_Result,
    void* arg_pContext);
```

Input

<code>arg_pContext</code>	Pointer to the calling application-defined context, provided as a parameter in the API call. Used by the calling application to match asynchronous request with a response.
---------------------------	---

Output/Returns

<code>arg_Result</code>	<p>Indicates error conditions for the call—returns a valid <code>ix_error</code>:</p> <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—operation succeeded, configuration successfully removed • <code>IX_CC_ERROR_ENTRY_NOT_FOUND</code>—operation failed, interface-specific configuration for this port has not been defined • <code>IX_ERROR_RANGE</code>—operation failed, the input port number is out of range
-------------------------	--

22.3.3 `ix_cc_classifier_dscp_async_update_if_config()`

This helper function builds and sends an update interface configuration message to the classifier core components. If the input port has interface-specific configuration, the core component changes the set of classification rules for the port. Otherwise it reports an error.

C Syntax

```
ix_error ix_cc_classifier_dscp_async_update_if_config(
    ix_uint16 arg_inputPort,
    ix_cc_classifier_dscp_result *arg_pClassifResults,
    ix_cc_classifier_dscp_cb_update_if_config arg_Callback,
    void* arg_pUserContext);
```

Input

<code>arg_inputPort</code>	Interface for which the classification results are changed.
<code>arg_pClassifResults</code>	64-element table indexed with DSCP value and containing the new set of classification rules. Each entry of this table contains QoS parameters assigned to packets received from the input port and carrying the DSCP value. The element structure is defined in section 7.3.5.2.1.
<code>arg_Callback</code>	A calling application-provided callback function. See ix_cc_classifier_dscp_cb_update_if_config .
<code>arg_pUserContext</code>	Pointer to the calling application-defined context. This value is opaque to the core component. The context is returned in a callback function.

Output/Returns

Return Value

Returns a valid `ix_error`:

- `IX_SUCCESS`—operation succeeded, message was successfully built and sent
- `IX_CC_ERROR_OOM_SYSTEM`—operation failed, not enough memory to send a message
- `IX_CC_ERROR_NULL`—operation failed, NULL pointer in input parameters
- `IX_CC_ERROR_SEND_FAIL`—operation failed, failure at sending a message

22.3.3.1 `ix_cc_classifier_dscp_cb_update_if_config`

The function prototype for message-handler callback functions provided by the calling application to report an operation status.

C Syntax

```
ix_error (*ix_cc_classifier_dscp_cb_update_if_config) (
    ix_error arg_Result,
    void* arg_pContext);
```

Input

`arg_pContext`

Pointer to a calling application-defined context, provided as a parameter in the API call. Used by the calling application to match asynchronous request with a response.

Output /Returns

`arg_Result`

indicates error conditions for the call

Return Value

- `IX_SUCCESS`—operation succeeded, configuration successfully updated
- `IX_CC_ERROR_ENTRY_NOT_FOUN`—operation failed, interface-specific rules for the port not exist
- `IX_ERROR_RANGE`—operation failed, the input port number is out of range

22.3.4 `ix_cc_classifier_dscp_async_get_if_config()`

This helper function builds and sends an get interface configuration message to the classifier core components. If the input port has interface-specific configuration, the core component returns all the 64 classification rules. Otherwise it reports an error.

C Syntax

```
ix_error ix_cc_classifier_dscp_async_get_if_config(
    ix_uint16 arg_inputPort,
    ix_cc_classifier_dscp_cb_search_entry arg_Callback,
    void* arg_pUserContext);
```

Input

<code>arg_inputPort</code>	Interface for which the classification results are retrieved.
<code>arg_Callback</code>	A calling application-provided callback function. See ix_cc_classifier_dscp_cb_get_if_config .
<code>arg_pUserContext</code>	Pointer to the calling application-defined context. This value is opaque to the core component. The context is returned in ix_cc_classifier_dscp_cb_get_if_config callback function.

Output/Returns

<code>arg_pClassifResult</code> <code>s</code>	Pointer to a placeholder for the interface-specific classification rules. It is a 64-element table. Each element contains QoS parameters assigned to packets received from the input port and carrying the DSCP value. The element structure is defined in Section 22.1.1, “Classification Result Data Type” on page 458. The calling application must only allocate memory for the table. When a response from the core component is received, the table is filled in and passed to the calling application callback function— ix_cc_classifier_dscp_cb_get_if_config .
Return Value	<ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—operation succeeded, message was successfully built and sent • <code>IX_CC_ERROR_OOM_SYSTEM</code>—operation failed, not enough memory to send a message • <code>IX_CC_ERROR_NULL</code>—operation failed, NULL pointer in input parameters • <code>IX_CC_ERROR_SEND_FAIL</code>—operation failed, failure at sending a message

22.3.4.1 ix_cc_classifier_dscp_cb_get_if_config

The function prototype for message-handler callback functions provided by the calling application to report an operation status and return input port configuration.

C Syntax

```
ix_error (*ix_cc_classifier_dscp_cb_get_if_config) (
    ix_error arg_Result,
    ix_cc_classifier_dscp_result *arg_pClassifResults,
    void* arg_pContext);
```

Input

arg_pClassifResults	If the error code is IX_SUCCESS, the argument contains the set of rules for the input port. Otherwise, the table is not changed. The table element structure is defined in Section 22.1.1, “Classification Result Data Type” on page 458.
	Note: The table content is valid only in the callback function. If an application must use the information outside the function, it must copy the data to its private memory.
arg_pContext	Pointer to calling application-defined context, provided as a parameter in the API call. Used by the calling application to match asynchronous request with a response.

Output/Returns

arg_Result	Error token indicating error conditions for the call.
Return Value	Returns a valid ix_error: <ul style="list-style-type: none"> IX_SUCCESS—operation succeeded, the requested configuration found IX_CC_ERROR_ENTRY_NOT_FOUND—operation failed, interface-specific rules for the port not exist IX_ERROR_RANGE—operation failed, the input port number is out of range

22.3.5 ix_cc_classifier_dscp_async_set_def_rules()

This helper function builds and sends a set default rules message to the classifier core components. The core component changes default classification rules for the interfaces that do not have their own interface-specific rules.

C Syntax

```
ix_error ix_cc_classifier_dscp_async_set_def_rules(
    ix_cc_classifier_dscp_result *arg_pClassifResults,
    ix_cc_classifier_dscp_cb_set_def_rules arg_Callback,
    void* arg_pUserContext);
```

Input

<code>arg_pClassifResult</code> <code>s</code>	64-element table indexed with DSCP value and containing the new set of classification rules. Each entry of this table contains QoS parameters assigned to packets carrying the DSCP value. The element structure is defined in Section 22.1.1, “Classification Result Data Type” on page 458.
<code>arg_Callback</code>	A calling application-provided callback function. See ix_cc_classifier_dscp_cb_set_def_rules .
<code>arg_pUserContext</code>	Pointer to the calling application-defined context. This value is opaque to the core component. The context is returned in ix_cc_classifier_dscp_cb_set_def_rules callback function.

Output/Returns

<code>arg_pContext</code>	Pointer to calling application-defined context, provided as a parameter in the API call. Used by the calling application to match asynchronous request with a response.
Return Value	<ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—operation succeeded, message was successfully built and sent • <code>IX_CC_ERROR_OOM_SYSTEM</code>—operation failed, not enough memory to send a message • <code>IX_CC_ERROR_NULL</code>—operation failed, NULL pointer in input parameters • <code>IX_CC_ERROR_SEND_FAIL</code>—operation failed, failure at sending a message

22.3.5.1 [ix_cc_classifier_dscp_cb_set_def_rules](#)

The function prototype for message-handler callback functions provided by the calling application to report an operation status.

C Syntax

```
ix_error (*ix_cc_classifier_dscp_cb_set_def_rules) (
    ix_error arg_Result,
    void* arg_pContext);
```

Output/Returns

<code>arg_Result</code>	Indicates error conditions for the call.
Return Value	Returns a valid <code>ix_error</code> : <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—operation succeeded, configuration successfully updated

22.3.6 ix_cc_classifier_dscp_async_get_def_rules()

This helper function builds and sends a get default rules message to the classifier core components. The core component returns all the 64 classification rules applicable to the input port without interface specific configuration.

C Syntax

```
ix_error ix_cc_classifier_dscp_async_get_def_rules(
    ix_cc_classifier_dscp_cb_get_def_rules arg_Callback,
    void* arg_pUserContext);
```

Input

arg_pClassifResult s	Pointer to a placeholder for the interface-specific classification rules. It is a 64-element table. Each element contains QoS parameters assigned to packets carrying the DSCP value. The element structure is defined in Section 22.1.1, “Classification Result Data Type” on page 458 . The calling application must only allocate memory for the table. When a response from the core component is received, the table is filled in and passed to the calling application callback function.
arg_Callback	A calling application-provided callback function. See ix_cc_classifier_dscp_cb_get_def_rules .
arg_pUserContext	Pointer to the calling application-defined context. This value is opaque to the core component. The context is returned in ix_cc_classifier_dscp_cb_get_def_rules callback function.

Output/Returns

Return Value	<ul style="list-style-type: none"> IX_SUCCESS—operation succeeded, message was successfully built and sent IX_CC_ERROR_OOM_SYSTEM—operation failed, not enough memory to send a message IX_CC_ERROR_NULL—operation failed, NULL pointer in input parameters IX_CC_ERROR_SEND_FAIL—operation failed, failure at sending a message
--------------	--

22.3.6.1 ix_cc_classifier_dscp_cb_get_def_rules

The function prototype for message-handler callback functions provided by the calling application to report an operation status return input port configuration.

C Syntax

```
ix_error (*ix_cc_classifier_dscp_cb_get_def_rules) (
    ix_error arg_Result,
    ix_cc_classifier_dscp_result *arg_pClassifResults,
    void* arg_pContext);
```


Input

<code>arg_pClassifResult</code> s	The argument contains the set of default rules. The table element structure is defined in Section 22.1.1, “Classification Result Data Type” on page 458. Note: The table content is valid only in the callback function. If an application must use the information outside the function, it must copy the data to its private memory.
<code>arg_pContext</code>	Pointer to a calling application-defined context, provided as a parameter in the API call. Used by the calling application to match asynchronous request with a response.

Output/Returns

<code>arg_Result</code>	Error token indicating error conditions for the call.
Return Value	<ul style="list-style-type: none"> <code>IX_SUCCESS</code>—operation succeeded, data retrieved successful

22.3.7 `ix_cc_classifier_dscp_async_get_statistics()`

This helper function builds and sends a get statistics message to the classifier core components. If the microblock supports statistics gathering, the core component retrieves statistics for this <input port, dscp value> pair and sends them back to the calling application.

Note: The function returns statistics regardless of the type of configuration applied to the port (default or interface specific). It returns statistics even if statistics gathering is disabled in the port configuration.

If the microblock does not support statistics, the core component returns an error token.

C Syntax

```
ix_error ix_cc_classifier_dscp_async_get_statistics(
    ix_uint16 arg_inputPort,
    ix_uint8 arg_dscp,
    ix_cc_classifier_dscp_statistics *arg_pStatistics,
    ix_cc_classifier_dscp_cb_get_statistics arg_Callback,
    void* arg_pUserContext);
```

Input

<code>arg_inputPort</code>	Interface for which statistics are requested.
<code>arg_dscp</code>	DSCP value for which statistics are requested.
<code>arg_Callback</code>	A calling application-provided callback function. See ix_cc_classifier_dscp_cb_get_statistics .

Input (Continued)

`arg_pUserContext` Pointer to the calling application-defined context. This value is opaque to the core component. The context is returned in a callback function.

Output/Returns

`arg_pStatistics` Pointer to a placeholder for the statistics values associated with the <input port, dscp value> pair. The calling application must allocate memory for the data structure. The structure is defined in [Section 22.1.2, “Statistics Data Type” on page 458](#).

Return value Returns valid `ix_error`:

- `IX_SUCCESS`—operation succeeded, message was successfully built and sent
- `IX_CC_ERROR_OOM_SYSTEM`—operation failed, not enough memory to send a message
- `IX_CC_ERROR_NULL`—operation failed, NULL pointer in input parameters
- `IX_CC_ERROR_SEND_FAIL`—operation failed, failure at sending a message

22.3.7.1 `ix_cc_classifier_dscp_cb_get_statistics`

The function prototype for message-handler callback functions provided by the calling application to report an operation status.

C Syntax

```
ix_error (*ix_cc_classifier_dscp_cb_get_statistics) (
    ix_error arg_Result,
    ix_cc_classifier_dscp_statistics *arg_pStatistics,
    void* arg_pContext);
```

Input

`arg_pStatistics` If the error code is `IX_SUCCESS`, the argument contains statistics associated with the requested <input port, dscp value> pair. Otherwise, the structure is not changed. The statistics structure is defined in [Section 22.1.2, “Statistics Data Type” on page 458](#). This is the pointer passed to `ix_cc_classifier_dscp_async_get_statistics()` function.

`arg_pContext` Pointer to calling application-defined context, provided as a parameter in the API call. Used by the calling application to match asynchronous request with a response.

Output/Returns

arg_Result	Error token indicating error conditions for the call.
Return Value	Returns a valid <code>ix_error</code> : <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—operation succeeded, entry found • <code>IX_ERROR_RANGE</code>—operation failed, the input port number is out of range • <code>IX_CC_CLASSIFIER_DSCP_ERROR_STATS_NOT_SUPPORTED</code>—operation failed, the block does not support statistics

22.4 Library API

The Library API provides direct C-style calls to the classifier core component. [Table 22-5](#) lists the library API provided by the DSCP Classifier core components.

Table 22-5. DSCP Classifier Library API

API Function	Description
<code>ix_cc_classifier_dscp_add_if_config()</code>	Adds an interface-specific set of classification rules.
<code>ix_cc_classifier_dscp_remove_if_config()</code>	Deletes an interface-specific set of classification rules
<code>ix_cc_classifier_dscp_update_if_config()</code>	Updates selected fields of the lookup result associated with a specified classification pattern.
<code>ix_cc_classifier_dscp_set_def_rules()</code>	Returns an interface-specific set of classification rules.
<code>ix_cc_classifier_dscp_get_def_rules()</code>	Changes default classification rules.
<code>ix_cc_classifier_dscp_get_if_config()</code>	Returns default classification rules
<code>ix_cc_classifier_dscp_get_statistics()</code>	Returns statistics associated with a <input port, DSCP value> pair.

The library API functions correspond one-to-one with message helper API. The difference is that library functions are executed in a calling application context. In addition, the Library API is synchronous. There are no callback functions to report operation results.

22.4.1 `ix_cc_classifier_dscp_add_if_config()`

This function assigns interface-specific classification rules to an input port. If the input port already has its own classification rules, the function returns an error code. Otherwise, the function copies classification results given in `arg_pClassifResults` table to the appropriate entries in the SRAM configuration table. Next, the function marks the input port as configured. It also asserts Input Port Configured flag in all the entries.

If the microblock supports statistics and the statistics flag in any of the altered SRAM configuration entries changes from zero to one, the function zeroes the corresponding entries in the statistics table in SRAM.

C Syntax

```
ix_error ix_cc_classifier_dscp_add_if_config(  
    ix_uint16 arg_inputPort,  
    ix_cc_classifier_dscp_result *arg_pClassifResults,  
    void* arg_pContext);
```

Input

<code>arg_inputPort</code>	Interface for which the classification results are added.
<code>arg_pClassifResults</code>	64-element table indexed with DSCP value. Each entry of this table contains QoS parameters assigned to packets received from the input port and carrying the DSCP value. The element structure is defined in Section 22.1.1, “Classification Result Data Type” on page 458.
<code>arg_pContext</code>	Pointer to the control block allocated in ix_cc_classifier_dscp_init() .

Output/Returns

Return Value	Returns a valid <code>ix_error</code> : <ul style="list-style-type: none">• <code>IX_SUCCESS</code>—configuration successfully added.• <code>IX_CC_ERROR_NULL</code>—operation failed, <code>arg_pContext</code> argument is NULL.• <code>IX_CC_ERROR_DUPLICATE_ENTRY</code>—operation failed, an entry for the key already configured.• <code>IX_ERROR_RANGE</code>—operation failed, the input port number is out of range.
--------------	--

22.4.2 `ix_cc_classifier_dscp_remove_if_config()`

This function removes a set of classification rules assigned to an interface. If the input port does not have its own classification rules, the function returns an error code. Otherwise, the function changes configuration of the input port to the default configuration. It copies the default classification rules to the appropriate entries of the SRAM configuration table. Next the function marks the input port as not configured—that is, using the default configuration. It also sets the Input Port Configured flag in all the entries to zero.

If the microblock supports statistics and the statistics flag in any of the altered SRAM configuration entries changes from zero to one, the function zeroes the corresponding entries in the statistics table in SRAM.

C Syntax

```
ix_error ix_cc_classifier_dscp_remove_if_config (
    ix_uint16 arg_inputPort,
    void* arg_pContext);
```

Input

<code>arg_inputPort</code>	Identifies input port whose configuration is deleted
<code>arg_pContext</code>	Pointer to the control block allocated in <code>ix_cc_classifier_dscp_init()</code> .

Output/Returns

Return Value	Returns a valid <code>ix-error</code> : <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—operation succeeded, entry successfully removed <code>IX_CC_ERROR_ENTRY_NOT_FOUND</code>—operation failed, interface-specific configuration for this port has not been defined <code>IX_CC_ERROR_NULL</code>—operation failed, <code>arg_pContext</code> argument is NULL <code>IX_ERROR_RANGE</code>—operation failed, the input port number is out of range
--------------	--

22.4.3 `ix_cc_classifier_dscp_update_if_config()`

This function updates a set of classification rules associated with an interface. If the input port does not have its own classification rules, the function returns an error code. Otherwise, the function copies the new classification rules to the appropriate entries of the SRAM configuration table.

If the microblock supports statistics and the statistics flag in any of the altered SRAM configuration entries changes from zero to one, the function zeroes the corresponding entries in the statistics table in SRAM.

C Syntax

```
ix_error ix_cc_classifier_dscp_update_if_config(
    ix_uint16 arg_inputPort,
    ix_cc_classifier_dscp_result *arg_pClassifResults,
    void* arg_pContext);
```

Input

<code>arg_inputPort</code>	Interface for which the classification results are added.
<code>arg_pClassifResults</code>	64-element table indexed with DSCP value. Each entry of this table contains QoS parameters assigned to packets received from the input port and carrying the DSCP value. The element data structure is defined in Section 22.1.1, “Classification Result Data Type” on page 458 .
<code>arg_pContext</code>	Pointer to the control block allocated in ix_cc_classifier_dscp_init() .

Output/Returns

Return Value	Returns a valid <code>ix_error</code> : <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—operation succeeded, configuration successfully updated <code>IX_CC_ERROR_ENTRY_NOT_FOUND</code>—operation failed, interface-specific rules for the port not exist <code>IX_CC_ERROR_NULL</code>—operation failed, <code>arg_pClassifyResults</code> or <code>arg_pContext</code> argument is NULL <code>IX_ERROR_RANGE</code>—operation failed, the input port number is out of range
--------------	---

22.4.4 ix_cc_classifier_dscp_set_def_rules()

This function changes default classification rules for the interfaces that do not have their own interface-specific rules. The function copies the new rules to the entries in SRAM configuration table that contain classification results for input ports using default rules. The function stores the new default rules in the control block.

If the microblock supports statistics and the statistics flag in any of the altered SRAM configuration entries changes from zero to one, the function zeroes the corresponding entries in the statistics table in SRAM.

C Syntax

```
ix_error ix_cc_classifier_dscp_set_def_rules (
    ix_cc_classifier_dscp_result *arg_pClassifResults,
    void* arg_pContext);
```

Input

<code>arg_pClassifResult</code> <code>s</code>	64-element table indexed with DSCP value and containing the new set of classification rules. Each entry of this table contains QoS parameters assigned to packets carrying the DSCP value. The element structure is defined in Section 22.1.1, “Classification Result Data Type” on page 458.
<code>arg_pContext</code>	Pointer to the control block allocated in <code>ix_cc_classifier_dscp_init()</code> .

Output/Returns

Return Value	Returns a valid <code>ix_error</code> : <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—operation succeeded, configuration successfully updated • <code>IX_CC_ERROR_NULL</code>—operation failed, <code>arg_pClassifyResults</code> or <code>arg_pContext</code> argument is NULL
--------------	--

22.4.5 `ix_cc_classifier_dscp_get_def_rules()`

This function returns all the 64 classification rules applicable to the input port without interface specific configuration.

C Syntax

```
ix_error ix_cc_classifier_dscp_get_def_rules (
    ix_cc_classifier_dscp_result *arg_pClassifResults,
    void* arg_pContext);
```

Input

<code>arg_pContext</code>	Pointer to the control block allocated in <code>ix_cc_classifier_dscp_init()</code> .
---------------------------	---

Output/Returns

<code>arg_pClassifResult</code> <code>s</code>	<p>pointer to a placeholder for the interface-specific classification rules. It is a 64-element table. Each element contains QoS parameters assigned to packets carrying the DSCP value. The element structure is defined in Section 22.1.1, “Classification Result Data Type” on page 458.</p> <p>The calling application must only allocate memory for the table. When a response from the core component is received, the table is filled in and passed to the calling application callback function.</p>
---	--

Output/Returns (Continued)

Return Value	Returns a valid <code>ix_error</code> : <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—operation succeeded, configuration successfully retrieved <code>IX_CC_ERROR_NULL</code>—operation failed, <code>arg_pClassifyResults</code> or <code>arg_pContext</code> argument is NULL
--------------	--

22.4.6 `ix_cc_classifier_dscp_get_if_config()`

The function returns interface-specific configuration of an input port. If the input port does not have its own classification rules, the function returns an error code.

C Syntax

```
ix_error ix_cc_classifier_dscp_get_if_config(
    ix_uint16 arg_inputPort,
    ix_cc_classifier_dscp_result *arg_pClassifResults,
    void* arg_pContext);
```

Input

<code>arg_inputPort</code>	interface for which the classification results are retrieved.
<code>arg_pClassifResults</code>	<p>Pointer to a placeholder for the interface-specific classification rules. It is a 64-element table. Each element contains QoS parameters assigned to packets received from the input port and carrying the DSCP value. The element structure is defined in Section 22.1.1, “Classification Result Data Type” on page 458.</p> <p>The calling application must only allocate memory for the table. When a response from the core component is received, the table is filled in and passed to the calling application callback function.</p>
<code>arg_pContext</code>	<p>pointer to the control block allocated in <code>ix_cc_classifier_dscp_init()</code>.</p>

Output/Returns

<code>arg_pLookupResult</code>	Pointer to a placeholder for the lookup results associated with the specified classification pattern. The calling application sets type field to <code>IX_CC_CLASSIFIER_7T_QOS</code> or <code>IX_CC_CLASSIFIER_7T_FWD</code> . The core component fills in the remaining fields with lookup result as defined in section “Error! Reference source not found”.
Return Value	<p>Returns a valid <code>ix_error</code>:</p> <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—operation succeeded, the requested configuration found <code>IX_CC_ERROR_ENTRY_NOT_FOUND</code>—operation failed, interface-specific rules for the port not exist <code>IX_ERROR_RANGE</code>—operation failed, the input port number is out of range <code>IX_CC_ERROR_NULL</code>—operation failed, <code>arg_pClassifyResults</code> or <code>arg_pContext</code> argument is NULL

22.4.7 `ix_cc_classifier_dscp_get_statistics()`

If the microblock supports statistics, the core component returns statistics for the <input port, dscp value> pair. Otherwise, the core component returns an error code and the `arg_pStatistics` argument is left unaltered.

The function uses SRAM Read operations to get 64-bit counter values, because the operation does not change the statistics and does not have to be synchronized with statistics updates. This means that the core component can retrieve a counter value in the middle of a read-modify-write atomic operation performed by the microblock. In such a case, the SRAM controller returns the counter value before the atomic operation.

C Syntax

```
ix_error ix_cc_classifier_dscp_get_statistics (
    ix_uint16 arg_inputPort,
    ix_uint8 arg_dscp,
    ix_cc_classifier_dscp_statistics *arg_pStatistics,
    void* arg_pContext);
```

Input

<code>arg_inputPort</code>	Interface for which statistics are requested
<code>arg_dscp</code>	DSCP value for which statistics are requested
<code>arg_pContext</code>	Pointer to the control block allocated in <code>ix_cc_classifier_dscp_init()</code> .

Output/Returns`arg_pStatistics`

Pointer to a placeholder for the statistics values associated with the <input port, dscp value> pair. The calling application must allocate memory for the data structure. The structure is defined in [Section 22.1.2, “Statistics Data Type” on page 458](#).

Return Value

Returns a valid `ix_error`:

- `IX_SUCCESS`—operation succeeded, statistics returned successfully
- `IX_CC_ERROR_NULL`—operation failed, `arg_pStatistics` or `arg_pContext` argument is `NULL`
- `IX_ERROR_RANGE`—operation failed, the input port number is out of range
- `IX_CC_CLASSIFIER_DSCP_ERROR_STATS_NOT_SUPPORTED`—operation failed, the block does not support statistics

Support Libraries

The following chapters are included in this section:

- [Chapter 23, “Route Table Manager”](#)
Route Table Manager(RTM) for IPv4 pipeline primarily supports the IPv4 Forwarder Core Component.
RTM provides services to the other core components for maintaining route tables. It provides a way to create a new table, delete an existing table, add routes to a table, remove routes from a table, and look up a route in a table.
- [Chapter 24, “Route Table Manager for IPV6”](#)
Route Table Manager(RTM) for IPv6 pipeline primarily supports the IPv6 Forwarder Core Component.
RTM provides services to the other core components for maintaining route tables. It provides a way to create a new table, delete an existing table, add routes to a table, remove routes from a table, and look up a route in a table.
- [Chapter 25, “L2 Table Manager”](#)
L2 Table Manager exposes API for initializing, managing, updating and searching L2 table.
- [Chapter 26, “Message Helper and Support Library”](#)
Message Support Library provides the layer of translation between messages and core component APIs. Message Helper Library translates message API into the messages and enforces the mechanism of delivering messages to the core component Library APIs and sending the result back to the calling application.

The following table lists the libraries supported on the Ingress and Egress side:

Libraries needed on th Ingress side	Libraries needed on the Egress side
Route Table Manager for IPv4	L2 Table Manager
Route Table Manager for IPv6	Message Helper and Message Support Library
Message Helper and Message Support Library	

The Route Table Manager (RTM) is utilized by the IPv4 Forwarder core component to add, remove or look up items in the route table. Route information is maintained and searched by a Longest Prefix Match (LPM) algorithm. To manage next hops, the Route Table Manager uses a Next Hop Database (NHDB).

The Route Table Manager uses data structures and stores the data in the same format as the IPv4 Forwarder microblock. These data structures are allocated through the Resource Manager.

This implementation of the Route Table Manager is specific to the IP protocol, version 4. Because a Route Table Manager designed for IPv6 would require a different interface (e.g. 128-bit parameters instead of 32-bit), the Route Table Manager exposes its entry points in such a way that it only deals with IPv4 data types and algorithms.

The Route Table Manager uses a lookup library and therefore could be used with TCAM hardware.

For complete details, see [Chapter 60, “Route Table Manager”](#) of the *Intel® Internet Exchange Architecture (IXA) Software Building Blocks Developer’s Manual*.

23.1 Data Structures, Types, and Macros

This section describes the Route Table Manager data structures, types and macros.

23.1.1 Data Structures, Types

The Route Table Manager data structures and types are described in this section. [Table 23-1](#) shows a summary of the data structures and types.

Table 23-1. Route Table Manager Data Structures and Types

Name	Description
<code>ix_cc_rtmv4</code>	Opaque handle to the Route Table Manager.
<code>ix_cc_rtmv4_nhid</code>	Data type representing an IPv4 Next Hop ID.
<code>ix_cc_rtmv4_next_hop_info</code>	Structure defining a next hop.
<code>ix_cc_rtmv4_symbols</code>	Structure containing values useful in the microengines.
<code>ix_cc_rtmv4_statistics</code>	Structure containing RTM statistics.
<code>ix_cc_rtmv4_lkup_type</code>	Enumeration of the supported TCAM and software algorithms.
<code>ix_cc_rtmv4_mem_type</code>	Enumeration of the supported memory types.
<code>ix_cc_rtmv4_config</code>	Data structure indicating how the Route Table Manager should be created.

23.1.1.1 `ix_cc_rtmv4`

An opaque handle to the Route Table Manager, obtained by calling `ix_cc_rtmv4_init()`.

C Syntax

```
typedef struct ix_s_cc_rtmv4* ix_cc_rtmv4;
```

23.1.1.2 `ix_cc_rtmv4_nhid`

This is the data type representing an IPv4 next hop ID—abbreviated as NHID in function names and other identifiers throughout this chapter. It is defined as a typedef to the intrinsic data type, unsigned 32-bit integer. Applications at the highest levels allocate and assign the next hop ID values. Some values are reserved—those from -64 to -1.

C Syntax

```
typedef ix_uint32 ix_cc_rtmv4_nhid;
```

23.1.1.3 `ix_cc_rtmv4_next_hop_info`

This data structure is used by the Route Table Manager to define a structure describing a next hop. This structure is used when:

- adding a next hop—see [Section 23.2.3, “`ix_cc_rtmv4_add_next_hop\(\)`” on page 495](#)
- retrieving next hop information—see [Section 23.2.6, “`ix_cc_rtmv4_get_next_hop\(\)`” on page 498](#)
- Updating Next Hop Information—see [Section 23.2.5, “`ix_cc_rtmv4_update_next_hop\(\)`” on page 497](#)
- looking up a next hop by IP address—see [Section 23.2.13, “`ix_cc_rtmv4_lookup\(\)`” on page 506](#)

Though this data is identical in meaning to the next hop information used by the IPv4 Microblock—see *Intel® Internet Exchange Architecture Software Building Blocks Developer's Manual*—the structure used to store the data is less restrictive. The Route Table Manager internally packs the next hop data into the format the microengines expect.

The IPv4 Forwarder exposes this structure in its interface, requiring the Forwarding Plane module to understand the structure.

C Syntax

```
typedef struct ix_s_cc_rtmv4_next_hop_info {
    ix_uint32 bladeID;
    ix_uint32 l2Index;
    ix_uint32 portID;
    ix_uint32 mtu;
    ix_uint32 flags;
    ix_uint32 ipAddr;
    ix_uint32 l2IndexType;
} ix_cc_rtmv4_next_hop_info;
```

Data Elements

<code>bladeID</code>	The blade through which this next hop is reached. Packets destined for a next hop are sent to the corresponding blade. The low eight bits of this field are exported to the next hop structure of the IPv4 Microblock in the <code>Fabric Port ID</code> field.
<code>l2Index</code>	An index into the L2 table on the egress side of the referenced blade. The low 16 bits are exported to the next hop structure of the IPv4 microBlock in the <code>Nexthop ID</code> field. The low 16 bits must not be all ones—for example, the value <code>0xFFFFFFFF</code> or <code>0x0000FFFF</code> —or the Route Table Manager returns an error in this condition.
<code>portID</code>	The lowest 16 bits of each of <code>portID</code> are exported to the IPv4 Microblock in the <code>Out port ID</code> area.
<code>mtu</code>	The lowest 16 bits of <code>mtu</code> are exported to the IPv4 Microblock in the <code>MTU</code> area.
<code>flags</code>	The lowest eight bits of <code>flags</code> are exported to the IPv4 microblock in the <code>Flags</code> area.
<code>ipAddr</code>	Stored and returned when looked up, but not exported to the IPv4 microblock.
<code>l2Index</code>	Type field indicates the meaning of the <code>l2Index</code> field.

23.1.1.4 `ix_cc_rtmv4_symbols`

The Route Table Manager defines a structure containing values useful in the microengines. This structure is obtained by calling `ix_cc_rtmv4_get_symbols()`.

C Syntax

```
typedef struct ix_s_cc_rtmv4_symbols {
    ix_uint32 lkupTableID;
    ix_uint32 lkupTableData1;
    ix_uint32 lkupTableData2;
    ix_uint32 nextHopBase;
} ix_cc_rtmv4_symbols;
```

The `lkupTableID`, `lkupTableData1`, and `lkupTableData2` members are provided for patching the IPv4 Forwarder Microblock. This microblock uses these values in `ix_sw_lkup_lpm_build_handle()` and `ix_hw_lkup_lpm_build_handle()`.

23.1.1.5 ix_cc_rtmv4_statistics

The calling application retrieve current content statistics—but not storage capacity—about the Route Table Manager by calling `ix_cc_rtmv4_get_statistics()`.

C Syntax

```
typedef struct ix_s_cc_rtmv4_statistics {
    ix_uint32 numberOfRoutes;
    ix_uint32 numberOfNextHops;
} ix_cc_rtmv4_statistics;
```

23.1.1.6 ix_cc_rtmv4_lkup_type

Enumeration of the supported TCAM and Software algorithms.

C Syntax

```
typedef enum ix_e_cc_rtmv4_lkup_type {
    IX_CC_RTMV4_LKUP_SOFTWARE,
    IX_CC_RTMV4_LKUP_TCAM,
    IX_CC_RTMV4_LKUP_LAST
} ix_cc_rtmv4_lkup_type;
```

23.1.1.7 ix_cc_rtmv4_mem_type

Enumeration of the supported memory types—currently SRAM and SDRAM.

C Syntax

```
typedef enum ix_e_cc_rtmv4_mem_type {
    IX_CC_RTMV4_MEM_SDRAM,
    IX_CC_RTMV4_MEM_SRAM,
    IX_CC_RTMV4_MEM_TCAM,
    IX_CC_RTMV4_MEM_LAST
} ix_cc_rtmv4_mem_type;
```

23.1.1.8 ix_cc_rtmv4_config

Data structure indicating how the Route Table Manager should be created.

C Syntax

```
typedef struct ix_s_cc_rtmv4_config {
    ix_uint32 routeTableSizeHint;
    ix_cc_rtmv4_lkup_type lkupType;
    ix_cc_rtmv4_mem_type lkupMemType;
    ix_uint32 lkupChannel;
    ix_uint32 nhdbSizeHint;
    ix_cc_rtmv4_mem_type nhdbMemType;
    ix_uint32 nhdbChannel;
```



```
} ix_cc_rtmv4_config;
```

Data Elements

routeTableSizeHint	An approximate number of routes supported. The quantity indicated is not guaranteed but is used in determining the number of resources required.
lkupType	The type of lookup to use. The value of this data element must match the algorithm used by the microblock.
lkupMemType	The type of memory to use in the lookup algorithm. The value of this data element must match the memory type used by the microblock.
lkupChannel	The channel number where the microblock accesses the Longest Prefix Match (LPM) tables.
nhdbSizeHint	An approximate number of <i>next hops</i> supported. The quantity indicated is not guaranteed but is used in determining the number of resources required.
nhdbMemType	The type of memory to use in the Next Hop Database. This must match the memory type used by the microblock.
nhdbChannel	The channel number where the microblock accesses the Next Hop Table.

23.1.2 Macros

Route Table Manager macros are described in this section. [Table 23-2](#) summarizes the RTM macros.

Table 23-2. Route Table Manager Macros

Name	Description
IX_CC_RTMV4_DUMP_ROUTE_SIZE()	Calculates the minimum size of a memory block for the Route Table—used for debugging only; returns a text string with the requested information.
IX_CC_RTMV4_DUMP_NEXT_HOP_SIZE	Calculates the minimum size of a memory block for the Next Hop Database—used for debugging only; returns a text string with the requested information.
IX_CC_RTMV4_NHID_NO_ROUTE	Returns a reserved next hop ID value.
IX_CC_RTMV4_L2INDEX_NO_ROUTE	Returns a reserved L2 index value.

23.1.2.1 IX_CC_RTMV4_DUMP_ROUTE_SIZE()

A macro for calculating the minimum size of a memory block, which is useful for dumping all the routes in the Route Table Manager. Used for debugging only—returns a text string with the requested information.

C Syntax

```
IX_CC_RTMV4_DUMP_ROUTE_SIZE(numberOfRoutes)
```

Input

numberOfRoutes	The number of routes in the Route Table Manager, obtained by calling <code>ix_cc_rtmv4_get_statistics()</code> .
----------------	--

Output/Returns

Return Value	The number of bytes of memory required for <code>ix_cc_rtmv4_dump_routes()</code> .
--------------	---

23.1.2.2 IX_CC_RTMV4_DUMP_NEXT_HOP_SIZE

A macro for calculating the minimum size of a memory block useful for dumping all the next hops in the Route Table Manager. Used for debugging only—returns a text string with the requested information.

C Syntax

```
IX_CC_RTMV4_DUMP_NEXT_HOP_SIZE(numberOfNextHops)
```

Input

numberOfNextHops	The number of next hops in the Route Table Manager, obtained by calling <code>ix_cc_rtmv4_get_statistics()</code> .
------------------	---

Return

The number of bytes of memory required for `ix_cc_rtmv4_dump_next_hops()`.

23.1.2.3 IX_CC_RTMV4_NHID_NO_ROUTE

The Route Table Manager provides a macro for a reserved next hop ID value. See [Section 23.2.3](#), “`ix_cc_rtmv4_add_next_hop()`” for more information.

Value

```
0xFFFFFFFF
```

23.1.2.4 IX_CC_RTMV4_L2INDEX_NO_ROUTE

The Route Table Manager provides a macro providing a reserved L2Index value. See `ix_cc_rtmv4_lookup()` for more information.

Value

0x0000FFFF

23.2 Core Component Infrastructure API

Table 23-3 shows the Route Table Manager core component infrastructure API.

Table 23-3. Route Table Manager Core Component Infrastructure API

Name	Description
<code>ix_cc_rtmv4_init()</code>	Creates a new Route Table Manager for the calling application.
<code>ix_cc_rtmv4_fini()</code>	Destroys an existing Route Table Manager, freeing all previously allocated resources.
<code>ix_cc_rtmv4_add_next_hop()</code>	Adds next hops to the Route Table Manager's next hop database.
<code>ix_cc_rtmv4_delete_next_hop()</code>	Removes a next hop from the Route Table Manager provided that no routes currently refer to the next hop.
<code>ix_cc_rtmv4_update_next_hop()</code>	Updates the information contained in a next hop.
<code>ix_cc_rtmv4_get_next_hop()</code>	Retrieves a structure containing a copy of the information originally given through <code>ix_cc_rtmv4_add_next_hop()</code> .
<code>ix_cc_rtmv4_set_mtu()</code>	Updates the maximum transmission unit for a given next hop.
<code>ix_cc_rtmv4_set_flags()</code>	Updates the flags field for a given next hop.
<code>ix_cc_rtmv4_add_route()</code>	Adds routes, relating a range of network addresses to a next hop.
<code>ix_cc_rtmv4_update_route()</code>	Updates routes, relating an existing range of network addresses to a different next hop.
<code>ix_cc_rtmv4_delete_route()</code>	Deletes a route.
<code>ix_cc_rtmv4_lookup()</code>	Looks up routing information for a given IP address.
<code>ix_cc_rtmv4_dump_next_hops()</code>	Prints a list of all next hops to a block of memory—used for debugging purposes.
<code>ix_cc_rtmv4_dump_routes()</code>	Prints a list of all routes to a block of memory—used for debugging purposes.
<code>ix_cc_rtmv4_purge()</code>	Removes all routes and next hops.
<code>ix_cc_rtmv4_purge_routes()</code>	Removes all routes.
<code>ix_cc_rtmv4_get_symbols()</code>	Retrieves symbols to patch to the microengines.
<code>ix_cc_rtmv4_get_statistics()</code>	Retrieves statistics of the Route Table Manager.

23.2.1 ix_cc_rtmv4_init()

Creates a new Route Table Manager for the calling application. The calling application must initialize the Route Table Manager before performing any other actions.

C Syntax

```
ix_error ix_cc_rtmv4_init(
    ix_cc_rtmv4_config *arg_pConfig,
    ix_cc_rtmv4 *arg_phRtm);
```

Input

<code>arg_pConfig</code>	Configuration information used to create the Route Table Manager and which indicates the amount and type of memory to use.
--------------------------	--

Output/Returns

<code>arg_phRtm</code>	A handle to the newly created Route Table Manager.
------------------------	--

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • A valid <code>ix_error</code>—the operation failed.
--------------	---

Error Types

<code>IX_CC_ERROR_RANGE</code>	One of the values in <code>arg_pConfig</code> was out of range.
<code>IX_CC_ERROR_NULL</code>	A <code>NULL</code> value was passed in for a required parameter.
<code>IX_CC_ERROR_ALIGN</code>	A pointer was not properly aligned.
<code>IX_CC_RTMV4_ERROR_LKUP_UNAVAILABLE</code>	The chosen lookup type is not available in the current environment.
<code>IX_CC_RTMV4_ERROR_CREATE_NHDB</code>	The Route Table Manager failed to create the next hop database.
<code>IX_CC_RTMV4_ERROR_INIT_LKUP</code>	The Route Table Manager failed to initialize the Lookup library.
<code>IX_CC_RTMV4_ERROR_CREATE_TABLE</code>	The Route Table Manager failed to create the route table.
<code>IX_CC_ERROR_OOM_SYSTEM</code>	The Route Table Manager could not allocate system memory for initialization.

23.2.2 ix_cc_rtmv4_fini()

Destroys an existing Route Table Manager, freeing all previously allocated resources. The calling application must not finalize the Route Table Manager until all microblocks have been disabled. After returning, the handle `arg_hRtm` is no longer valid.

C Syntax

```
ix_error ix_cc_rtmv4_fini (  
    ix_cc_rtmv4 arg_hRtm);
```

Input

<code>arg_hRtm</code>	The handle to the Route Table Manager, as returned by <code>ix_cc_rtmv4_init()</code> .
-----------------------	---

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none">• <code>IX_SUCCESS</code>—the operation succeeded.• A valid <code>ix_error</code>—the operation failed.
--------------	--

Error Types

<code>IX_CC_ERROR_NULL</code>	A <code>NULL</code> value was passed in for a required parameter.
<code>IX_CC_ERROR_ALIGN</code>	A pointer was not properly aligned.
<code>IX_CC_ERROR_INTERNAL</code>	The Route Table Manager signaled an internal error while freeing resources.

23.2.3 `ix_cc_rtmv4_add_next_hop()`

Adds next hops to the Route Table Manager's next hop database. Typically, clients would do this any time a new router is detected on a network. Once a next hop is entered, any number of routes may be added referring to that next hop.

Next hops are uniquely identified by their next hop ID, as provided in the next hop information structure. Any attempt to add a duplicate next hop ID results in an error.

A next hop value of `IX_CC_RTMV4_NHID_NO_ROUTE` is special. The calling application is prevented from adding, updating, or deleting this next hop ID value using `ix_cc_rtmv4_add_next_hop()` or `ix_cc_rtmv4_delete_next_hop()`.

C Syntax

```
ix_error ix_cc_rtmv4_add_next_hop(
    ix_cc_rtmv4 arg_hRtm,
    ix_cc_rtmv4_nhid arg_NextHopID,
    ix_cc_rtmv4_next_hop_info *arg_pNextHopInfo);
```

Input

<code>arg_hRtm</code>	A handle to the Route Table Manager as returned by <code>ix_cc_rtmv4_init()</code> .
<code>arg_NextHopID</code>	The next hop ID used to identify this next hop value in the database.
<code>arg_pNextHopInfo</code>	A pointer to the next hop information structure. See <code>ix_cc_rtmv4_next_hop_info</code> .

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. A valid <code>ix_error</code>—the operation failed.
--------------	---

Error Types

<code>IX_CC_ERROR_NULL</code>	A <code>NULL</code> value was passed in for a required parameter.
<code>IX_CC_ERROR_ALIGN</code>	A pointer was not properly aligned.
<code>IX_CC_ERROR_DUPLICATE_ENTRY</code>	The next hop ID is already in use—the next hop information is identical.
<code>IX_CC_ERROR_CONFLICTING_ENTRY</code>	The next hop ID is already in use—the next hop information is different.

Error Types (Continued)

<code>IX_CC_ERROR_FULL</code>	The route table manager is full and cannot add another next hop.
<code>IX_CC_RTMV4_ERROR_INVALID_NH_INFO</code>	One or more of the next hop fields in the <code>ix_cc_rtmv4_next_hop_info</code> data structure was invalid.

23.2.4 `ix_cc_rtmv4_delete_next_hop()`

Removes a next hop from the Route Table Manager provided that no routes currently refer to the next hop. If a calling application wishes to remove a next hop, it must first use `ix_cc_rtmv4_delete_route()` to remove all routes referring to the next hop, then use `ix_cc_rtmv4_delete_next_hop()` to remove the next hop. This may require the client to maintain a complete list of routes associated with next hops. A client would typically remove a next hop when a peer router is removed from the network.

This call fails if any route, including the default route, uses the indicated `nextHopID`.

A next hop value of `IX_CC_RTMV4_NHID_NO_ROUTE` is special and the calling application is prevented from adding it using `ix_cc_rtmv4_add_next_hop()`, deleting it using `ix_cc_rtmv4_delete_next_hop()`, or changing it using `ix_cc_rtmv4_update_next_hop()`.

C Syntax

```
ix_error ix_cc_rtmv4_delete_next_hop (
    ix_cc_rtmv4 arg_hRtm,
    ix_cc_rtmv4_nhid arg_NextHopID);
```

Input

<code>arg_hRtm</code>	A handle to the Route Table Manager as returned by <code>ix_cc_rtmv4_init()</code> .
<code>arg_NextHopID</code>	The identifier of the next hop to be removed.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. A valid <code>ix_error</code>—the operation failed.
--------------	---

Error Types

<code>IX_CC_ERROR_NULL</code>	A NULL value was passed in for a required parameter.
-------------------------------	--

Error Types (Continued)

<code>IX_CC_ERROR_ALIGN</code>	A pointer was not properly aligned.
<code>IX_CC_ERROR_ENTRY_NOT_FOUND</code>	The next hop ID could not be found in the next hop Database.
<code>IX_CC_RTMV4_ERROR_NHID_IN_USE</code>	The next hop ID is in use by one or more routes—the next hop ID was not deleted.
<code>IX_CC_ERROR_INTERNAL</code>	An internal error has occurred, probably caused by corrupted state.

23.2.5 `ix_cc_rtmv4_update_next_hop()`

Updates the information contained in a next hop. The update operation requires that the next hop already exists—next hops cannot be created using the update operation, use `ix_cc_rtmv4_add_next_hop()` to add next hops.

A next hop value of `IX_CC_RTMV4_NHID_NO_ROUTE` is special and the calling application is prevented from adding it using `ix_cc_rtmv4_add_next_hop()`, deleting it using `ix_cc_rtmv4_delete_next_hop()`, or changing it using `ix_cc_rtmv4_update_next_hop()`.

C Syntax

```
ix_error ix_cc_rtmv4_update_next_hop (
    ix_cc_rtmv4 arg_hRtm,
    ix_cc_rtmv4_nhid arg_NextHopID,
    ix_cc_rtmv4_next_hop_info *arg_pNextHopInfo);
```

Input

<code>arg_hRtm</code>	A handle to the RTM, as returned by <code>ix_cc_rtmv4_init()</code> .
<code>arg_NextHopID</code>	The identifier of the Next Hop to be updated.
<code>arg_pNextHopInfo</code>	A pointer to the new Next Hop Information structure.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. A valid <code>ix_error</code>—the operation failed.
--------------	---

Error Types

<code>IX_CC_ERROR_NULL</code>	A <code>NULL</code> value was passed in for a required parameter.
<code>IX_CC_ERROR_ALIGN</code>	A pointer was not properly aligned.
<code>IX_CC_ERROR_ENTRY_NOT_FOUND</code>	The next hop ID could not be found in the next hop database.
<code>IX_CC_RTMV4_ERROR_INVALID_NH_INFO</code>	One or more of the next hop fields was invalid.
<code>IX_CC_ERROR_INTERNAL</code>	An internal error has occurred, probably caused by corrupted state.

23.2.6 `ix_cc_rtmv4_get_next_hop()`

Retrieves a structure containing a copy of the information originally given through `ix_cc_rtmv4_add_next_hop()`.

C Syntax

```
ix_error ix_cc_rtmv4_get_next_hop(
    ix_cc_rtmv4 arg_hRtm,
    ix_cc_rtmv4_nhid arg_NextHopID,
    ix_cc_rtmv4_next_hop_info *arg_pNextHopInfo);
```

Input

<code>arg_hRtm</code>	A handle to the Route Table Manager, as returned by <code>ix_cc_rtmv4_init()</code> .
<code>arg_NextHopID</code>	The identifier of the next hop to be retrieved.

Output/Returns

<code>arg_pNextHopInfo</code>	The data structure provided by the calling application to contain the results of this operation.
Return Value	<p>Returns a valid <code>ix_error</code>.</p> <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. A valid <code>ix_error</code>—the operation failed.

Error Types

<code>IX_CC_ERROR_NULL</code>	A NULL value was passed in for a required parameter.
<code>IX_CC_ERROR_ALIGN</code>	A pointer was not properly aligned.
<code>IX_CC_ERROR_ENTRY_NOT_FOUND</code>	The next hop ID could not be found in the next hop database.
<code>IX_CC_ERROR_INTERNAL</code>	An internal error has occurred, probably caused by corrupted state.

23.2.7 `ix_cc_rtmv4_set_mtu()`

Updates the maximum transmission unit for a given next hop. If the default route matches the `arg_NextHopID`, the default route's maximum transmission unit is affected as well. Because the Route Table Manager does not interpret the meaning of the maximum transmission unit it does not validate the value.

Note: Only 16 bits are stored in shared memory. This function returns an error if the MTU value is too large to store. However, the function does *not* check the logical accuracy of the value—is there a value of 1500 for Ethernet, a value of 9000 for ATM, and so on.

C Syntax

```
ix_error ix_cc_rtmv4_set_mtu(
    ix_cc_rtmv4 arg_hRtm,
    ix_cc_rtmv4_nhid arg_NextHopID,
    ix_uint32 arg_Mtu);
```

Input

<code>arg_hRtm</code>	A handle to the Route Table Manager as returned by <code>ix_cc_rtmv4_init()</code> .
<code>arg_NextHopID</code>	The identifier of the next hop to be modified.
<code>arg_Mtu</code>	A new maximum transmission unit for the next hop.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. A valid <code>ix_error</code>—the operation failed.
--------------	---

Error Types

<code>IX_CC_ERROR_NULL</code>	A NULL value was passed in for a required parameter.
<code>IX_CC_ERROR_ALIGN</code>	A pointer was not properly aligned.
<code>IX_CC_ERROR_ENTRY_NOT_FOUND</code>	The next hop ID could not be found in the next hop database
<code>IX_CC_RTMV4_ERROR_INVALID_NH_INFO</code>	The maximum transmission unit was invalid—that is, the value could not be stored in 16 bits of shared memory. NOTE: The logical accuracy of the value is <i>not</i> checked—see this function's description.

23.2.8 `ix_cc_rtmv4_set_flags()`

Updates the flags field for a given next hop. If the default route matches the `arg_NextHopID`, the default route's flags are affected as well. The meaning of each flag is determined by the caller.

Note: The function dedicates eight bits of shared memory to this value. If the application attempts to set a value larger than eight bits an error is returned. Because the meaning of these flags is determined by the calling application no other checks of accuracy are performed.

C Syntax

```
ix_error ix_cc_rtmv4_set_flags (
    ix_cc_rtmv4 arg_hRtm,
    ix_cc_rtmv4_nhid arg_NextHopID,
    ix_uint32 arg_Flags);
```

Input

<code>arg_hRtm</code>	A handle to the Route Table Manager, as returned by <code>ix_cc_rtmv4_init()</code> .
<code>arg_NextHopID</code>	The identifier of the next hop to be modified.
<code>arg_Flags</code>	The new flags for the next hop.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. A valid <code>ix_error</code>—the operation failed.
--------------	---

Error Types

<code>IX_CC_ERROR_NULL</code>	A <code>NULL</code> value was passed in for a required parameter.
<code>IX_CC_ERROR_ALIGN</code>	A pointer was not properly aligned.
<code>IX_CC_ERROR_ENTRY_NOT_FOUND</code>	The next hop ID could not be found in the next hop database.
<code>IX_CC_RTMV4_ERROR_INVALID_NH_INFO</code>	An invalid value was passed in for <code>arg_Flags</code> .

23.2.9 `ix_cc_rtmv4_add_route()`

Adds routes, relating a range of network addresses to a next hop. In order to add a route, the calling application must first add the next hop ID and its corresponding data into the Route Table Manager, using `ix_cc_rtmv4_add_next_hop()`.

After adding a route, lookup operations from core components through the Route Table Manager or lookup operations from microblocks use the newly added route.

Adding a duplicate route to the same or different next hop is an error. Instead, use `ix_cc_rtmv4_update_route()` to modify an existing route.

The client may add a default route by setting the `netAddr` and `netMask` to zero. As always, the `nextHopID` must refer to a valid next hop. Setting the default route is a special case and the caller need not remove an existing default route before setting the default route to a new next hop.

C Syntax

```
ix_error ix_cc_rtmv4_add_route (
    ix_cc_rtmv4 arg_hRtm,
    ix_uint32 arg_NetAddr,
    ix_uint32 arg_NetMask,
    ix_cc_rtmv4_nhid arg_NextHopID);
```

Input

<code>arg_hRtm</code>	A handle to the Route Table Manager, as returned by <code>ix_cc_rtmv4_init()</code> .
<code>arg_NetAddr</code>	The network address identifying the route.
<code>arg_NetMask</code>	The network mask for the route.
<code>arg_NextHopID</code>	The identifier of the next hop to be used for the route.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • A valid <code>ix_error</code>—the operation failed.
--------------	---

Error Types

<code>IX_CC_ERROR_NULL</code>	A NULL value was passed in for a required parameter.
<code>IX_CC_ERROR_ALIGN</code>	A pointer was not properly aligned.
<code>IX_CC_ERROR_ENTRY_NOT_FOUND</code>	The next hop ID could not be found in the next hop database.
<code>IX_CC_ERROR_DUPLICATE_ENTRY</code>	The route was already added to this next hop ID.
<code>IX_CC_ERROR_CONFLICTING_ENTRY</code>	The route was already added to another next hop ID.
<code>IX_CC_RTMV4_ERROR_INVALID_MASK</code>	Invalid network mask value was passed into <code>arg_NetMask</code> .
<code>IX_CC_ERROR_FULL</code>	The Route Table is full. The Route Table Manager cannot add another route.
<code>IX_CC_ERROR_INTERNAL</code>	An internal error occurred, probably caused by corrupted state within the Route Table Manager.

23.2.10 `ix_cc_rtmv4_update_route()`

Updates routes, relating an existing range of network addresses to a different next hop. In order to update a route, the client must first add the next hop ID and its corresponding data into the Route Table Manager, using `ix_cc_rtmv4_add_next_hop()`.

After adding a route, lookup operations, either from core components through the Route Table Manager or from microblocks, use the modified route.

The calling application should not use this update operation to modify the default route. Use `ix_cc_rtmv4_add_route()` or `ix_cc_rtmv4_delete_route()` instead.

C Syntax

```
ix_error ix_cc_rtmv4_update_route(
    ix_cc_rtmv4 arg_hRtm,
    ix_uint32 arg_NetAddr,
    ix_uint32 arg_NetMask,
    ix_cc_rtmv4_nhid arg_NextHopID);
```

Input

<code>arg_hRtm</code>	A handle to the Route Table Manager—as returned by <code>ix_cc_rtmv4_init()</code> .
<code>arg_NetAddr</code>	The network address identifying the route.
<code>arg_NetMask</code>	The network mask for the route.
<code>arg_NextHopID</code>	The identifier of the next hop to be used to identify the route to the Route Table Manager.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none">• <code>IX_SUCCESS</code>—the operation succeeded.• A valid <code>ix_error</code>—the operation failed.
--------------	--

Error Types

<code>IX_CC_ERROR_NULL</code>	A NULL value was passed in for a required parameter.
<code>IX_CC_ERROR_ALIGN</code>	A pointer was not properly aligned.
<code>IX_CC_ERROR_ENTRY_NOT_FOUND</code>	The next hop ID could not be found in the next hop database.
<code>IX_CC_ERROR_CONFLICTING_ENTRY</code>	The calling application attempted to update the default route.
<code>IX_CC_RTMV4_ERROR_INVALID_MASK</code>	Invalid network mask value was passed into <code>arg_NetMask</code> .
<code>IX_CC_ERROR_INTERNAL</code>	An internal error occurred, probably caused by corrupted state within the Route Table Manager.

23.2.11 ix_cc_rtmv4_delete_route()

Deletes a route. The calling application is responsible for removing the next hop. Deleting the next hop is not an implicit operation of `ix_cc_rtmv4_delete_route()`.

The calling application may remove the default route by setting the `arg_NetAddr` and `arg_NetMask` to zero. Removing the default route is a special case and the calling application may do so at any time—even if the default route did not previously exist.

C Syntax

```
ix_error ix_cc_rtmv4_delete_route(
    ix_cc_rtmv4 arg_hRtm,
    ix_uint32 arg_NetAddr,
    ix_uint32 arg_NetMask,
);
```

Input

<code>arg_hRtm</code>	A handle to the Route Table Manager as returned by <code>ix_cc_rtmv4_init()</code> .
<code>arg_NetAddr</code>	The network address identifying the route.
<code>arg_NetMask</code>	The subnet mask for the route.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. A valid <code>ix_error</code>—the operation failed.
--------------	---

Error Types

<code>IX_CC_ERROR_NULL</code>	A NULL value was passed in for a required parameter.
<code>IX_CC_ERROR_ALIGN</code>	A pointer was not properly aligned.
<code>IX_CC_ERROR_ENTRY_NOT_FOUND</code>	The network address and network mask pair do not exist in the Route Table Manager.
<code>IX_CC_RTMV4_ERROR_INVALID_MASK</code>	The network mask was invalid.
<code>IX_CC_ERROR_INTERNAL</code>	An internal error occurred, probably caused by corrupted state within the Route Table Manager.

23.2.12 ix_cc_rtmv4_get_route

Retrieves the NextHopID of the specified route.

C Syntax

```
ix_error ix_cc_rtmv4_get_route (
    ix_cc_rtmv4 arg_hRtm,
    ix_uint32 arg_NetAddr,
    ix_uint32 arg_NetMask,
    ix_cc_rtmv4_nhid *arg_pNextHopID
);
```

Input

arg_hRtm	A handle to the RTM, as returned by ix_cc_rtmv4_init().
arg_NetAddr	Network address identifying the route.
arg_NetMask	Subnet mask for the route.

Output/Returns

arg_pNextHopID	The next hop identifier, previously set with ix_cc_rtmv4_add_route() or ix_cc_rtmv4_update_route() .
Return Value	<p>Returns a valid ix_error.</p> <ul style="list-style-type: none"> IX_SUCCESS—returned if the operation succeeds. A valid ix_error—returned if the operation fails

Error Codes

IX_CC_ERROR_NULL	A required parameter was NULL.
IX_CC_ERROR_ALIGN	A pointer was not properly aligned.
IX_CC_ERROR_ENTRY_NOT_FOUND	The network address / network mask pair do not exist in the RTM
IX_CC_ERROR_INTERNAL	An internal error occurred, probably caused by corrupted state within the RTM
IX_CC_RTMV4_ERROR_INVALID_MASK	Invalid network mask

23.2.13 ix_cc_rtmv4_lookup()

Looks up routing information for a given IP address. The Route Table Manager uses a longest-prefix match (LPM) algorithm to match the IP address with a network—as given by calling `ix_cc_rtmv4_add_route()`—providing the information needed to forward the packet to the appropriate next hop—that is, the corresponding next hop information structure.

This function is the workhorse of the Route Table Manager and is expected to be the most-used entry point—conceivably every packet arriving on the Intel XScale® core requires a lookup. This function is therefore designed with efficiency in mind.

If the `l2Index` field in `pNextHopInfo` is `IX_CC_RTMV4_NHID_NO_ROUTE`, all other fields are invalid and are ignored—the IP address has no route.

C Syntax

```
ix_error ix_cc_rtmv4_lookup(
    ix_cc_rtmv4 arg_hRtm,
    ix_uint32 arg_IpAddr,
    ix_cc_rtmv4_next_hop_info *arg_pNextHopInfo);
```

Input

<code>arg_hRtm</code>	A handle to the Route Table Manager as returned by <code>ix_cc_rtmv4_init()</code> .
<code>arg_IpAddr</code>	The IP address to be looked up.

Output/Returns

<code>arg_pNextHopInfo</code>	The results of this function are returned in this structure passed in by the calling application.
Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. A valid <code>ix_error</code>—the operation failed.

Error Types

<code>IX_CC_ERROR_NULL</code>	A NULL value was passed in for a required parameter.
<code>IX_CC_ERROR_ALIGN</code>	A pointer was not properly aligned.
<code>IX_CC_ERROR_INTERNAL</code>	An internal error occurred, probably caused by corrupted state within the Route Table Manager.

23.2.14 ix_cc_rtmv4_dump_next_hops()

Prints, as a human-readable form, all next hops to a block of memory—used for debugging purposes. The memory is provided by the calling application along with the size of the memory block in bytes. The calling application should use `IX_CC_RTMV4_DUMP_NEXT_HOP_SIZE` to determine how many bytes to allocate.

Note: If the function returns `IX_CC_ERROR_FULL`, the error should be considered non-fatal. The caller may allocate more memory and call the function again.

C Syntax

```
ix_error ix_cc_rtmv4_dump_next_hops(
    ix_cc_rtmv4 arg_hRtm,
    char *arg_pBuffer,
    ix_uint32 arg_BufferSize);
```

Input

<code>arg_hRtm</code>	A handle to the Route Table Manager, as returned by <code>ix_cc_rtmv4_init()</code> .
<code>arg_pBuffer</code>	A pointer to a block of memory where the list of next hops is to be stored.
<code>arg_BufferSize</code>	The size in bytes of the buffer in host-byte order.

Output/Returns

<code>arg_pBuffer</code>	A human-readable ASCII-Z string listing the next hops.
Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. A valid <code>ix_error</code>—the operation failed.

Error Types

<code>IX_CC_ERROR_NULL</code>	A NULL value was passed in for a required parameter.
<code>IX_CC_ERROR_ALIGN</code>	A pointer was not properly aligned.
<code>IX_CC_ERROR_FULL</code>	The buffer was too small to contain all the information. NOTE: The calling application may choose to invoke this operation a second time after allocating a larger buffer.
<code>IX_CC_ERROR_INTERNAL</code>	An internal error occurred, probably caused by corrupted state within the Route Table Manager.
<code>IX_CC_ERROR_RANGE</code>	The value of <code>arg_BufferSize</code> was too small.

23.2.15 ix_cc_rtmv4_dump_routes()

Prints, as a human-readable string, a list of all routes to a block of memory—used for debugging purposes. The memory is provided by the caller, along with the size of the memory block in bytes. The caller should use `IX_CC_RTMV4_DUMP_ROUTE_SIZE()` to determine how many bytes to allocate.

Note: If the function returns `IX_CC_ERROR_FULL` the error should be considered non-fatal. The caller may allocate more memory and call the function again.

C Syntax

```
ix_error ix_cc_rtmv4_dump_routes(
    ix_cc_rtmv4 arg_hRtm,
    char *arg_pBuffer,
    ix_uint32 arg_BufferSize);
```

Input

<code>arg_hRtm</code>	A handle to the Route Table Manager as returned by <code>ix_cc_rtmv4_init()</code> .
<code>arg_pBuffer</code>	A pointer to a block of memory where the next hops are to be stored.
<code>arg_BufferSize</code>	The size in bytes of the buffer in host-byte order.

Output/Returns

<code>arg_pBuffer</code>	A pointer to the list of routes returned as a human-readable ASCII-Z string.
Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. A valid <code>ix_error</code>—the operation failed.

Error Types

<code>IX_CC_ERROR_NULL</code>	A NULL value was passed in for a required parameter.
<code>IX_CC_ERROR_ALIGN</code>	A pointer was not properly aligned.
<code>IX_CC_ERROR_FULL</code>	The buffer was too small to contain all the information. NOTE: The calling application may choose to invoke this operation a second time after allocating a larger buffer.
<code>IX_CC_ERROR_INTERNAL</code>	An internal error occurred, probably caused by corrupted state within the Route Table Manager.

Error Types

IX_CC_ERROR_RANGE	The value of <code>arg_BufferSize</code> was too small.
-------------------	---

23.2.16 ix_cc_rtmv4_purge()

Removes all routes and next hops. Typically, this would only be called for debugging purposes.

C Syntax

```
ix_error ix_cc_rtmv4_purge (
    ix_cc_rtmv4 arg_hRtm);
```

Input

<code>arg_hRtm</code>	A handle to the Route Table Manager as returned by <code>ix_cc_rtmv4_init()</code> .
-----------------------	--

Output/Returns

Return Value	<p>Returns a valid <code>ix_error</code>.</p> <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • A valid <code>ix_error</code>—the operation failed.
--------------	---

Error Types

IX_CC_ERROR_NULL	A <code>NULL</code> value was passed in for a required parameter.
IX_CC_ERROR_ALIGN	A pointer was not properly aligned.
IX_CC_ERROR_INTERNAL	An internal error occurred, probably caused by corrupted state within the Route Table Manager.

23.2.17 ix_cc_rtmv4_purge_routes()

This function removes all routes. Typically, this would only be called for debugging purposes.

C Syntax

```
ix_error ix_cc_rtmv4_purge_routes (  
    ix_cc_rtmv4 arg_hRtm);
```

Input

arg_hRtm	A handle to the Route Table Manager as returned by ix_cc_rtmv4_init() .
----------	---

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none">• <code>IX_SUCCESS</code>—the operation succeeded.• A valid <code>ix_error</code>—the operation failed.
--------------	--

Error Types

<code>IX_CC_ERROR_NULL</code>	A NULL value was passed in for a required parameter.
<code>IX_CC_ERROR_ALIGN</code>	A pointer was not properly aligned.
<code>IX_CC_ERROR_INTERNAL</code>	An internal error occurred, probably caused by corrupted state within the Route Table Manager.

23.2.18 `ix_cc_rtmv4_get_symbols()`

This function retrieves symbols to patch to the microengines.

C Syntax

```
ix_error ix_cc_rtmv4_get_symbols(
    ix_cc_rtmv4 arg_hRtm,
    ix_cc_rtmv4_symbols *arg_pSymbols);
```

Input

<code>arg_hRtm</code>	A handle to the Route Table Manager as returned by <code>ix_cc_rtmv4_init()</code> .
-----------------------	--

Output/Returns

<code>arg_pSymbols</code>	A structure containing values representing the microengine symbols the calling application is going to patch.
Return Value	<p>Returns a valid <code>ix_error</code>.</p> <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • A valid <code>ix_error</code>—the operation failed.

Error Types

<code>IX_CC_ERROR_NULL</code>	A <code>NULL</code> value was passed in for a required parameter.
<code>IX_CC_ERROR_ALIGN</code>	A pointer was not properly aligned.
<code>IX_CC_ERROR_INTERNAL</code>	An internal error occurred, probably caused by corrupted state within the Route Table Manager.

23.2.19 ix_cc_rtmv4_get_statistics()

This function retrieves statistics of the Route Table Manager.

C Syntax

```
ix_error ix_cc_rtmv4_get_statistics(
    ix_cc_rtmv4 arg_hRtm,
    ix_cc_rtmv4_statistics *arg_pStatistics);
```

Input

arg_hRtm	A handle to the Route Table Manager as returned by <code>ix_cc_rtmv4_init()</code> .
----------	--

Output/Returns

arg_pStatistics	A calling-application-provided structure used to return the requested statistics.
Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • A valid <code>ix_error</code>—the operation failed.

Error Types

<code>IX_CC_ERROR_NULL</code>	A NULL value was passed in for a required parameter.
<code>IX_CC_ERROR_ALIGN</code>	A pointer was not properly aligned.
<code>IX_CC_ERROR_INTERNAL</code>	An internal error occurred, probably caused by corrupted state within the Route Table Manager.

The Route Table Manager (RTMv6) stores and retrieves data on behalf of the IPv6 Forwarder core component. In addition, the RTMv6 stores the data in a format consistent with the requirements of the IPv6 Forwarder microblock. RTMv6 uses the Lookup library API to access the route tables for IPv6. To manage next hops, RTMv6 uses a Next Hop Database (NHDB). The RTMv6 exposes an API allowing the IPv6 core component to add to, remove from, or lookup items in the route table. This API is implemented as a library supporting a single client.

This implementation of RTMv6 is specific to the IP, version 6. Because an RTMv6 designed for IPv6 would require a different interface (for example, 128-bit parameters instead of 32-bit), it exposes its entry points in such a way as to indicate that it only deals with IPv6 data types and algorithms. This avoids naming conflicts with an RTMv6 designed for IPv4. The RTM is implemented as a library of functions providing services to the IPv4 Forwarder. For details on the Route Table Manager for IPv4 refer to [Chapter 23, “Route Table Manager”](#).

This section describes the interfaces exposed to the application calling the RTM, explained from the perspective of the calling application.

The RTMv6 is implemented as a library of functions providing services to the IPv6 Forwarder. This section describes the interfaces exposed to the client of the RTMv6, explained from the perspective of the calling application.

For complete details see [Chapter 61, “Route Table Manager for IPV6 Core Component”](#) of the *Intel® Internet Exchange Architecture Software Building Blocks Developer’s Manual*

24.1 Data Structures, Types, and Macros

Table 24-1 lists the RTMv6 data structures, types and macros.

Table 24-1. RTMv6 Data Structures, Types and Macros

Data Structures, Types, macros	Description
<code>ix_cc_rtmv6</code>	Opaque handle to the RTM
<code>ix_cc_rtmv6_nhid</code>	Typedef to an intrinsic data type; represents a Next Hop ID
<code>ix_cc_rtmv6_next_hop_info</code>	Describes a Next Hop
<code>ix_cc_rtmv6_symbols</code>	Defines a structure containing values useful in the microengines
<code>ix_cc_rtmv6_statistics</code>	Retrieves statistics about the RTM
<code>ix_cc_rtmv6_lkup_type</code>	Enumeration of the supported algorithm types, TCAM and Software
<code>ix_cc_rtmv6_mem_type</code>	Enumeration of the supported memory types, SRAM and SDRAM

Table 24-1. RTMv6 Data Structures, Types and Macros

Data Structures, Types, macros	Description
IX_CC_RTMV6_DUMP_ROUTE_SIZE	Calculates the minimum size of a memory block useful for dumping all the routes in the RTM.
IX_CC_RTMV6_DUMP_NEXT_HOP_SIZE	Calculates the minimum size of a memory block useful for dumping all the next hops in the RTM
IX_CC_RTMV6_NHID_NO_ROUTE	Provides a macro for a reserved Next Hop ID value

24.1.1 [ix_cc_rtmv6](#)

Opaque handle to the RTM, obtained by calling [ix_cc_rtmv6_init\(\)](#).

C Syntax

```
typedef struct ix_s_cc_rtmv6* ix_cc_rtmv6;
```

24.1.2 [ix_cc_rtmv6_nhid](#)

Typedef to an intrinsic data type; represents a Next Hop ID. Applications at the highest levels allocate and assign the NHID values. The value is stored in network byte order.

C Syntax

```
typedef ix_uint32 ix_cc_rtmv6_nhid;
```

24.1.3 [ix_cc_rtmv6_next_hop_info](#)

Though this data is identical in meaning to the Next Hop Information used by the IPv6 microblock [BLDBLK], the structure used to store the data is less restrictive. The RTM will internally pack the Next Hop data into the format the microengines expect.

- adding a Next Hop (see [ix_cc_rtmv6_add_next_hop\(\)](#)),
- retrieving Next Hop Information (see [ix_cc_rtmv6_get_next_hop\(\)](#)),
- looking up a Next Hop by IPv6 address (see [ix_cc_rtmv6_lookup\(\)](#)).

It is anticipated that the IPv6 Forwarder will expose this structure in its interface, requiring Forwarding Plane Module to understand the structure. It may be appropriate to move the definition to a more central location later. Doing so will affect the RTM interfaces [ix_cc_rtmv6_add_next_hop\(\)](#), [ix_cc_rtmv6_get_next_hop\(\)](#) and [ix_cc_rtmv6_lookup\(\)](#).

The structure for [ix_cc_rtmv6_next_hop_info](#) is similar to [ix_cc_rtmv4_next_hop_info](#) (see [Section 23.1.1.3, “ix_cc_rtmv4_next_hop_info” on page 486](#)) for details. Note that the structure contains a field for the IPv4 address, which needs to be cleared out before calling any of the RTMv6 APIs (does not need to be defined).

24.1.4 ix_cc_rtmv6_symbols

The RTM defines a structure containing values useful in the microengines. This structure is obtained by calling `ix_cc_rtmv6_get_symbols()`. All fields are in network byte order.

C Syntax

```
typedef struct ix_s_cc_rtmv6_symbols {
    ix_uint32 lkupTableCookie;
    ix_uint32 nextHopBase;
} ix_cc_rtmv6_symbols;
```

24.1.5 ix_cc_rtmv6_statistics

The calling application may retrieve statistics about the RTM by calling `ix_cc_rtmv6_get_statistics()`.

C Syntax

```
typedef struct ix_s_cc_rtmv6_statistics {
    ix_uint32 numberOfRoutes;
    ix_uint32 numberOfNextHops;
} ix_cc_rtmv6_statistics;
```

24.1.6 ix_cc_rtmv6_lkup_type

Enumeration of the supported algorithm types, TCAM and Software.

C Syntax

```
typedef enum ix_e_cc_rtmv6_lkup_type {
    IX_CC_RTMV6_LKUP_SOFTWARE,
    IX_CC_RTMV6_LKUP_TCAM,
    IX_CC_RTMV6_LKUP_LAST,
} ix_cc_rtmv6_lkup_type;
```

24.1.7 ix_cc_rtmv6_mem_type

Enumeration of the supported memory types, SRAM and SDRAM.

C Syntax

```
typedef enum ix_e_cc_rtmv6_mem_type {  
    IX_CC_RTMV6_MEM_SDRAM,  
    IX_CC_RTMV6_MEM_SRAM,  
    IX_CC_RTMV6_MEM_LAST  
} ix_cc_rtmv6_mem_type;
```

24.1.8 IX_CC_RTMV6_DUMP_ROUTE_SIZE

Macro for calculating the minimum size of a memory block useful for dumping all the routes in the RTM.

C Syntax

```
IX_CC_RTMV6_DUMP_ROUTE_SIZE(numberOfRoutes)
```

Input

numberOfRoutes	The number of routes in the RTM, obtained by calling <code>ix_cc_rtmv6_get_statistics()</code> .
----------------	--

Output/Returns

Return Value	The number of bytes of memory required for <code>ix_cc_rtmv6_dump_routes()</code> .
--------------	---

24.1.9 IX_CC_RTMV6_DUMP_NEXT_HOP_SIZE

Macro for calculating the minimum size of a memory block useful for dumping all the next hops in the RTM.

C Syntax

```
IX_CC_RTMV6_DUMP_NEXT_HOP_SIZE(numberOfNextHops)
```

Input

numberOfNextHop	The number of next hops in the RTM, obtained by calling <code>ix_cc_rtmv6_get_statistics()</code> .
-----------------	---

Output/Returns

Return Value The number of bytes of memory required for `ix_cc_rtmv6_dump_next_hops()`.

24.1.10 IX_CC_RTMV6_NHID_NO_ROUTE

The RTM provides a macro for a reserved Next Hop ID value. See `ix_cc_rtmv6_lookup()` for more information.

Value = 0x0

24.2 Core Component Infrastructure API

Table 24-2 lists the RTMv6 Core Component Infrastructure API.

Table 24-2. RTMv6 Core Component Infrastructure API

API	Description
<code>ix_cc_rtmv6_init()</code>	Creates a new RTMv6 for the calling application
<code>ix_cc_rtmv6_fini()</code>	Terminates an existing RTMv6 and frees all previously allocated resources
<code>ix_cc_rtmv6_add_next_hop()</code>	Adds Next Hops to the RTMv6's next hop database
<code>ix_cc_rtmv6_delete_next_hop()</code>	Removes a Next Hop from the RTMv6
<code>ix_cc_rtmv6_update_next_hop()</code>	Updates a Next Hop in the RTMv6's next hop database
<code>ix_cc_rtmv6_get_next_hop()</code>	Retrieves a structure containing a copy of the information originally given
<code>ix_cc_rtmv6_set_mtu()</code>	Updates the MTU for a given Next Hop
<code>ix_cc_rtmv6_set_flags()</code>	Updates the Flags field for a given Next Hop
<code>ix_cc_rtmv6_add_route()</code>	Adds routes, relating a range of network addresses to a Next Hop
<code>ix_cc_rtmv6_delete_route()</code>	Deletes a route
<code>ix_cc_rtmv6_update_route()</code>	Updates routes, relating a range of network addresses to a Next Hop
<code>ix_cc_rtmv6_lookup()</code>	Looks up routing information for a given IPv6 address
<code>ix_cc_rtmv6_dump_next_hops()</code>	Prints all Next Hops to a block of memory for debugging purposes
<code>ix_cc_rtmv6_dump_routes()</code>	Prints all routes to a block of memory for debugging purposes
<code>ix_cc_rtmv6_purge()</code>	Removes all routes and next hops
<code>ix_cc_rtmv6_purge_routes()</code>	Removes all routes
<code>ix_cc_rtmv6_get_symbols()</code>	Retrieves symbols to patch to the microengines
<code>ix_cc_rtmv6_get_statistics()</code>	Retrieves statistics of the RTMv6

24.2.1 ix_cc_rtmv6_init()

Creates a new RTMv6 for the calling application. The calling application must initialize the RTMv6 before performing any other actions.

C Syntax

```
ix_error ix_cc_rtmv6_init(
    ix_uint32 routeTableSizeHint,
    ix_cc_rtmv6_lkup_type lkupType,
    ix_cc_rtmv6_mem_type lkupMemType,
    ix_uint32 nhdbSizeHint,
    ix_cc_rtmv6_mem_type nhdbMemType,
    ix_cc_rtmv6 *pRtm);
```

Input

routeTableSizeHint	An approximate value of the routes supported. The quantity indicated is not guaranteed, but may be used to determine the number of resources required.
lkupType	Type of lookup to use. This must match the algorithm used by the microengines.
lkupMemType	Type of memory to use in the lookup algorithm. This must match the memory type used by the microengines.
nhdbSizeHint	An approximate value of the next hops supported. The quantity indicated is not guaranteed, but may be used to determine the number of resources required.
nhdbMemType	Type of memory to use in the Next Hop Database. This must match the memory type used by the microengines.

Output/Returns

*pRtm	Handle to the RTMv6
Return Value	<p>Returns a valid <code>ix_error</code>.</p> <ul style="list-style-type: none"> IX_SUCCESS—the operation succeeded. IX_CC_ERROR_RANGE—operation failed, parameter is out of range. IX_CC_ERROR_NULL—operation failed, null parameter IX_CC_ERROR_ALIGN—operation failed, parameter not aligned as expected IX_CC_RTMV6_ERROR_LKUP_UNAVAILABLE—operation failed, unsupported Lookup type IX_CC_RTMV6_ERROR_CREATE_NHDB—operation failed, failed to create NHDB IX_CC_RTMV6_ERROR_ERROR_INIT_LKUP—operation failed, failed to initialize lookup library.

24.2.2 `ix_cc_rtmv6_fini()`

Terminates an existing RTMv6 and frees all previously allocated resources. The calling application must not finalize the RTMv6 until all microblocks have been disabled. After returning, the RTMv6 handle is no longer valid.

C Syntax

```
ix_error ix_cc_rtmv6_fini (
    ix_cc_rtmv6 rtm);
```

Input

rtm	Handle to the RTMv6, as returned by <code>ix_cc_rtmv6_init()</code> .
-----	---

Output/Returns

Return Value	<p>Returns a valid <code>ix_error</code>.</p> <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • <code>IX_CC_ERROR_INTERNAL</code>—operation failed, RTMV6 internal error. • <code>IX_CC_ERROR_NULL</code>—operation failed, null parameter • <code>IX_CC_ERROR_ALIGN</code>—operation failed, parameter not aligned as expected
--------------	---

24.2.3 `ix_cc_rtmv6_add_next_hop()`

Adds Next Hops to the RTMv6's next hop database. Typically, calling applications would do this any time a new router is detected on a network. Once a Next Hop is entered, any number of routes may be added referring to that Next Hop.

Next Hops are uniquely identified by their NHID, as provided in the Next Hop Information Structure. Any attempt to add a duplicate NHID results in an error.

A next hop value of `IX_CC_RTMV6_NHID_NO_ROUTE` is special and the calling application is prevented from adding or deleting it using `ix_cc_rtmv6_add_next_hop()` or `ix_cc_rtmv6_delete_next_hop()`.

C Syntax

```
ix_error ix_cc_rtmv6_add_next_hop (
    ix_cc_rtmv6 rtm,
    ix_cc_rtmv6_next_hop_info *pNextHopInfo);
```

Input

rtm	Handle to the RTMv6, as returned by <code>ix_cc_rtmv6_init()</code> .
pNextHopInfo	Pointer to the Next Hop Information structure.

Output/Returns

Return Value	<p>Returns a valid <code>ix_error</code>.</p> <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • <code>IX_CC_ERROR_NULL</code>—operation failed, null parameter • <code>IX_CC_ERROR_ALIGN</code>—operation failed, parameter not aligned as expected • <code>IX_CC_ERROR_DUPLICATE_ENTRY</code>—operation failed, parameter is a duplicate. • <code>IX_CC_ERROR_CONFLICTING_ENTRY</code>—operation failed, parameter conflicts with stored entry. • <code>IX_CC_ERROR_FULL</code>—operation failed, table is full. • <code>IX_CC_RTMV6_ERROR_INVALID_NH_INFO</code>—operation failed, invalid next hop info provided.
--------------	---

24.2.4 `ix_cc_rtmv6_delete_next_hop()`

Removes a Next Hop from the RTMv6, provided no routes currently refer to the Next Hop. If a calling application wishes to remove a next hop, it must first use `ix_cc_rtmv6_delete_route()` to remove all routes referring to the next hop, then use `ix_cc_rtmv6_delete_next_hop()` to remove the next hop. This may require the calling application to maintain a complete list of routes associated with next hops. A calling application would typically remove a Next Hop when a peer router is removed from the network.

This call will fail if any route, including the default route, uses the indicated nextHopID.

A next hop value of `IX_CC_RTMV6_NHID_NO_ROUTE` is special and the calling application is prevented from adding or deleting it using `ix_cc_rtmv6_add_next_hop()` or `ix_cc_rtmv6_delete_next_hop()`.

C Syntax

```
ix_error ix_cc_rtmv6_delete_next_hop (
    ix_cc_rtmv6 rtm,
    ix_cc_rtmv6_nhid nextHopID);
```

Input

<code>rtm</code>	Handle to the RTMv6, as returned by <code>ix_cc_rtmv6_init()</code> .
<code>nextHopID</code>	Identifier of the Next Hop to be removed; network byte order.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • <code>IX_CC_ERROR_NULL</code>—operation failed, null parameter • <code>IX_CC_ERROR_ALIGN</code>—operation failed, parameter not aligned as expected • <code>IX_CC_ERROR_ENTRY_NOT_FOUND</code>—operation failed, next hop ID does not exist. • <code>IX_CC_RTMV6_ERROR_NHID_IN_USE</code>—operation failed, next hop ID in use and cannot be deleted • <code>IX_CC_ERROR_INTERNAL</code>—operation failed, RTMv6 internal error.
--------------	--

24.2.5 `ix_cc_rtmv6_update_next_hop()`

Updates a Next Hop in the RTMv6 next hop database. This operation requires the next hop ID to exist in the NHDB. Next hops cannot be created using this operation. A next hop value of `IX_CC_RTMV6_NHID_NO_ROUTE` is reserved and cannot be used as a parameter to this function.

C Syntax

```
ix_error ix_cc_rtmv6_update_next_hop (
    ix_cc_rtmv6 rtm,
    ix_cc_rtmv6_next_hop_info *pNextHopInfo);
```

Input

<code>rtm</code>	Handle to the RTMv6, as returned by <code>ix_cc_rtmv6_init()</code> .
<code>pNextHopInfo</code>	Pointer to the Next Hop Information structure.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • <code>IX_CC_ERROR_NULL</code>—operation failed, null parameter • <code>IX_CC_ERROR_ALIGN</code>—operation failed, parameter not aligned as expected • <code>IX_CC_ERROR_ENTRY_NOT_FOUND</code>—operation failed, parameter does not exist. • <code>IX_CC_RTMV6_ERROR_NHID_IN_USE</code>—operation failed, trying to update entry in use. • <code>IX_CC_ERROR_INTERNAL</code>—operation failed, internal RTMv6 error.
--------------	--

24.2.6 ix_cc_rtmv6_get_next_hop()

Retrieves a structure containing a copy of the information originally given via `ix_cc_rtmv6_add_next_hop()`.

C Syntax

```
ix_error ix_cc_rtmv6_get_next_hop(
    ix_cc_rtmv6 rtm,
    ix_cc_rtmv6_nhid nextHopID,
    ix_cc_rtmv6_next_hop_info *pNextHopInfo);
```

Input

<code>rtm</code>	Handle to the RTMv6, as returned by <code>ix_cc_rtmv6_init()</code> .
<code>nextHopID</code>	Identifier of the Next Hop to be retrieved; network byte order.

Output/Returns

<code>pNextHopInfo</code>	Results are placed into the structure provided by the calling application.
Return Value	<p>Returns a valid <code>ix_error</code>.</p> <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • <code>IX_CC_ERROR_NULL</code>—operation failed, null parameter • <code>IX_CC_ERROR_ALIGN</code>—operation failed, parameter not aligned as expected • <code>IX_CC_ERROR_ENTRY_NOT_FOUND</code>—operation failed, Next hop ID does not exist. • <code>IX_CC_RTMV6_ERROR_NHID_IN_USE</code>—operation failed, next hop ID in use and cannot be deleted

24.2.7 ix_cc_rtmv6_set_mtu()

Updates the MTU for a given Next Hop. If the default route matches the `nextHopID`, the default route's MTU is affected as well. Because the RTMv6 does not interpret the meaning of the MTU, it does not validate the value.

C Syntax

```
ix_error ix_cc_rtmv6_set_mtu (
    ix_cc_rtmv6 rtm,
    ix_cc_rtmv6_nhid nextHopID,
    ix_uint32 mtu);
```

Input

<code>rtm</code>	Handle to the RTMv6, as returned by <code>ix_cc_rtmv6_init()</code> .
<code>nextHopID</code>	Identifier of the Next Hop to be modified; network byte order.
<code>mtu</code>	New maximum transmission unit for the Next Hop; network byte order.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. <code>IX_CC_ERROR_NULL</code>—operation failed, null parameter <code>IX_CC_ERROR_ALIGN</code>—operation failed, parameter not aligned as expected <code>IX_CC_ERROR_ENTRY_NOT_FOUND</code>—operation failed, next hop ID does not exist. <code>IX_CC_INVALID_NH_INFO</code>—operation failed, invalid next hop information.
--------------	--

24.2.8 `ix_cc_rtmv6_set_flags()`

Updates the Flags field for a given Next Hop. If the default route matches the `nextHopID`, the default route flags are affected as well. The meanings of the flags are determined by the calling application.

C Syntax

```
ix_error ix_cc_rtmv6_set_flags(
    ix_cc_rtmv6 rtm,
    ix_cc_rtmv6_nhid nextHopID,
    ix_uint32 flags);
```

Input

<code>rtm</code>	Handle to the RTMv6, as returned by <code>ix_cc_rtmv6_init()</code> .
<code>nextHopID</code>	Identifier of the Next Hop to be modified; network byte order.
<code>flags</code>	New flags for the Next Hop; network byte order.

Output/Returns

Return Value	<p>Returns a valid <code>ix_error</code>.</p> <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • <code>IX_CC_ERROR_NULL</code>—operation failed, null parameter • <code>IX_CC_ERROR_ALIGN</code>—operation failed, parameter not aligned as expected • <code>IX_CC_ERROR_ENTRY_NOT_FOUND</code>—operation failed, next hop ID does not exist. • <code>IX_CC_INVALID_NH_INFO</code>—operation failed, invalid next hop information.
--------------	--

24.2.9 `ix_cc_rtmv6_add_route()`

Adds routes, relating a range of network addresses to a Next Hop. In order to add a route, the calling application must first add the Next Hop ID and its corresponding data into the RTMv6, using `ix_cc_rtmv6_add_next_hop()`.

After adding a route, lookup operations, either from core components through the RTMv6 or from microblocks, will use the newly added route.

Adding a duplicate route to the same or different next hop is considered an error condition.

C Syntax

```
ix_error ix_cc_rtmv6_add_route (
    ix_cc_rtmv6 rtm,
    ix_uint128 netAddr,
    ix_uint8 prefixLength,
    ix_cc_rtmv6_nhid nextHopID
);
```

Input

<code>rtm</code>	Handle to the RTMv6, as returned by <code>ix_cc_rtmv6_init()</code> .
<code>netAddr</code>	Network address identifying the route; network byte order.
<code>prefixLength</code>	Prefix length for the route; network byte order.
<code>nextHopID</code>	Identifier of the Next Hop to be used for the route; network byte order.

Output/Returns

Return Value	<p>Returns a valid <code>ix_error</code>.</p> <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • <code>IX_CC_ERROR_NULL</code>—operation failed, null parameter • <code>IX_CC_ERROR_ALIGN</code>—operation failed, parameter not aligned as expected • <code>IX_CC_ERROR_ENTRY_NOT_FOUND</code>—operation failed, next hop ID does not exist. • <code>IX_CC_ERROR_DUPLICATE_ENTRY</code>—operation failed, entry already exists. • <code>IX_CC_ERROR_CONFLICTING_ENTRY</code>—operation failed, conflict with an existing entry. • <code>IX_CC_RTMV6_ERROR_INVALID_PREFIX_LENGTH</code>—operation failed, the supplied prefix length is invalid. • <code>IX_CC_ERROR_FULL</code>—operation failed, table full. • <code>IX_CC_ERROR_INTERNAL</code>—operation failed, internal error.
--------------	--

24.2.10 `ix_cc_rtmv6_delete_route()`

Deletes a route. The calling application is responsible for removing the next hop; deleting the next hop is not an implicit operation of `ix_cc_rtmv6_delete_route()`.

C Syntax

```
ix_error ix_cc_rtmv6_delete_route (
    ix_cc_rtmv6 rtm,
    ix_uint128 netAddr,
    ix_uint8 prefixLength,);
```

Input

<code>rtm</code>	Handle to the RTMv6, as returned by <code>ix_cc_rtmv6_init()</code> .
<code>netAddr</code>	Network address identifying the route; network byte order.
<code>prefixLength</code>	Prefix length for the route; network byte order.

Output/Returns

Return Value	<p>Returns a valid <code>ix_error</code>.</p> <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • <code>IX_CC_ERROR_NULL</code>—operation failed, null parameter • <code>IX_CC_ERROR_ALIGN</code>—operation failed, parameter not aligned as expected • <code>IX_CC_ERROR_ENTRY_NOT_FOUND</code>—operation failed, next hop ID does not exist. • <code>IX_CC_RTMV6_ERROR_INVALID_PREFIX_LENGTH</code>—operation failed, the supplied prefix length is invalid. • <code>IX_CC_ERROR_INTERNAL</code>—operation failed, internal error.
--------------	--

24.2.11 ix_cc_rtmv6_update_route()

Updates routes, relating a range of network addresses to a Next Hop. In order to update a route, the calling application must first add the Next Hop ID and its corresponding data into the RTMv6, using `ix_cc_rtmv6_add_next_hop()`. The calling application must also first add the routes.

C Syntax

```
ix_error ix_cc_rtmv6_update_route(
ix_cc_rtmv6 rtm,
    ix_uint128 netAddr,
    ix_uint8 prefixLength,
    ix_cc_rtmv6_nhid nextHopID
);
```

Input

<code>rtm</code>	Handle to the RTMv6, as returned by <code>ix_cc_rtmv6_init()</code> .
<code>netAddr</code>	Network address identifying the route; network byte order.
<code>prefixLength</code>	Prefix length for the route; network byte order.
<code>nextHopID</code>	Identifier of the Next Hop to be used for the route; network byte order.

Output/Returns

Return Value	<p>Returns a valid <code>ix_error</code>.</p> <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. <code>IX_CC_ERROR_NULL</code>—operation failed, null parameter <code>IX_CC_ERROR_ALIGN</code>—operation failed, parameter not aligned as expected <code>IX_CC_ERROR_ENTRY_NOT_FOUND</code>—operation failed, next hop ID does not exist. <code>IX_CC_ERROR_CONFLICTING_ENTRY</code>—operation failed, conflict with an existing entry. <code>IX_CC_RTMV6_ERROR_INVALID_PREFIX_LENGTH</code>—operation failed, the supplied prefix length is invalid. <code>IX_CC_ERROR_FULL</code>—operation failed, table full. <code>IX_CC_ERROR_INTERNAL</code>—operation failed, internal error.
--------------	--

24.2.12 ix_cc_rtmv6_lookup()

Looks up routing information for a given IPv6 address. The RTMv6 uses an LPM algorithm to match the IP address with a network (as given by calling `ix_cc_rtmv6_add_route()`) and thereby provide the information needed to forward the packet to the appropriate next hop—the corresponding Next Hop Information structure.

This function is the workhorse of the RTMv6, and is expected to be the most-used entry point; conceivably every packet arriving on the XScale core requires a lookup. Therefore, this function is designed with efficiency in mind.

If the `nextHopID` field in `pNextHopInfo` is `IX_CC_RTMV6_NHID_NO_ROUTE`, all other fields are invalid; the IP address has no route.

C Syntax

```
ix_error ix_cc_rtmv6_lookup (
    ix_cc_rtmv6 rtm,
    ix_uint128 ipAddr,
    ix_cc_rtmv6_next_hop_info *pNextHopInfo);
```

Input

<code>rtm</code>	Handle to the RTMv6, as returned by <code>ix_cc_rtmv6_init()</code> .
<code>ipAddr</code>	IP address to be looked up; network byte order.

Output/Returns

<code>*pNextHopInfo</code>	Results are placed into the structure provided by the calling application.
Return Value	<p>Returns a valid <code>ix_error</code>.</p> <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. <code>IX_CC_ERROR_NULL</code>—operation failed, null parameter <code>IX_CC_ERROR_ALIGN</code>—operation failed, parameter not aligned as expected <code>IX_CC_ERROR_INTERNAL</code>—operation failed, internal error.

24.2.13 `ix_cc_rtmv6_dump_next_hops()`

Prints all Next Hops to a block of memory for debugging purposes. The memory is provided by the calling application, along with the size of the memory block in bytes. The calling application should use `IX_CC_RTMV6_DUMP_NEXT_HOP_SIZE` to determine how many bytes to allocate.

If the function returns `IX_CC_RTMV6_ERROR_INSUFFICIENT_MEMORY`, the error should be considered non-fatal. The calling application may allocate more memory and call the function again.

C Syntax

```
ix_error ix_cc_rtmv6_dump_next_hops (
    ix_cc_rtmv6 rtm,
    char *buffer,
    ix_uint32 buffer_size
);
```

Input

<code>rtm</code>	Handle to the RTMv6, as returned by <code>ix_cc_rtmv6_init()</code> .
<code>buffer</code>	Pointer to a block of memory where the next hops are to be stored.
<code>buffer_size</code>	Size in bytes of the buffer; host byte order.

Output/Returns

<code>*buffer</code>	Human-readable ASCII-Z string
Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. <code>IX_CC_ERROR_NULL</code>—operation failed, null parameter <code>IX_CC_ERROR_ALIGN</code>—operation failed, parameter not aligned as expected <code>IX_CC_ERROR_INTERNAL</code>—operation failed, internal error.

24.2.14 ix_cc_rtmv6_dump_routes()

Prints all routes to a block of memory for debugging purposes. The memory is provided by the calling application, along with the size of the memory block in bytes. The calling application should use `IX_CC_RTMV6_DUMP_ROUTE_SIZE` to determine how many bytes to allocate.

If the function returns `IX_CC_RTMV6_ERROR_INSUFFICIENT_MEMORY`, the error should be considered non-fatal. The calling application may allocate more memory and call the function again.

C Syntax

```
ix_error ix_cc_rtmv6_dump_routes(
    ix_cc_rtmv6 rtm,
    char *buffer,
    ix_uint32 buffer_size
);
```

Input

<code>rtm</code>	Handle to the RTMv6, as returned by <code>ix_cc_rtmv6_init()</code> .
<code>buffer</code>	Pointer to a block of memory where the next hops are to be stored.
<code>buffer_size</code>	Size in bytes of the buffer; host byte order.

Output/Returns

<code>*buffer</code>	Human-readable ASCII-Z string
Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. <code>IX_CC_ERROR_NULL</code>—operation failed, null parameter <code>IX_CC_ERROR_ALIGN</code>—operation failed, parameter not aligned as expected <code>IX_CC_ERROR_INTERNAL</code>—operation failed, internal error

24.2.15 ix_cc_rtmv6_purge()

Removes all routes and next hops. This function would only be done for debugging purposes.

C Syntax

```
ix_error ix_cc_rtmv6_purge (
    ix_cc_rtmv6 rtm);
```

Input

<code>rtm</code>	Handle to the RTMv6, as returned by <code>ix_cc_rtmv6_init()</code> .
------------------	---

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none">• <code>IX_SUCCESS</code>—the operation succeeded.• <code>IX_CC_ERROR_NULL</code>—operation failed, null parameter• <code>IX_CC_ERROR_ALIGN</code>—operation failed, parameter not aligned as expected• <code>IX_CC_ERROR_INTERNAL</code>—operation failed, internal error.
--------------	--

24.2.16 `ix_cc_rtmv6_purge_routes()`

Removes all routes. Most likely, this would only be done for debugging purposes.

C Syntax

```
ix_error ix_cc_rtmv6_purge_routes (  
    ix_cc_rtmv6 rtm);
```

Input

<code>rtm</code>	Handle to the RTMv6, as returned by <code>ix_cc_rtmv6_init()</code> .
------------------	---

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none">• <code>IX_SUCCESS</code>—the operation succeeded.• <code>IX_CC_ERROR_NULL</code>—operation failed, null parameter• <code>IX_CC_ERROR_ALIGN</code>—operation failed, parameter not aligned as expected• <code>IX_CC_ERROR_INTERNAL</code>—operation failed, internal error.
--------------	--

24.2.17 ix_cc_rtmv6_get_symbols()

Retrieves symbols to patch to the microengines.

Syntax

```
ix_error ix_cc_rtmv6_get_symbols (
    ix_cc_rtmv6 rtm,
    ix_cc_rtmv6_symbols *pSymbols);
```

Input

rtm	Handle to the RTMv6, as returned by <code>ix_cc_rtmv6_init()</code> .
-----	---

Output/Returns

*pSymbols	Structure containing values that the calling application may patch to the microengines.
Return Value	<p>Returns a valid <code>ix_error</code>.</p> <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • <code>IX_CC_ERROR_NULL</code>—operation failed, null parameter • <code>IX_CC_ERROR_ALIGN</code>—operation failed, parameter not aligned as expected • <code>IX_CC_ERROR_INTERNAL</code>—operation failed, internal error.

24.2.18 ix_cc_rtmv6_get_statistics()

Retrieves statistics of the RTMv6.

C Syntax

```
ix_error ix_cc_rtmv6_get_statistics (  
    ix_cc_rtmv6 rtm,  
    ix_cc_rtmv6_statistics *pStatistics);
```

Input

rtm	Handle to the RTMv6, as returned by ix_cc_rtmv6_init() .
-----	--

Output/Returns

*pStatistics	Structure containing the statistics.
Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none">• <code>IX_SUCCESS</code>—the operation succeeded.• <code>IX_CC_ERROR_NULL</code>—operation failed, null parameter• <code>IX_CC_ERROR_ALIGN</code>—operation failed, parameter not aligned as expected• <code>IX_CC_ERROR_INTERNAL</code>—operation failed, internal error.

The L2 Table Manager provides an API to manage, add and delete entries in the L2 Table, which corresponds to the Route Table and Next Hop Database on the ingress side. See [Chapter 23, “Route Table Manager”](#) for details on the Next Hop Database.

The L2 Table Manager (L2TM) library is not part of Core Component Infrastructure and does not need to conform infrastructure specific APIs. For complete details see [Chapter 62, “L2 Table Manager”](#) of the *Intel® Internet Exchange Architecture Software Building Blocks Developer’s Manual*.

The L2 Table Manager exports an interface for calling applications to control the L2 table. The interface consists of a number of enumerations, data structures, functions, etc. This section describes the exported API.

25.1 Data Structures, Types and Definitions

Table 25-1 lists the data structures, types, definitions and enumerations used by L2 Table Manager.

Table 25-1. Data Structures, Types, Definitions and Enumerations in L2TM

Data Structures, Types and Definitions	Description
<code>ix_s_cc_l2tm_atm_header</code>	Describes an ATM entry in the L2 Table.
<code>ix_s_cc_l2tm_ether_header</code>	Describes an Ethernet entry in the L2 Table.
<code>ix_u_cc_l2tm_ipaddr</code>	Union containing an IP address.
<code>ix_s_cc_l2tm_entry</code>	Describes an entry in the L2 Table.
<code>ix_s_cc_l2tm_stats</code>	Statistics about the L2 Table Manager and its associated table.
<code>ix_s_cc_l2tm_symbols</code>	Contains values useful to microblocks.
<code>ix_s_cc_l2tm_config</code>	Configuration structure for creating a new L2 Table.
<code>ix_cc_l2tm</code>	Opaque handle to the L2Table Manager.
<code>ix_cc_l2tm_ipaddr_type Enumeration</code>	Indicates the type of the IP address to be stored.
<code>ix_cc_l2tm_l2addr_type Enumeration</code>	Indicates the type of the header to be stored.
<code>ix_e_cc_l2tm_error Enumeration</code>	Error codes specific to the L2 Table Manager.
<code>ix_cc_l2tm_memory_type Enumeration</code>	Types of memory supported for creating a new L2 Table.
<code>ix_cc_l2tm_state Enumeration</code>	Enumerations of the states each of the entries could occupy.

25.1.1 ix_s_cc_l2tm_atm_header

Describes an ATM entry in the L2 Table. This structure is used when adding an L2 entry or looking up an L2 Entry by Next Hop IP address or L2 Index.

C Syntax

```
typedef struct ix_s_cc_l2tm_atm_header
{
    ix_uint32 vpiVci;
    ix_uint32 vcq;
    ix_uint64 llcSnapHeader;
} ix_cc_l2tm_atm_header;
```

Data Elements

vpiVci	Virtual Path Identifier/Virtual Channel Identifier. The low 28-bits are kept for the microblocks. The high 4-bits are reserved and must be zero. The L2TM does not validate the VPI/VCI fields, other than to limit it to 28-bits.
vcq	Virtual Channel Queue. The low 16-bits are stored for the microblocks. The high 16-bits are reserved and must be zero. The L2TM does not validate the vcq, other than to limit it to 16-bits.
llcSnapHeader	LLC / SNAP Header. The L2TM does not validate the LLC and SNAP headers.

25.1.2 ix_s_cc_l2tm_config

Configuration structure for creating a new L2 Table.

C Syntax

```
typedef struct ix_s_cc_l2tm_config
{
    ix_uint32 numEntries;
    ix_cc_l2tm_memory_type memoryType;
    ix_uint32 memoryChannel;
} ix_cc_l2tm_config;
```

Data Elements

numEntries	Requested number of L2 entries for the table.
memoryType	Type of memory to use for L2 Table.
memoryChannel	Channel of memory to use for the L2 Table.

Note: The range of channels allowed for a given memory type is determined by the Resource Manager.

25.1.3 ix_s_cc_l2tm_entry

Describes an entry in the L2 Table.

This structure is used when adding an L2 Entry, retrieving an L2 Entry, or looking up an L2 Entry by Next Hop IP address or L2 Index.

Though this data is identical in meaning to the L2 Table Entry used by the Interface TX microblock, the structure used to store the data is less restrictive. The L2TM internally packs the L2 Entry data into the format the microblocks expect.

C Syntax

```
typedef struct ix_s_cc_l2tm_entry
{
    ix_cc_l2tm_l2addr_type l2AddrType;
    ix_cc_l2tm_ipaddr_type ipAddrType;
    union
    {
        ix_cc_l2tm_ether_header ether;
        ix_cc_l2tm_atm_header atm;
    } header;
    ix_cc_l2tm_ipaddr l3Address;
};
```

Data Elements

l2AddrType	The type of header to be added, or the type of header stored in the specified location.
ipAddrType	The type of address to be added, or the type of address stored in the specified location.
ether	Header for an Ethernet entry.
atm	Header for an ATM entry.
l3Address	The L3 address of the entry being added or looked up

25.1.4 `ix_s_cc_l2tm_ether_header`

Describes an Ethernet entry in the L2 Table. This structure is used when adding an L2 entry or looking up an L2 Entry by Next Hop IP address or L2 Index.

C Syntax

```
typedef struct ix_s_cc_l2tm_ether_header ix_cc_l2tm_ether_header;
struct ix_s_cc_l2tm_ether_header
{
    ix_uint8 destMAC[6];
    ix_uint8 sourceMAC[6];
    ix_uint16 protocol;
};
```

Data Elements

<code>destMAC</code>	Destination MAC address. The L2TM does not validate the destination MAC address.
<code>sourceMAC</code>	Source MAC address. The L2TM does not validate the source MAC address.
<code>protocol</code>	Protocol type. The L2TM does not validate the protocol.

25.1.5 `ix_s_cc_l2tm_stats`

Statistics about the L2 Table Manager and its associated table. The statistics represent a snapshot of the state of the L2TM and table, as well as some static information such as the number of entries the table could contain. See also `ix_cc_l2tm_get_statistics()`.

C Syntax

```
typedef struct ix_s_cc_l2tm_stats
{
    ix_uint32 numberOfValidEntries;
    ix_uint32 numberOfIncompleteEntries;
    ix_uint32 tableCapacity;
    ix_uint32 heapMemoryUsed;
    ix_uint32 sharedMemoryUsed;
};
```


Data Elements

<code>numberOfValidEntries</code>	Number of “Valid” L2 entries currently contained within the L2 Table.
<code>numberOfIncompleteEntries</code>	Number of “Incomplete” L2 entries currently contained within the L2 Table.
<code>tableCapacity</code>	Maximum number of entries the L2 Table could contain.
<code>heapMemoryUsed</code>	Amount of heap memory used for the L2 Table Manager.
<code>sharedMemoryUsed</code>	Amount of shared memory used for the L2 Table.

25.1.6 `ix_s_cc_l2tm_symbols`

Contains values useful to microblocks. This structure is returned by `ix_cc_l2tm_get_symbols()`.

C Syntax

```
typedef struct ix_s_cc_l2tm_symbols
{
    ix_uint32 l2TableBase;
};
```

Data Element

<code>l2TableBase</code>	Base address of the L2 Table.
--------------------------	-------------------------------

25.1.7 `ix_u_cc_l2tm_ipaddr`

Union containing an IP address. When using this union, a corresponding `ix_cc_l2tm_ipaddr_type` indicates which address is in use. See also [ix_cc_l2tm_l2addr_type Enumeration](#).

C Syntax

```
typedef struct ix_s_cc_l2tm_ipaddr
{
    union ix_u_cc_l2tm_ipaddr
    {
        ix_uint32 ipv4Address;
        ix_uint128 ipv6Address;
    };
} ix_cc_l2tm_ipaddr;
```

Data Element

ipv4Address	32-bit address for IPv4.
ipv6Address	128-bit address for IPv6.

25.1.8 ix_cc_l2tm

Opaque handle to the L2Table Manager. The handle is generated for each calling application by calling `ix_cc_l2tm_init()` or `ix_cc_l2tm_create()`. Each generated handle should be released by calling `ix_cc_l2tm_fini()` or `ix_cc_l2tm_destroy()`.

C Syntax

```
typedef struct _ix_s_cc_l2tm* ix_cc_l2tm
```

25.1.9 ix_cc_l2tm_ipaddr_type Enumeration

Indicates the type of the IP address to be stored.

C Syntax

```
typedef enum ix_e_cc_l2tm_ipaddr_type
{
    IX_CC_L2TM_IPADDR_IPV4, /* The IP Address is for IPv4, and
                           therefore 32 bits */
    IX_CC_L2TM_IPADDR_IPV6, /* The IP Address is for IPv6, and
                           therefore 128 bits */
    IX_CC_L2TM_IPADDR_LAST, /* Last entry in the IP address enumeration */
} ix_cc_l2tm_ipaddr_type;
```

25.1.10 ix_cc_l2tm_l2addr_type Enumeration

Indicates the type of the header to be stored.

C Syntax

```
typedef enum ix_e_cc_l2tm_l2addr_type
{
    IX_CC_L2TM_L2ADDR_ETHER, /* The L2 Header is for Ethernet */
    IX_CC_L2TM_L2ADDR_ATM, /* The L2 Header is for ATM */
    IX_CC_L2TM_L2ADDR_LAST, /* Last entry in the header enumeration */
} ix_cc_l2tm_l2addr_type;
```

25.1.11 ix_e_cc_l2tm_error Enumeration

Error codes specific to the L2 Table Manager.

C Syntax

```
enum ix_e_cc_l2tm_error
{
    IX_CC_L2TM_ERROR_CREATE_L2TABLE, /* Failed to create the L2 Table */
    IX_CC_L2TM_ERROR_DELETE_TABLE, /* Failed to delete the L2 Table */
    IX_CC_L2TM_ERROR_NOT_FOUND, /* The L2TM with the specified name
                                could not be found */
    IX_CC_L2TM_ERROR_REGISTRY, /* A problem with the registry was
                                encountered */
    IX_CC_L2TM_ERROR_INVALID_ENTRY, /* The entry structure contains
                                invalid information */
    IX_CC_L2TM_ERROR_LAST, /* Last entry in the L2TM error enumeration */
};
```

25.1.12 ix_cc_l2tm_memory_type Enumeration

Types of memory supported for creating a new L2 Table.

C Syntax

```
enum ix_e_cc_l2tm_memory_type
{
    IX_CC_L2TM_MEMORY_TYPE_SRAM /* Use SRAM for the L2 Table */
    IX_CC_L2TM_MEMORY_TYPE_DRAM /* Use DRAM for the L2 Table */
    IX_CC_L2TM_MEMORY_TYPE_LAST /* Last entry in the
                                ix_cc_l2tm_l2addr_type Enumeration*/
} ix_cc_l2tm_memory_type;
```

25.1.13 ix_cc_l2tm_state Enumeration

Enumerations of the states each of the entries could occupy.

C Syntax

```
typedef enum ix_e_cc_l2tm_state
{
    IX_CC_L2TM_STATE_IDLE /* The entry is unused.
                           This is indicated by the 'in use' flag being zero.
    IX_CC_L2TM_STATE_INCOMPLETE /* The entry is in use, but not usable
                                by Microblock. It has L3 information, but no L2
                                information at this time. The 'in use' flag is set,
                                and the 'valid' flag in shared memory is zero. */
    IX_CC_L2TM_STATE_VALID /* The entry is usable by the microblocks.
                           It has valid L3 information and L2 information.
                           The 'in use' and 'valid' flags are set. */
} ix_cc_l2tm_state;
```

25.2 Library API

Table 25-2 lists the L2 Table Manager library API.

Table 25-2. L2 Table Manager Library API

Functions	Description
<code>ix_cc_l2tm_create()</code>	Returns a handle to a named L2 Table Manager.
<code>ix_cc_l2tm_destroy()</code>	Destroys a named L2 Table Manager.
<code>ix_cc_l2tm_init()</code>	Creates a new L2TM for the calling application.
<code>ix_cc_l2tm_fini()</code>	Destroys an L2 Table Manager.
<code>ix_cc_l2tm_add_entry()</code>	Adds an L2 Entry to the L2TM's table database.
<code>ix_cc_l2tm_update_entry()</code>	Updates an L2 Entry to the L2TM's table database.
<code>ix_cc_l2tm_delete_entry()</code>	Removes an L2 Entry from the L2TM.
<code>ix_cc_l2tm_get_entry()</code>	Gets an L2 Entry from the L2TM's table database.
<code>ix_cc_l2tm_add_l3_info()</code>	Adds L3 information into the L2 Table.
<code>ix_cc_l2tm_update_l3_info()</code>	Updates L3 information in the L2 Table.
<code>ix_cc_l2tm_assign_l2_info()</code>	Updates an L2 Entry with new information.
<code>ix_cc_l2tm_clear_l2_info()</code>	Clears the L2 Information from an entry from the L2TM.
<code>ix_cc_l2tm_get_l2_index()</code>	Gets the index of the entry containing a given IP address.
<code>ix_cc_l2tm_get_symbols()</code>	Gets symbols useful to the microblocks.
<code>ix_cc_l2tm_get_statistics()</code>	Gets statistics about the L2 Table.
<code>ix_cc_l2tm_purge()</code>	Remove all entries from the L2 Table.

25.2.1 `ix_cc_l2tm_create()`

Returns a handle to a named L2 Table Manager. If the named L2TM does not yet exist, its parameters are retrieved from the registry and the L2TM is created. If the L2TM already exists, it is provided to the caller.

A named L2 Table Manager should be used when a single table must be shared among multiple calling applications. If a table is intended to be used for only a single calling application, the calling application could choose to use `ix_cc_l2tm_create()` or `ix_cc_l2tm_init()`.

The name provided must match a configuration in the registry. All calling applications sharing a table must invoke `ix_cc_l2tm_create()` with the same name.

A handle obtained using `ix_cc_l2tm_create()` should be freed using `ix_cc_l2tm_destroy()`.

C Syntax

```
ix_error ix_cc_l2tm_create (
    const char * arg_pName,
    ix_cc_l2tm * arg_phL2tm);
```

Input

`arg_pName` Name of the L2 Table Manager to be created.

Output/Returns

`arg_phL2tm` A handle to the requested L2 Table Manager.

Return Value Returns a valid `ix_error`.

- `IX_SUCCESS`—the operation succeeded.
- `IX_CC_ERROR_RANGE`—operation failed, one of the provided values was out of range.
- `IX_CC_ERROR_NULL`—operation failed, a required pointer was NULL.
- `IX_CC_ERROR_ALIGN`—operation failed, a pointer was not properly aligned.
- `IX_CC_ERROR_OOM_SYSTEM`—operation failed, memory could not be allocated to create the L2TM handle.
- `IX_CC_L2TM_ERROR_CREATE_L2TABLE`—operation failed, the L2TM failed to create the L2 Table.
- `IX_CC_L2TM_ERROR_NOT_FOUND`—operation failed, the L2TM with the specified name could not be found.
- `IX_CC_L2TM_ERROR_REGISTRY`—operation failed, a problem with the registry was encountered.

25.2.2 `ix_cc_l2tm_destroy()`

Destroys a named L2 Table Manager. The named L2 Table Manager is freed when the last calling application calls this function. This function should only be used with handles obtained from `ix_cc_l2tm_create()`.

Depending on the runtime environment, `ix_cc_l2tm_destroy()` sometimes invokes `ix_cc_l2tm_fini()`

Note: See the warnings about calling `ix_cc_l2tm_fini()`.

C Syntax

```
ix_error ix_cc_l2tm_destroy (ix_cc_l2tm arg_hL2tm);
```

Input

`arg_hL2tm` Handle to the named L2 Table Manager to be terminated.

Output/Returns

Return Value	<p>Returns a valid <code>ix_error</code>.</p> <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • <code>IX_CC_ERROR_NULL</code>—operation failed, a required pointer was NULL. • <code>IX_CC_ERROR_ALIGN</code>—operation failed, a pointer was not properly aligned. • <code>IX_CC_L2TM_ERROR_DELETE_TABLE</code>—operation failed, the L2TM was unable to free the resources for the L2 Table. • <code>IX_CC_L2TM_ERROR_REGISTRY</code>—operation failed, a problem with the registry was encountered.
--------------	---

25.2.3 `ix_cc_l2tm_init()`

Creates a new L2TM for the calling application. Note that this function always initializes a new unnamed L2TM; if the L2TM is intended to be shared with other calling applications, use `ix_cc_l2tm_create()` instead.

A handle obtained using `ix_cc_l2tm_init()` should be freed using `ix_cc_l2tm_fini()`.

C Syntax

```
ix_error ix_cc_l2tm_init (
    const ix_cc_l2tm_config * arg_pConfig,
    ix_cc_l2tm * arg_phL2tm);
```

Input

<code>arg_pConfig</code>	Configuration to use when creating the new L2 Table.
--------------------------	--

Output/Returns

<code>arg_phL2tm</code>	A handle to the L2 Table Manager
Return Value	<p>Returns a valid <code>ix_error</code>.</p> <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • <code>IX_CC_ERROR_RANGE</code>—operation failed, one of the provided values was out of range. • <code>IX_CC_ERROR_NULL</code>—operation failed, a required pointer was NULL. • <code>IX_CC_ERROR_ALIGN</code>—operation failed, a pointer was not properly aligned. • <code>IX_CC_ERROR_OOM_SYSTEM</code>—operation failed, memory could not be allocated to create the L2TM handle. • <code>IX_CC_L2TM_ERROR_CREATE_L2TABLE</code>—operation failed, the L2TM failed to create the L2 Table.

25.2.4 ix_cc_l2tm_fini()

Destroys an L2 Table Manager. The unnamed L2 Table Manager is freed immediately; because `init` and `fini` do not account for multiple calling applications, calling applications should use `ix_cc_l2tm_create()` and `ix_cc_l2tm_destroy()` when a single table is to be shared.

Because microblocks use the tables managed by the L2TM, the tables should be valid whenever the microblocks are executing. Therefore, any calling application responsible for microblocks should disable its microengines before destroying the L2TM.

C Syntax

```
ix_error ix_cc_l2tm_fini (ix_cc_l2tm arg_hL2tm);
```

Input

<code>arg_hL2tm</code>	Handle to the L2TM, as returned by <code>ix_cc_l2tm_init()</code> .
------------------------	---

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none">• <code>IX_SUCCESS</code>—the operation succeeded.• <code>IX_CC_ERROR_NULL</code>—operation failed, a required pointer was <code>NULL</code>.• <code>IX_CC_ERROR_ALIGN</code>—operation failed, a pointer was not properly aligned.• <code>IX_CC_L2TM_ERROR_DELETE_TABLE</code>—operation failed, the L2TM was unable to free the resources for the L2 Table.
--------------	--

25.2.5 ix_cc_l2tm_add_entry()

Adds an L2 Entry to the L2TM database. Typically, calling applications would do this any time a new Next Hop is detected on a network. Next Hops are uniquely identified by their layer 2 index. The L2TM does not perform any checks for duplicate entries elsewhere in the table, but does check for an existing entry at the location in the table indicated by the L2Index.

`ix_cc_l2tm_create()` should be used when assigning both valid L2 information and an IP address. If only the IP address is known, use `ix_cc_l2tm_add_l3_info()` instead.

C Syntax

```
ix_error ix_cc_l2tm_add_entry (
    ix_cc_l2tm arg_hL2tm,
    ix_uint16 arg_L2Index,
    ix_cc_l2tm_entry * arg_pL2Entry);
```

Input

<code>arg_hL2tm</code>	Handle to the L2TM, as returned by <code>ix_cc_l2tm_init()</code> .
<code>arg_L2Index</code>	Index into the L2 Table to be added.
<code>arg_pL2Entry</code>	Pointer to the L2 Entry structure to be added.

Output/Returns

Return Value	<p>Returns a valid <code>ix_error</code>.</p> <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • <code>IX_CC_ERROR_NULL</code>—operation failed, a required pointer was NULL. • <code>IX_CC_ERROR_ALIGN</code>—operation failed, a pointer was not properly aligned. • <code>IX_CC_ERROR_RANGE</code>—operation failed, the L2 Index was out of range (beyond the capacity of the L2 Table). • <code>IX_CC_L2TM_ERROR_INVALID_ENTRY</code>—operation failed, the entry structure contains invalid information. • <code>IX_CC_ERROR_DUPLICATE_ENTRY</code>—operation failed, an entry with the same contents already exists at the specified location. • <code>IX_CC_ERROR_CONFLICTING_ENTRY</code>—operation failed, an entry with different contents already exists at the specified location.
--------------	---

25.2.6 ix_cc_l2tm_update_entry()

Updates an L2 Entry to the L2TM's table database. Typically, calling applications would do this when a next hop's information has changed. The L2TM does not perform any checks for duplicate entries elsewhere in the table, but does verify the entry has valid data.

C Syntax

```
ix_error ix_cc_l2tm_update_entry (
    ix_cc_l2tm arg_hL2tm,
    ix_uint16 arg_L2Index,
    ix_cc_l2tm_entry * arg_pL2Entry);
```

Input

arg_hL2tm	Handle to the L2TM, as returned by <code>ix_cc_l2tm_init()</code> .
arg_L2Index	Index into the L2 Table to be updated.
arg_pL2Entry	Pointer to the L2 Entry structure to be added.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> IX_SUCCESS—the operation succeeded. IX_CC_ERROR_NULL—operation failed, a required pointer was NULL. IX_CC_ERROR_ALIGN—operation failed, a pointer was not properly aligned. IX_CC_ERROR_RANGE—operation failed, the L2 Index was out of range (beyond the capacity of the L2 Table). IX_CC_ERROR_ENTRY_NOT_FOUND—operation failed, the L2 Index does not contain valid data. IX_CC_L2TM_ERROR_INVALID_ENTRY—operation failed, the entry structure contains invalid information.
--------------	--

25.2.7 ix_cc_l2tm_delete_entry()

Removes an L2 Entry from the L2TM. A calling application would typically delete an L2 Entry when a next hop router is removed from the network.

C Syntax

```
ix_error ix_cc_l2tm_delete_entry (
    ix_cc_l2tm arg_hL2tm,
    ix_uint16 arg_L2Index);
```

Input

<code>arg_hL2tm</code>	Handle to the L2TM, as returned by <code>ix_cc_l2tm_init()</code> .
<code>arg_L2Index</code>	Index into the L2 Table to be deleted.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. <code>IX_CC_ERROR_NULL</code>—operation failed, a required pointer was NULL. <code>IX_CC_ERROR_ALIGN</code>—operation failed, a pointer was not properly aligned. <code>IX_CC_ERROR_RANGE</code>—operation failed, the L2 Index was out of range (beyond the capacity of the L2 Table). <code>IX_CC_ERROR_ENTRY_NOT_FOUND</code>—operation failed, the entry at the specified location does not exist.
--------------	---

25.2.8 `ix_cc_l2tm_get_entry()`

Gets an L2 Entry from the L2TM's table database.

C Syntax

```
ix_error ix_cc_l2tm_get_entry (
    ix_cc_l2tm arg_hL2tm,
    ix_uint16 arg_L2Index,
    ix_cc_l2tm_state * arg_pState,
    ix_cc_l2tm_entry * arg_pL2Entry);
```

Input

<code>arg_hL2tm</code>	Handle to the L2TM, as returned by <code>ix_cc_l2tm_init()</code> .
<code>arg_L2Index</code>	Index into the L2 Table to be retrieved.

Output/Returns

<code>arg_pState</code>	Current state of the entry. This parameter is optional; if the pointer is NULL, the state is not returned.
<code>arg_pL2Entry</code>	Pointer to the L2 Entry structure to be retrieved. This parameter is optional; if the pointer is NULL, the contents of the structure are not returned.

Output/Returns (Continued)

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. <code>IX_CC_ERROR_NULL</code>—operation failed, a required pointer was NULL. <code>IX_CC_ERROR_ALIGN</code>—operation failed, a pointer was not properly aligned. <code>IX_CC_ERROR_RANGE</code>—operation failed, the L2 Index was out of range (beyond the capacity of the L2 Table).
--------------	---

25.2.9 `ix_cc_l2tm_add_l3_info()`

Adds L3 information into the L2 Table. The entry is marked as in use, but not valid, as the L2 information is not yet valid. It is expected that the calling application will call `ix_cc_l2tm_assign_l2_info()` to complete the entry.

C Syntax

```
ix_error ix_cc_l2tm_add_l3_info (
    ix_cc_l2tm arg_hL2tm,
    ix_uint16 arg_L2Index,
    const ix_cc_l2tm_ipaddr * arg_pIpAddress,
    ix_cc_l2tm_ipaddr_type arg_IpAddressType);
```

Input

<code>arg_hL2tm</code>	Handle to the L2TM, as returned by <code>ix_cc_l2tm_init()</code> .
<code>arg_L2Index</code>	Index into the L2 Table to be reserved.
<code>arg_pIpAddress</code>	L3 Information to add to the entry.
<code>arg_IpAddressType</code>	Type of the L3 address to be added.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. <code>IX_CC_ERROR_NULL</code>—operation failed, a required pointer was NULL. <code>IX_CC_ERROR_ALIGN</code>—operation failed, a pointer was not properly aligned. <code>IX_CC_ERROR_RANGE</code>—operation failed, the L2 Index was out of range (beyond the capacity of the L2 Table). <code>IX_CC_ERROR_ENTRY_NOT_FOUND</code>—operation failed, the entry at the specified location does not exist.
--------------	---

25.2.10 `ix_cc_l2tm_update_l3_info()`

Updates L3 information in the L2 Table. It is expected that the calling application will call `ix_cc_l2tm_assign_l2_info()` to complete the entry.

A caller might use this entry point to modify the IP address of an existing entry in the table. Note that doing so causes the L2 information to be invalidated; the caller must re-assign new L2 information. (If the caller wants to change the L2 information simultaneously with updating the IP address, use `ix_cc_l2tm_update_entry()` instead).

C Syntax

```
ix_error ix_cc_l2tm_update_l3_info (
    ix_cc_l2tm arg_hL2tm,
    ix_uint16 arg_L2Index,
    const ix_cc_l2tm_ipaddr * arg_pIpAddress,
    ix_cc_l2tm_ipaddr_type arg_IPAddressType);
```

Input

<code>arg_hL2tm</code>	Handle to the L2TM, as returned by <code>ix_cc_l2tm_init()</code> .
<code>arg_L2Index</code>	Index into the L2 Table to be reserved.
<code>arg_pIpAddress</code>	L3 Information to add to the entry.
<code>arg_IPAddressType</code>	Type of the L3 address to be added.

Output/Returns

Return Value	<p>Returns a valid <code>ix_error</code>.</p> <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • <code>IX_CC_ERROR_NULL</code>—operation failed, a required pointer was NULL. • <code>IX_CC_ERROR_ALIGN</code>—operation failed, a pointer was not properly aligned. • <code>IX_CC_ERROR_RANGE</code>—operation failed, the L2 Index was out of range (beyond the capacity of the L2 Table). • <code>IX_CC_ERROR_ENTRY_NOT_FOUND</code>—operation failed, the entry at the specified location does not exist.
--------------	---

25.2.11 `ix_cc_l2tm_assign_l2_info()`

Updates an L2 Entry with new information. Typically this would be used to complete an entry after ARP has determined the L2 information.

C Syntax

```
ix_error ix_cc_l2tm_assign_l2_info (
    ix_cc_l2tm arg_hL2tm,
    ix_cc_l2tm_entry * arg_pL2Entry,
    ix_uint16 * arg_pL2Index);
```

Input

<code>arg_hL2tm</code>	Handle to the L2TM, as returned by <code>ix_cc_l2tm_init()</code> .
<code>arg_pL2Entry</code>	L2 Table Entry structure—uses the IP address from the data structure as an index to populate the L2 Table.

Output/Returns

<code>arg_pL2Index</code>	Index in the L2 Table where the structure was added. This argument is optional; if the pointer is NULL, the L2 Index is not returned to the caller.
Return Value	<p>Returns a valid <code>ix_error</code>.</p> <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • <code>IX_CC_ERROR_NULL</code>—operation failed, a required pointer was NULL. • <code>IX_CC_ERROR_ALIGN</code>—operation failed, a pointer was not properly aligned. • <code>IX_CC_ERROR_RANGE</code>—operation failed, the L2 Index was out of range (beyond the capacity of the L2 Table). • <code>IX_CC_L2TM_ERROR_INVALID_ENTRY</code>—operation failed, the entry structure contains invalid information.

25.2.12 ix_cc_l2tm_clear_l2_info()

Clears the L2 Information from an entry from the L2TM. A calling application would typically clear the L2 Information when an ARP timeout has occurred and no further information is available for the next hop.

C Syntax

```
ix_error ix_cc_l2tm_clear_l2_info (
    ix_cc_l2tm arg_hL2tm,
    ix_uint16 arg_L2Index);
```

Input

arg_hL2tm	Handle to the L2TM, as returned by <code>ix_cc_l2tm_init()</code> .
arg_L2Index	Index into the L2 Table to be updated.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none">• <code>IX_SUCCESS</code>—the operation succeeded.• <code>IX_CC_ERROR_NULL</code>—operation failed, a required pointer was NULL.• <code>IX_CC_ERROR_ALIGN</code>—operation failed, a pointer was not properly aligned.• <code>IX_CC_ERROR_RANGE</code>—operation failed, the L2 Index was out of range (beyond the capacity of the L2 Table).• <code>IX_CC_ERROR_ENTRY_NOT_FOUND</code>—operation failed, the entry at the specified location does not exist.
--------------	---

25.2.13 ix_cc_l2tm_get_l2_index()

Gets the index of the entry containing a given IP address. This function is a general-purpose means of converting an IP address to an L2Index, given the IP address exists in the L2 Table.

C Syntax

```
ix_error ix_cc_l2tm_get_l2_index (
    ix_cc_l2tm arg_hL2tm,
    ix_cc_l2tm_ipaddr_type arg_IPAddressType,
    const ix_cc_l2tm_ipaddr * arg_pIPAddress,
    ix_uint16 * arg_pL2Index);
```

Input

<code>arg_hL2tm</code>	Handle to the L2TM, as returned by <code>ix_cc_l2tm_init()</code> .
<code>arg_IpAddressType</code>	Indicates the type of the L3 Address.
<code>arg_pIpAddress</code>	Layer-3 address to be looked up.

Output/Returns

<code>arg_pL2Index</code>	Index in the L2 Table where the structure was added. If this argument is NULL, the L2 Index is not returned to the caller.
Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • <code>IX_CC_ERROR_NULL</code>—operation failed, a required pointer was NULL. • <code>IX_CC_ERROR_ALIGN</code>—operation failed, a pointer was not properly aligned. • <code>IX_CC_ERROR_RANGE</code>—operation failed, Invalid L3 address type. • <code>IX_CC_ERROR_ENTRY_NOT_FOUND</code>—operation failed, the L3 address was not found in the L2 Table.

25.2.14 `ix_cc_l2tm_get_symbols()`

Gets symbols useful to the microblocks. Typically the symbols are patched to the microblocks.

C Syntax

```
ix_error ix_cc_l2tm_get_symbols (
ix_cc_l2tm arg_hL2tm,
ix_cc_l2tm_symbols * arg_pSymbols);
```

Input

<code>arg_hL2tm</code>	A handle to the L2TM, as returned by <code>ix_cc_l2tm_init()</code> .
------------------------	---

Output/Returns

<code>arg_pSymbols</code>	A structure with symbols that need to be patched
Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • <code>IX_CC_ERROR_NULL</code>—operation failed, a required pointer was NULL. • <code>IX_CC_ERROR_ALIGN</code>—operation failed, a pointer was not properly aligned.

25.2.15 ix_cc_l2tm_get_statistics()

Gets statistics about the L2 Table. The statistics describe the current state of the L2TM and its associated table.

C Syntax

```
ix_error ix_cc_l2tm_get_statistics (
    ix_cc_l2tm arg_hL2tm,
    ix_cc_l2tm_stats * arg_pStatistics);
```

Input

arg_hL2tm Handle to the L2TM, as returned by `ix_cc_l2tm_init()`.

(OUT)

Output/Returns

arg_pStatistics Structure containing statistics about the L2 Table.

Return Value Returns a valid `ix_error`.

- `IX_SUCCESS`—the operation succeeded.
- `IX_CC_ERROR_NULL`—operation failed, a required pointer was NULL.
- `IX_CC_ERROR_ALIGN`—operation failed, a pointer was not properly aligned.

25.2.16 ix_cc_l2tm_purge()

Remove all entries from the L2 Table. This is primarily intended to be used for development and debugging.

C Syntax

```
ix_error ix_cc_l2tm_purge (ix_cc_l2tm arg_hL2tm);
```

Input

arg_hL2tm Handle to the L2TM, as returned by `ix_cc_l2tm_init()`.

Output/Returns

Return Value Returns a valid `ix_error`.

- `IX_SUCCESS`—returned if the operation succeeds.
- `IX_CC_ERROR_NULL`—operation failed, a required pointer was null.
- `IX_CC_ERROR_ALIGN`—operation failed, a pointer was not properly aligned.

Message Helper and Support Library 26

The Core Component Infrastructure provides a way to pass messages from one core component to another core component by using message handlers. A message can be anything - it is an array of bytes. An individual core components need to know how to interpret each message. The APIs exposed by the core components can be invoked through the messaging mechanism.

The Message Helper and Support Library is a translation layer to the actual Core Component Infrastructure function calls which simplifies calling the API from the clients of the core components.

For complete details, see [Chapter 63, “Message Helper and Support Library”](#) of the *Intel® Internet Exchange Architecture (IXA) Software Building Blocks Developer’s Manual*.

26.1 Message Support Library API

Table 26-1 shows the message support library API.

Table 26-1. Message Support Library API

API	Description
ix_cc_msup_init()	Initializes the message support library.
ix_cc_msup_fini()	Shuts down the message support library.
ix_cc_msup_send_msg()	Sends a fire-and-forget message.
ix_cc_msup_send_async_msg()	Sends an asynchronous message and invokes the callback function when the return message is received.
ix_cc_msup_send_bcast_msg()	Sends a fire-and-forget message.
ix_cc_msup_send_sync_msg()	Sends a synchronous message.
IX_MSUP_EXTRACT_MSG()	Converts the message passed by the framework (msg) to the message sent by the message helper function (mymsg).
ix_cc_msup_send_reply_msg()	Sends a reply message to the caller for asynchronous and synchronous call types.

26.1.1 [ix_cc_msup_init\(\)](#)

This function initializes the message support library and must be called once for each execution engine.

C Syntax

```
ix_error ix_cc_msup_init (
    ix_buffer_free_list_handle arg_hMsgFreelist,
    ix_uint32 arg_msgBuffSize,
    void **context);
```

Input

<code>arg_hMsgFreeList</code> <code>t</code>	handle of the message freelist
<code>art_msgBuffSize</code>	Size of the data area for buffers in <code>arg_hMsg_FreeList</code>
:	

Output/Returns

<code>context</code>	The context returned is passed to the fini function.
Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • A valid <code>ix_error</code>—the operation failed

26.1.2 `ix_cc_msup_fini()`

This function shuts down the message support library. It must be called once for each execution engine.

C Syntax

```
ix_error ix_cc_msup_fini (void *arg_pContext);
```

Input

<code>arg_pContext</code>	The context that was returned by the init function.
---------------------------	---

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • A valid <code>ix_error</code>—the operation failed
--------------	--

26.1.3 `ix_cc_msup_send_msg()`

Sends a fire-and-forget message and no response is made.

C Syntax

```
ix_error ix_cc_msup_send_msg (
    ix_uint32 arg_CommID,
    ix_uint32 arg_msgtype,
    void *arg_msg,
    ix_uint32 arg_msglen);
```

Input

<code>arg_CommID</code>	The ID specifying the destination.
<code>arg_msgtype</code>	The core component specific message type used.
<code>arg_msg</code>	The pointer to the message data.
<code>arg_msglen</code>	The length of the message data.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. A valid <code>ix_error</code>—the operation failed
--------------	--

26.1.4 `ix_cc_msup_send_async_msg()`

This function sends an asynchronous message. When the return message for this call is received, the callback function is invoked with `arg_pContext` as a parameter.

C Syntax

```
ix_error ix_cc_msup_send_async_msg (
    ix_uint32 arg_CommID,
    ix_cc_msghlp_callback *arg_pCallback,
    void *arg_pContext,
    ix_uint32 arg_msgtype,
    void *arg_msg,
    ix_uint32 arg_msglen);
```

The callback function is specified as follows:

C Syntax

```
ix_error (*ix_cc_msghlp_callback)(
    ix_error arg_Result,
    void *arg_pContext,
    void *arg_msg,
    ix_uint32 arg_msg_len);
```

Input

arg_CommID	This ID specifies the destination.
arg_pCallback	A pointer to the callback function to be called when a return message is received.
arg_pContext	A caller-provided context.
arg_msgtype	A core component specific message type.
arg_msg	A pointer to the message data.
arg_msglen	The length of the message data.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> IX_SUCCESS—the operation succeeded. A valid <code>ix_error</code>—the operation failed
--------------	---

26.1.5 ix_cc_msup_send_bcast_msg ()

Sends a fire-and-forget message to multiple targets and no response is be made.

C Syntax

```
ix_error ix_cc_msup_send_bcast_msg (
    ix_uint32 arg_CommID_List,
    ix_uint32 arg_msgtype,
    void *arg_msg,
    ix_uint32 arg_msglen);
```

Input

arg_CommID_List	List of commIDs to send the message.
arg_msgtype	core component specific message type used.

Input

<code>arg_msg</code>	Pointer to the data
<code>arg_msglen</code>	Length of the data.

.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. A valid <code>ix_error</code>—the operation failed
--------------	--

26.1.6 `ix_cc_msup_send_sync_msg()`

This function sends a synchronous message. A block of memory the size of the return message is allocated and its pointer is stored in `arg_return_msg`. This allows the API to return a variable sized block of data to be interpreted by the calling code.

C Syntax

```
ix_error ix_cc_msup_send_sync_msg (
    ix_uint32 arg_CommID,
    ix_uint32 arg_msgtype,
    void *arg_msg,
    ix_uint32 arg_msglen
    void ** arg_return_msg,
    uint32 *arg_return_msglen,
    ix_error *arg_ret_error);
```

Input

<code>arg_CommID</code>	An ID specifying the destination.
<code>arg_msgtype</code>	A core component specific message type.
<code>arg_msg</code>	A pointer to the message data.
<code>msglen</code>	The length of the message data.

Output

<code>arg_return_msg</code>	The location of the return message is placed in this pointer.
<code>arg_return_msg_len</code>	The length of the return message.
<code>arg_ret_error</code>	This is the <code>ix_error</code> returned by the remote call.

Output

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. A valid <code>ix_error</code>—the operation failed.
--------------	---

26.1.7 `IX_MSUP_EXTRACT_MSG()`

This macro is used at the beginning of the core components message handler. It converts the message passed by the framework (`msg`) to the message sent by the message helper function (`mymsg`). A context is also returned and should be stored and passed along with the return message.

26.1.8 `ix_cc_msup_send_reply_msg()`

This function sends a reply message to the caller for asynchronous and synchronous call types. If the message helper sent the original message as fire-and-forget, this call is ignored.

C Syntax

```
ix_error ix_cc_msup_send_reply_msg (
    void *arg_pContext,
    void *arg_return_msg,
    uint32 arg_return_msg_len,
    ix_error arg_ret_error);
```

Input

<code>arg_pContext</code>	The context returned from the <code>IX_MSUP_EXTRACT_MSG</code> macro.
<code>arg_return_msg</code>	A pointer to the return data.
<code>arg_return_msg_len</code>	The length of the return data.
<code>arg_ret_error</code>	An <code>ix_error</code> to return to the caller or 0 for success.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. A valid <code>ix_error</code>—the operation failed.
--------------	---

Stack Driver

The following chapter is included in this section:

- [Chapter 27, “Stack Driver”](#)

In SDK 3.1, the Stack Driver core component supports the IPv4 pipeline and does not support IPv6. Stack driver has been integrated with local TCP/IP stack for IPv4 pipeline running on VxWorks but not on Linux.

The Stack Driver provides a core component compatible abstraction of the operating system's network stack. It is a specialized type of core component that allows I/O to and from any network stack and other core components. It appears to the network stack as a media driver and appears to the IXA SDK as a core component. The main functions of the Stack Driver are:

- To send data received by the Stack Driver from an upstream core component up to the operating system's network stack or a remote stack.
- To send data received by the Stack Driver from the operating system's network stack or a remote stack to the bounded downstream core component.

The operating system's network stack with which the Stack Driver interacts can be either local or remote.

This chapter describes the design of the Stack Driver core component for the IXP2400 and IXP2800 network processors. The Stack Driver provides an interface between the IXA SDK and the local TCP/IP stack. The Stack Driver also supports packet transmission and basic interface configuration between the IXA SDK and the Control Plane PDK (CP PDK). However, it does not handle updates of forwarding tables.

This chapter focuses on an overview of the stack driver's high-level design but provides some low-level details, especially in the VxWorks-specific sections, as some of the components of the stack driver (that is, local VIDD) are OS-specific.

For complete details, see [Chapter 64, “Stack Driver”](#) of the *Intel® Internet Exchange Architecture (IXA) Software Building Blocks Developer's Manual*.

27.1 Core Component Module Data Structures and Types

Table 27-1 shows the stack driver core component data structures and types.

Table 27-1. Stack Driver Core Component Data Structures and Types

Name	Description
<code>ix_cc_stkdrv_packet_type</code>	Used to label a packet's destination—either to the local legacy stacks, local NPF stacks, remote legacy stacks, or remote NPF stacks.
<code>ix_cc_stkdrv_handler_id</code>	Identifies the type of an OS-dependent packet handler.
<code>ix_cc_stkdrv_handler_func</code>	During initialization, each handler object populates this data structure with function addresses and corresponding contexts.
<code>ix_cc_stkdrv_virtual_if</code>	This structure represents a node in the linked list of available virtual interfaces.
<code>ix_cc_stkdrv_physical_if_info</code>	This structure represents physical interface properties.
<code>ix_cc_stkdrv_physical_if_node</code>	This structure represents a node in the link list of available physical interfaces.
<code>ix_cc_stkdrv_handler_module</code>	This data structure describes a registered communication handler.
<code>ix_cc_stkdrv_fp_node</code>	This is a linked list of forwarding plane information structures.
<code>ix_cc_stkdrv_ctrl</code>	The structure represents the control block for the Stack Driver core component.

27.1.1 `ix_cc_stkdrv_packet_type`

Used to label a packet's destination—either to the local legacy stacks, local NPF stacks, remote legacy stacks, or remote NPF stacks. This data type is used by the core component, the communication handlers, and the packet classifier, as described in [Section 27.3.4, “Properties API.”](#)

C Syntax

```
/* enumeration for packet type */
typedef enum ix_e_cc_stkdrv_packet_type {
    IX_CC_STKDRV_PACKET_TYPE_FIRST = 0;
    IX_CC_STKDRV_PACKET_TYPE_LOCAL_LEGACY = IX_CC_STKDRV_PACKET_TYPE_FIRST,
    IX_CC_STKDRV_PACKET_TYPE_LOCAL_NPF,
    IX_CC_STKDRV_PACKET_TYPE_REMOTE_LEGACY,
    IX_CC_STKDRV_PACKET_TYPE_REMOTE_NPF,
    IX_CC_STKDRV_PACKET_TYPE_UNKNOWN,
    IX_CC_STKDRV_PACKET_TYPE_LAST
} ix_cc_stkdrv_packet_type;
```

27.1.2 ix_cc_stkdrv_handler_id

Identifies the type of an OS-dependent packet handler.

C Syntax

```
typedef enum ix_e_cc_stkdrv_handler_id {
    IX_CC_STKDRV_HANDLER_ID_FIRST = 0,
    IX_CC_STKDRV_HANDLER_ID_LOCAL_DRIVER = IX_CC_STKDRV_HANDLER_ID_FIRST,
    IX_CC_STKDRV_HANDLER_ID_TRANSP_MODULE,
    IX_CC_STKDRV_HANDLER_ID_UNDEFINED_HANDLER,
    IX_CC_STKDRV_HANDLER_ID_LAST
} ix_cc_stkdrv_handler_id;
```

27.1.3 `ix_cc_stkdrv_handler_func`

During initialization, each handler object populates the `ix_cc_stkdrv_handler_func` data structure with function addresses and corresponding contexts. The functions supplied in `ix_cc_stkdrv_handler_func` structure are used for:

- Communication Handler Packet Processing, see [Section 27.2.1](#)
- Communication Handler Message Processing, see [Section 27.2.2](#)
- Communication Handler Shutdown, see [Section 27.2.3](#)

C Syntax

```
typedef struct ix_s_cc_stkdrv_handler_func {
    ix_cc_stkdrv_packet_cb receive_pkt;
    ix_cc_stkdrv_msg_str_cb receive_msg_str;
    ix_cc_stkdrv_msg_int_cb receive_msg_int;
    ix_cc_stkdrv_handler_module_fini_cb fini;
    void* pPktContext;
    void* pMsgStrContext;
    void* pMsgIntContext;
    void* pFiniContext;
} ix_cc_stkdrv_handler_func;
```

Data Members

<code>receive_pkt</code>	The callback for receiving a packet.
<code>receive_msg_str</code>	The callback for receiving a string message.
<code>receive_msg_int</code>	The callback for receiving an integer message.
<code>fini</code>	The callback for finalizing the handler module.
<code>pPktContext</code>	A pointer to a packet context.
<code>pMsgStrContext</code>	A pointer to a string message context.
<code>pMsgIntContext</code>	A pointer to an integer message context.
<code>pFiniContext</code>	The context passed in to the finalizing function, <code>fini</code> , of the handler.

27.1.4 ix_cc_stkdrv_virtual_if

This structure represents a node in the linked list of available virtual interfaces. It is the internal representation of virtual interfaces existing on the forwarding plane. See also [ix_cc_ip_version](#) type in [Section 2.2.2.1.4](#) and the [ix_cc_ip_properties](#) data structure in [Section 2.2.2.1.5](#).

C Syntax

```
typedef struct ix_s_cc_stkdrv_virtual_if ix_cc_stkdrv_virtual_if;
struct ix_s_cc_stkdrv_virtual_if {
    ix_uint8                name[IX_CC_STKDRV_DEV_NAME_LEN];
    ix_uint8                macAddr[IX_CC_MAC_ADDR_LEN];
    ix_cc_ip_version        ipVersion;
    ix_cc_ip_properties     ipProp;
    ix_cc_stkdrv_virtual_if *pNextVirtualIf;};
```

Data Members

name	The virtual interface name.
macAddr	The MAC address of the interface.
ipVersion	The interface—IPv4 or IPv6.
ipProp	The IP address—subnet mask, gateway, and broadcast.
pNextVirtualIf	The pointer—that is, the link—to the next interface structure in the list of the structures.

27.1.5 ix_cc_stkdrv_physical_if_info

This structure represents physical interface properties.

C Syntax

```
struct ix_s_cc_stkdrv_physical_if_info {
    ix_uint32 fpId;
    ix_uint32 portId;
    ix_cc_physical_if_status physical_if_status;
    ix_media_type mediaType;
    ix_uint16 MTU;
    ix_uint32 linkSpeed;
    ix_cc_stkdrv_virtual_if* pVirtualIfs;
};
```

Data Members

fpId	The forwarding plane ID.
portId	The port number of the interface.
physical_if_status	Status of the port—up or down, and so on. This value is modified when an application brings the port up or down using the property API.
mediaType	The media type of the interface—one of <i>Fast Ethernet</i> , <i>1GB Ethernet</i> , <i>ATM</i> , or <i>POS</i> .
MTU	The maximum transmission unit for this interface.
linkSpeed	The link speed in megabits per second.
pVirtualIfs	A pointer to the first node in a linked list of virtual interface nodes associated with this physical interface.

27.1.6 ix_cc_stkdrv_physical_if_node

This structure represents a node in the link list of available physical interfaces.

C Syntax

```
typedef struct ix_s_cc_stkdrv_physical_if_node ix_cc_stkdrv_physical_if_node;
struct ix_s_cc_stkdrv_physical_if_node {
    ix_cc_stkdrv_physical_if_info *pPhysicalIfInfo;
    ix_cc_stkdrv_physical_if_node *pNextPhysicalIf;
    void* pPacketContext;
    void* pMsgStrContext;
    void* pMsgIntContext;};
```

Data Members

pPhysicalIfInfo	A pointer to the data structure describing the physical interface represented by this link-list node.
pNextPhysicalIf	A pointer to the next node in the link list of physical interfaces.
pPacketContext	An interface-specific context that is passed to the virtual interface device driver for this physical interface when passing packets.
pMsgStrContext	An interface-specific context that is passed to the virtual interface device driver for this physical interface when passing string messages.
pMsgIntContext	An interface-specific context that is passed to the virtual interface device driver for this physical interface when passing integer messages.

27.1.7 ix_cc_stkdrv_handler_module

This data structure describes a registered communication handler. The structure maintains information to identify a handler. Each instance of this data structure forms a node in a linked list of handler module nodes.

C Syntax

```
typedef struct ix_s_cc_stkdrv_handler_module ix_cc_stkdrv_handler_module;
struct ix_s_cc_stkdrv_handler_module {
    ix_cc_stkdrv_handler_id id;
    ix_cc_stkdrv_handler_func* pHandler_func;
    ix_cc_stkdrv_handler_module* pNextHandler;};
```

Data Members

<code>id</code>	The ID of the handler.
<code>pHandler_func</code>	A pointer to the holder of packet and message callbacks.
<code>pNextHandler</code>	A pointer to the next structure in the list of handler modules.

27.1.8 `ix_cc_stkdrv_fp_node`

This is a linked list of forwarding plane information structures and supports up to 64 forwarding planes.

C Syntax

```
typedef struct ix_s_cc_stkdrv_fp_node ix_cc_stkdrv_fp_node;
struct ix_s_cc_stkdrv_fp_node {
    ix_uint32                fpId;
    ix_cc_stkdrv_physical_if_node* pPhysicalIfs;
    ix_uint32                numPorts;
    ix_cc_stkdrv_fp_node      * pNextFP; };
```

Data Members

<code>fpId</code>	The ID of the forwarding plane.
<code>pPhysicalIfs</code>	The first node of the linked list of physical interfaces for the forwarding plane.
<code>numPorts</code>	The number of ports on this forwarding plane.
<code>pNextFP</code>	The next node in the linked list of forwarding planes.

Note: The forwarding plane and physical interface core component structures are similar to or the same as the corresponding structures in the virtual interface device driver module. While it seems redundant, these representations are kept separately to maintain loose coupling between the virtual interface device driver and the core component. Data is shared between the virtual interface device driver and core component infrastructure as much as possible. It is, however, important that the design of the core component be independent of the virtual interface device driver data structures.

27.1.9 ix_cc_stkdrv_ctrl

This structure represents the control block for the core component. It is used as a context in the core component Infrastructure APIs described in [Section 27.3.1, “Core Component Infrastructure API.”](#) The filter control structure, described in [Section 27.4.1.11, “ix_cc_stkdrv_filter_ctrl,”](#) is only used if packet classification is enabled.

C Syntax

```
typedef struct ix_s_cc_stkdrv_ctrl {  
    ix_cc_stkdrv_handler_module *pHandlerList;  
    ix_cc_stkdrv_fp_node *pFPList;  
    ix_cc_stkdrv_filter_ctrl *pFilterCtrl;  
} ix_cc_stkdrv_ctrl;
```

Data Members

pHandlerList	A pointer to the first node in the linked list of all handlers. See Section 27.1.7.
pFPList	A pointer to the first node in the linked list of all forwarding planes. See Section 27.1.8.
pFilterCtrl	A pointer to the filter control structure. See Section 27.4.1.11.

27.2 Stack Driver Callback Prototypes

Table 27-2 shows the stack driver callback prototypes.

Table 27-2. Stack Driver Callback Prototypes

Name	Description
<code>ix_cc_stkdrv_packet_cb()</code>	Provides the Stack Driver core component with a callback to send packet data to the communication handler.
<code>ix_cc_stkdrv_msg_str_cb()</code>	Provides the Stack Driver core component with a callback to send byte string messages to the communication handler.
<code>ix_cc_stkdrv_msg_int_cb()</code>	Provides the Stack Driver core component with a callback to send integer messages to the communication handler.
<code>ix_cc_stkdrv_handler_module_fini_cb()</code>	Provides the Stack Driver core component with a callback to finalize a communication handler.

27.2.1 Communication Handler Packet Processing

27.2.1.1 `ix_cc_stkdrv_packet_cb()`

This function prototype provides the Stack Driver core component with a callback to send packet data to the driver. Each handler module must implement a packet processing function and provide the address of the function to the core component.

C Syntax

```
ix_error (*ix_cc_stkdrv_packet_cb)(
    ix_buffer_handle arg_hBuffer,
    void* arg_pContext,
    ix_cc_stkdrv_packet_type arg_packetType);
```

Input

<code>arg_hBuffer</code>	The handle to the IXA SDK representation of network packets.
<code>arg_pContext</code>	A pointer to the context. Depending on the operation, context is provided by either the handler context stored in the handler module structure or the port-specific context stored in the physical interface structure.
<code>arg_packetType</code>	The type of the packet. See Section 27.1.1 , “ <code>ix_cc_stkdrv_packet_type</code> .”

Output/Returns

Return Value	Returns a valid <code>ix_error</code> .
	<ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • A valid <code>ix_error</code>—the operation failed.

27.2.2 Communication Handler Message Processing

Callback function prototypes are provided to the Stack Driver core component in support of sending configuration and control messages to the communication handler. Two types of messages are defined—byte strings and integer values. Each driver module must implement such functions and provide addresses of the functions to the core component.

27.2.2.1 `ix_cc_stkdrv_msg_str_cb()`

This function prototype provides the Stack Driver core component with a callback to send byte string messages to the communication handler.

C Syntax

```
ix_error (*ix_cc_stkdrv_msg_str_cb)(
    ix_uint32 arg_id,
    const char* arg_pMsg,
    ix_uint32 arg_msgSize,
    void *arg_pContext);
```

Input

<code>arg_id</code>	The identification of the message.
<code>arg_pMsg</code>	A pointer to the string of bytes in the message.
<code>arg_msgSize</code>	The length of the message—pass in the actual number of bytes and do <i>not</i> make any addition for a null terminator character.
<code>arg_pContext</code>	A pointer to the context. Depending on the operation this is either a handler context stored in the handler module structure or a port-specific context stored in the physical interface structure.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> .
	<ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • A valid <code>ix_error</code>—the operation failed.

27.2.2.2 ix_cc_stkdrv_msg_int_cb()

This function prototype provides the Stack Driver core component with a callback to send integer messages to the communication handler.

C Syntax

```
typedef ix_error (*ix_cc_stkdrv_msg_int_cb)(
    ix_uint32 arg_msgId,
    ix_uint32 arg_value,
    void *arg_pContext);
```

Input

arg_msgId	The identification of the message.
arg_value	The message value.
arg_pContext	A pointer to the context. It is used by the Stack Driver core component for packet classification. Details on the use of the context is specified in a later release of this document.

Output/Returns

Return Value	Returns a valid ix_error. <ul style="list-style-type: none"> IX_SUCCESS—the operation succeeded. A valid ix_error—the operation failed.
--------------	---

27.2.3 Shutdown

This section describes the finalization handler for the Stack driver.

27.2.3.1 ix_cc_stkdrv_handler_module_fini_cb()

The function prototype provides the Stack Driver core component with a callback to finalize a communication handler.

C Syntax

```
typedef ix_error (*ix_cc_stkdrv_handler_module_fini_cb)(
    void *arg_pContext
);
```

Input

arg_pContext	A pointer to the handler-module context.
--------------	--

Output/Returns

Return Value	Returns a valid <code>ix_error</code> .
	<ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • A valid <code>ix_error</code>—the operation failed.

27.3 Core Component API

This section describes the external interface exposed by the Stack Driver core component.

27.3.1 Core Component Infrastructure API

Table 27-3 shows the Stack Driver core component infrastructure API.

Table 27-3. Stack Driver Core Component Infrastructure API

Name	Description
<code>ix_cc_stkdrv_init()</code>	Initializes the core component.
<code>ix_cc_stkdrv_fini()</code>	The termination function used to free Stack Driver core component memory and data structures.
<code>ix_cc_stkdrv_high_priority_pkt_handler()</code>	Packet processing function for high-priority signaling packets.
<code>ix_cc_stkdrv_low_priority_pkt_handler()</code>	Packet processing function for low-priority data packets.
<code>ix_cc_stkdrv_pkt_to_remote_handler()</code>	Sends incoming packets to the Transport module without any classification.
<code>ix_cc_stkdrv_msg_handler()</code>	Message processing function for the stack driver.

27.3.1.1 `ix_cc_stkdrv_init()`

Initializes the core component—creates a Stack Driver core component control structure, fills it in with values from the registry, and makes function calls to initialize and register communication handlers. If packet classification is enabled it also initializes packet filters.

C Syntax

```
ix_error ix_cc_stkdrv_init(
    ix_cc_handle arg_hCC,
    void **arg_ppContext);
```

Input

<code>arg_hCC</code>	The handle to the Stack Driver core component—this is later used to get other services from the core component infrastructure.
----------------------	--

Input/Output

<code>arg_ppContext</code>	A pointer to the location where the pointer to the control block (<code>ix_cc_stkdrv_ctrl</code>), allocated by the Stack Driver core component, is returned. The control block is internal to the Stack Driver core component. It contains information about Stack Driver internal data structures, allocated memory and other relevant information. This pointer is later used to free memory when it is passed into the <code>ix_cc_stkdrv_fini()</code> function at the time that the Stack Driver core component is destroyed. On input, this pointer contains the location of the input context passed in by the caller.
----------------------------	--

Output/Returns

Return Value	<p>Returns a valid <code>ix_error</code>.</p> <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • A valid <code>ix_error</code>—the operation failed • <code>IX_CC_ERROR_OOM_SYSTEM</code> • <code>IX_CC_STKDRV_ERROR_REPOSITORY_OPEN</code> • <code>IX_CC_STKDRV_ERROR_REPOSITORY_READ</code> • <code>IX_CC_STKDRV_ERROR_HANDLER_INIT</code> • <code>IX_CC_STKDRV_ERROR_CCI_ADD_HANDLER</code> • <code>IX_CC_STKDRV_ERROR_RM_ADD_HANDLER</code>
--------------	--

27.3.1.2 `ix_cc_stkdrv_fini()`

This is the termination function, used to free Stack Driver core component memory and data structures.

C Syntax

```
ix_error ix_cc_stkdrv_fini(
    ix_cc_handle arg_hCC,
    void *arg_pContext);
```

Input

<code>arg_hCC</code>	A handle to the core component to be finalized.
<code>arg_pContext</code>	A pointer to the control block memory allocated earlier in <code>ix_cc_stkdrv_init()</code> function.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • A valid <code>ix_error</code>—the operation failed • <code>IX_CC_STKDRV_ERROR_CCI_REMOVE_HANDLER</code> • <code>IX_CC_STKDRV_ERROR_RM_REMOVE_HANDLER</code> • <code>IX_CC_STKDRV_ERROR_HANDLER_FINI</code>
--------------	---

27.3.1.3 `ix_cc_stkdrv_high_priority_pkt_handler()`

This function is the packet processing function for high-priority signaling packets. This operation calls the internal function `_ix_cc_stkdrv_process_pkt()`.

C Syntax

```
ix_error ix_cc_stkdrv_high_priority_pkt_handler(
    ix_buffer_handle arg_hDataToken,
    ix_uint32 arg_ExceptionCode,
    void* arg_pComponentContext);
```

Input

<code>arg_hDataToken</code>	A handle to a buffer which contains exception packets from the IPv4 Forwarder core component.
<code>arg_ExceptionCode</code>	The exception code for the packet. Ignored in the Stack Driver core component.

Input (Continued)

<code>arg_pComponentContext</code>	The pointer to Stack Driver core component context that is passed to the core component when a packet arrives. This context is defined by the core component and passed to the Core Component Infrastructure through the <code>ix_cc_stkdrv_init()</code> function.
------------------------------------	---

Output/Returns

Return Value	<p>Returns a valid <code>ix_error</code>.</p> <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • A valid <code>ix_error</code>—the operation failed • <code>IX_CC_ERROR_ENTRY_NOT_FOUND</code> • <code>IX_CC_STKDRV_ERROR_CANNOT_CLASSIFY_PKT</code> • <code>IX_CC_STKDRV_ERROR_HANDLER_RECV</code>
--------------	---

27.3.1.4 `ix_cc_stkdrv_low_priority_pkt_handler()`

Packet processing function for low priority data packets. Calls the internal function `_ix_cc_stkdrv_process_pkt()`.

C Syntax

```
ix_error ix_cc_stkdrv_low_priority_pkt_handler(
    ix_buffer_handle arg_hDataToken,
    ix_uint32        arg_ExceptionCode,
    void*            arg_pComponentContext);
```

Input

<code>arg_hDataToken</code>	A handle to a buffer which contains exception packets from the IPV4 Forwarder core component.
<code>arg_ExceptionCode</code>	The exception code for the packet. Ignored in the Stack Driver core component.
<code>arg_pComponentContext</code>	A pointer to the Stack Driver core component context that is passed to the core component when a packet arrives. This context is defined by the core component and passed to core component infrastructure through the <code>ix_cc_stkdrv_init()</code> function.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • A valid <code>ix_error</code>—the operation failed • <code>IX_CC_ERROR_ENTRY_NOT_FOUND</code> • <code>IX_CC_STKDRV_ERROR_CANNOT_CLASSIFY_PKT</code> • <code>IX_CC_STKDRV_ERROR_HANDLER_RECV</code>
--------------	---

27.3.1.5 `ix_cc_stkdrv_pkt_to_remote_handler()`

This function sends all incoming packets to the Transport module without any classification. Calls the internal function `_ix_cc_stkdrv_process_pkt_to_remote()`.

C Syntax

```
ix_error ix_cc_stkdrv_pkt_to_remote_handler (
    ix_buffer_handle arg_hDataToken,
    ix_uint32 arg_ExceptionCode,
    void* arg_pComponentContext)
```

Input

<code>arg_hDataToken</code>	The handle to a buffer which contains exception packets from the IPv4 Forwarder core component.
<code>arg_ExceptionCode</code>	Exception code for the packet. Ignored in the Stack Driver core component.
<code>arg_pComponentContext</code>	A pointer to the Stack Driver core component context that is passed to the core component when a packet arrives. This context is defined by the core component and passed to Core Component Infrastructure through the <code>ix_cc_stkdrv_init()</code> function.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • A valid <code>ix_error</code>—the operation failed
--------------	--

27.3.1.6 `ix_cc_stkdrv_msg_handler()`

Message processing function for the stack driver. Handles port property updates and requests to get the number of ports or port properties.

C Syntax

```
ix_error ix_cc_stkdrv_msg_handler(
    ix_buffer_handle arg_hDataToken,
    ix_uint32 arg_UserData,
    void* arg_pComponentContext)
```

Input

<code>arg_hDataToken</code>	The buffer handle embedding information for the message passed in <code>arg_UserData</code> .
<code>arg_UserData</code>	The message type. Valid values are: <ul style="list-style-type: none"> • <code>IX_CC_COMMON_MSG_ID_PROP_UPDATE</code> • <code>IX_CC_STKDRV_MSG_ID_GET_NUM_PORTS</code> • <code>IX_CC_STKDRV_MSG_ID_GET_PROPERTY</code>
<code>arg_pComponentContext</code>	A pointer to the Stack Driver core component context that is passed to the core component when a message arrives. This context was defined by the core component and passed to core component infrastructure through the <code>ix_cc_stkdrv_init()</code> function.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • A valid <code>ix_error</code>—the operation failed • <code>IX_CC_ERROR_NULL</code> • <code>IX_CC_ERROR_ENTRY_NOT_FOUND</code> • <code>IX_CC_ERROR_OOM_SYSTEM</code>
--------------	--

27.3.2 Core Component Infrastructure Separation

This API abstracts the Core Component Infrastructure providing reuse of the code for the Stack Driver core component functionality in systems that do not use the Core Component Infrastructure. This API is summarized in [Table 27-4](#).

Table 27-4. Stack Driver Core Component Infrastructure Separation API

Name	Description
<code>_ix_cc_stkdrv_process_pkt()</code>	Processes a locally-destined packet.
<code>_ix_cc_stkdrv_process_pkt_to_remote()</code>	Sends a packet directly to the forwarding plane module for remote processing, forgoing any classification.

27.3.2.1 `_ix_cc_stkdrv_process_pkt()`

Processes a locally-destined packet.

C Syntax

```
ix_error _ix_cc_stkdrv_process_pkt(
    ix_buffer_handle arg_hDataToken,
    ix_cc_stkdrv_ctrl* arg_pStkdrvCtrl);
```

Input

<code>arg_hDataToken</code>	A handle to the packet.
<code>arg_pStkdrvCtrl</code>	The pointer to the Stack Driver core component control block.

Output/Returns

Return Value	<p>Returns a valid <code>ix_error</code>.</p> <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • A valid <code>ix_error</code>—the operation failed • <code>IX_CC_ERROR_ENTRY_NOT_FOUND</code> • <code>IX_CC_STKDRV_ERROR_CANNOT_CLASSIFY_PKT</code> • <code>IX_CC_STKDRV_ERROR_HANDLER_RECV</code>
--------------	---

27.3.2.2 `_ix_cc_stkdrv_process_pkt_to_remote()`

Sends a packet directly to the forwarding plane module for remote processing, forgoing any classification.

C Syntax

```
ix_error _ix_cc_stkdrv_process_pkt_to_remote(
    ix_buffer_handle arg_hDataToken,
    ix_cc_stkdrv_ctrl* arg_pStkdrvCtrl);
```

Input

`arg_hDataToken` A handle to the packet.

`arg_pStkdrvCtrl` A pointer to the Stack Driver core component control block.

Output/Returns

Return Value Returns a valid `ix_error`.

- `IX_SUCCESS`—the operation succeeded.
- A valid `ix_error`—the operation failed
- `IX_CC_ERROR_ENTRY_NOT_FOUND`
- `IX_CC_STKDRV_ERROR_HANDLER_RECV`

27.3.3 Packet and Message Processing API

In addition to the callback definition, each communication handler uses direct API calls for packet and message forwarding to the Stack Driver core component from the driver handler. This API is summarized in [Table 27-5](#).

Table 27-5. Stack Driver Packet and Message Processing API

Name	Description
<code>ix_cc_stkdrv_send_packet()</code>	Queues a packet for transmission.
<code>ix_cc_stkdrv_send_msg_str()</code>	Sends a string message from a handler module to the Stack Driver core component.
<code>ix_cc_stkdrv_send_msg_int()</code>	Sends an integer message from a handler module to the Stack Driver core component.

27.3.3.1 ix_cc_stkdrv_send_packet()

Queues a packet for transmission. This function is called by the handler module to the Stack Driver. This function invokes the downstream classifier function which classifies the outgoing packet to IPv4 or IPv6 core components. In a future implementation of the classifier function, it may perform a downstream classification to determine which output core component to send the packet to, for example, IPv4 or IPv6 core components, or any other core components.

C Syntax

```
ix_error ix_cc_stkdrv_send_packet(
    ix_buffer_handle arg_hBuffer,
    void* arg_pContext),
    ix_uint8 arg_pktType;
```

Input

arg_hBuffer	A handle to the packet to be transmitted.
arg_pContext	A pointer to the context. In this case it is a pointer to the core component port structure, ix_cc_stkdrv_physical_if_node .
arg_pktType	Type of packet. Legal values are IPv4 or IPv6.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> IX_SUCCESS—the operation succeeded. A valid <code>ix_error</code>—the operation failed IX_CC_ERROR_SEND_FAIL
--------------	--

27.3.3.2 ix_cc_stkdrv_send_msg_str()

Sends a string message from a handler module to the Stack Driver core component.

Note: In this release, there is no need to send any messages from a handler module to the Stack Driver core component so these functions are stubs.

C Syntax

```
ix_error ix_cc_stkdrv_send_msg_str(
    ix_cc_stkdrv_cc_module_msg_id arg_msgID,
    const char* arg_pMsg,
    ix_uint32 arg_MsgSize,
    void* arg_pContext);
```

Input

<code>arg_msgID</code>	The ID of the message.
<code>arg_pMsg</code>	A pointer to the message data.
<code>arg_MsgSize</code>	The size of the message in bytes.
<code>arg_pContext</code>	A pointer to the context.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. A valid <code>ix_error</code>—the operation failed
--------------	--

27.3.3.3 `ix_cc_stkdrv_send_msg_int()`

Sends an integer message from a handler module to the Stack Driver core component.

Note: In this release, there is no need to send any messages from a handler module to the Stack Driver core component so these functions are stubs.

C Syntax

```
ix_error ix_cc_stkdrv_send_msg_int(
    ix_cc_stkdrv_cc_module_msg_id arg_msgID,
    ix_uint32 arg_MsgValue,
    void* arg_pContext);
```

Input

<code>arg_msgID</code>	The ID of the message.
<code>arg_MsgValue</code>	The integer message value.
<code>arg_pContext</code>	A pointer to the context.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. A valid <code>ix_error</code>—the operation failed
--------------	--

27.3.4 Properties API

The Stack Driver must export APIs to set dynamic properties for which it is a master. See *Intel® Internet Exchange Architecture (IXA) Software Building Blocks Developer's Manual*, Section 64.3.1.3, "Synchronizing Properties."

Table 27-6. Stack Driver Properties API

Name	Description
<code>ix_cc_stkdrv_async_get_property()</code>	Returns the interface properties of a specific port.
<code>ix_cc_stkdrv_cb_get_property()</code>	The callback function prototype used by the function <code>ix_cc_stkdrv_async_get_property()</code> .
<code>ix_cc_stkdrv_async_get_num_ports()</code>	Returns the number of ports configured on the Stack Driver core component.
<code>ix_cc_stkdrv_cb_get_num_ports()</code>	The callback function prototype used by the function <code>ix_cc_stkdrv_async_get_num_ports()</code> .

27.3.4.1 `ix_cc_stkdrv_async_get_property()`

Returns the interface properties of a specific port.

C Syntax

```
ix_error ix_cc_stkdrv_async_get_property(
    ix_cc_stkdrv_cb_get_property arg_pCallback,
    void* arg_pContext,
    ix_uint32 arg_propId,
    ix_uint32 arg_portId);
```

Input

<code>arg_pCallback</code>	A pointer to the calling application callback function to invoke on completion of the request.
<code>arg_pContext</code>	A pointer to the client context.
<code>arg_propId</code>	The property IDs for the requested properties.
<code>arg_portId</code>	The ID of queried port. Pass the value, <code>IX_CC_GET_ALL_PORTS</code> , to get properties for all ports.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. A valid <code>ix_error</code>—the operation failed <code>IX_CC_ERROR_OOM_SYSTEM</code>
--------------	---

27.3.4.1.1 `ix_cc_stkdrv_cb_get_property()`

The callback function prototype used by the function `ix_cc_stkdrv_async_get_property()`.

C Syntax

```
typedef ix_error (*ix_cc_stkdrv_cb_get_property)(
    ix_error arg_Result,
    void *arg_pContext,
    ix_cc_properties* arg_pProperty);
```

Input

<code>arg_Result</code>	The error code returned from the messaging operation. Returns <code>IX_SUCCESS</code> if the operation succeeds or a valid <code>ix_error</code> if the operation fails.
<code>arg_pContext</code>	A pointer to the user context.

Input/Output

<code>arg_pProperty</code>	<p>A calling application-provided pointer to an interface-properties structure. The function returns this structure when the operation successfully returns.</p> <p>If the calling application requested <i>properties for all ports</i> this operation returns a pointer to a contiguous block of memory containing an array of property structures. This memory is only valid during this routine and must be copied into local memory.</p>
----------------------------	---

Output/Returns

Return Value	<p>Returns a valid <code>ix_error</code>.</p> <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • A valid <code>ix_error</code>—the operation failed
--------------	--

27.3.4.2 `ix_cc_stkdrv_async_get_num_ports()`

Returns the number of ports configured on the Stack Driver.

C Syntax

```
ix_error ix_cc_stkdrv_aync_get_num_ports(
    ix_cc_stkdrv_cb_get_num_ports* arg_pCallback,
    void* arg_pContext);
```

Input

<code>arg_pCallback</code>	A pointer to the calling application callback function that is invoked when the request completes.
<code>arg_pContext</code>	A pointer to the client context.

Output/Returns

<code>arg_pNumPorts</code>	The pointer to the unsigned integer value describing the number of ports.
----------------------------	---

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none">• <code>IX_SUCCESS</code>—the operation succeeded.• A valid <code>ix_error</code>—the operation failed• <code>IX_CC_ERROR_NULL</code>• <code>IX_CC_ERROR_OOM_SYSTEM</code>
--------------	---

27.3.4.2.1 `ix_cc_stkdrv_cb_get_num_ports()`

The callback function prototype used by the function `ix_cc_stkdrv_async_get_num_ports()`.

C Syntax

```
typedef ix_error (*ix_cc_stkdrv_cb_get_num_ports)(  
    ix_error arg_Result,  
    void *arg_pContext,  
    ix_uint32* arg_pNumPorts);
```

Input

<code>arg_Result</code>	The error code returned from the messaging operation.
<code>arg_pContext</code>	A pointer to the user context.

Output/Returns

<code>arg_pNumPorts</code>	A pointer to the number of ports is returned.
Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none">• <code>IX_SUCCESS</code>—the operation succeeded.• A valid <code>ix_error</code>—the operation failed

27.4 Packet Classifier

In order to classify the packet correctly, the Stack Driver core component has the flexibility to activate different types of filters and apply the filters based on certain ranges. These filters are not hard-coded because this would prevent filters from dynamically changing in the future.

27.4.1 Packet Classifier Data Structures

Packet Classifier data structures are summarized in [Table 27-7](#).

Table 27-7. Stack Driver Packet Classifier Data Structures

Name	Description
<code>ix_cc_stkdrv_filter_type</code>	Represents filter type.
<code>ix_cc_stkdrv_filter_priority</code>	Represents the priority of a filter—either high, medium, or low priority.
<code>ix_cc_stkdrv_ipv4_address_range</code>	Specifies an inclusive IPv4 address range.
<code>ix_cc_stkdrv_ipv6_address_range</code>	Specifies an inclusive IPv6 address range.
<code>ix_cc_stkdrv_port_range</code>	Specifies a port range.
<code>ix_cc_stkdrv_ipv4_range_filter</code>	Specifies an IPv4 range filter.
<code>ix_cc_stkdrv_ipv6_range_filter</code>	Specifies an IPv6 range filter.
<code>ix_cc_stkdrv_filter</code>	Represents a filter.
<code>ix_cc_stkdrv_filter_index_node</code>	Represents a node in a linked list of filter indices.
<code>ix_cc_stkdrv_filter_handle</code>	An opaque handle to a filter.
<code>ix_cc_stkdrv_filter_ctrl</code>	Specifies the control structure for all filters.

27.4.1.1 `ix_cc_stkdrv_filter_type`

Represents filter type. In this release, only the following filter types are supported:

- `IX_CC_STKDRV_FILTER_TYPE_IPV4_DIP_INPORT`
based on IPv4 destination IP and input port ID
- `IX_CC_STKDRV_FILTER_TYPE_IPV6_DIP_INPORT`
based on IPv6 destination IP and input port ID

C Syntax

```
typedef enum ix_e_stkdrv_filter_type
{
    IX_CC_STKDRV_FILTER_TYPE_FIRST = 0,
    /* IPv4 destination IP and input portID */
    IX_CC_STKDRV_FILTER_TYPE_IPV4_DIP_INPORT = IX_CC_STKDRV_FILTER_TYPE_FIRST,
    IX_CC_STKDRV_FILTER_TYPE_IPV6_DIP_INPORT,
    IX_CC_STKDRV_FILTER_TYPE_IPV4_5TUPLE,
    IX_CC_STKDRV_FILTER_TYPE_IPV4_6TUPLE,
    IX_CC_STKDRV_FILTER_TYPE_IPV6_SIMPLE,
    IX_CC_STKDRV_FILTER_TYPE_ARP_FILTER,
```

```

        IX_CC_STKDRV_FILTER_TYPE_ATM_FILTER,
        IX_CC_STKDRV_FILTER_TYPE_FRAME_RELAY,
        IX_CC_STKDRV_FILTER_TYPE_POS_FILTER,
        IX_CC_STKDRV_FILTER_TYPE_UNDEFINED,
        IX_CC_STKDRV_FILTER_TYPE_LAST
    } ix_cc_stkdrv_filter_type;

```

27.4.1.2 **ix_cc_stkdrv_filter_priority**

Represents the priority of a filter—either high, medium, or low priority. Higher priority filters are checked first during packet classification.

C Syntax

```

typedef enum ix_e_cc_stkdrv_filter_priority
{
    IX_CC_STKDRV_FILTER_PRIORITY_FIRST = 0,
    IX_CC_STKDRV_FILTER_PRIORITY_HIGH = IX_CC_STKDRV_FILTER_PRIORITY_FIRST,
    IX_CC_STKDRV_FILTER_PRIORITY_MED,
    IX_CC_STKDRV_FILTER_PRIORITY_LOW,
    IX_CC_STKDRV_FILTER_PRIORITY_LAST
} ix_cc_stkdrv_filter_priority;

```

27.4.1.3 **ix_cc_stkdrv_ipv4_address_range**

Specifies an inclusive IPv4 address range.

C Syntax

```

/* Structure for specifying an inclusive address range */
typedef struct ix_s_cc_stkdrv_ipv4_address_range
{
    ix_uint32 start;
    ix_uint32 end;
} ix_cc_stkdrv_ipv4_address_range;

```

Data Members

start	The start address, inclusive—in host order.
end	The end address, inclusive—in host order.

27.4.1.4 **ix_cc_stkdrv_ipv6_address_range**

Specifies an inclusive IPv6 address range.

C Syntax

```

typedef struct ix_s_cc_stkdrv_ipv6_address_range
{
    ix_uint128 start; /* Start address */

```

```

        ix_uint128 end;    /* End address */
    } ix_cc_stkdrv_ipv6_address_range;

```

Data Members

start	The start address, inclusive—in host order.
end	The end address, inclusive—in host order.

27.4.1.5 ix_cc_stkdrv_port_range

Specifies a port range.

C Syntax

```

typedef struct ix_s_cc_stkdrv_port_range {
    ix_uint32 start;
    ix_uint32 end;
} ix_cc_stkdrv_port_range;

```

Data Members

start	The start port ID.
end	The end port ID.

27.4.1.6 ix_cc_stkdrv_ipv4_range_filter

Specifies an IPv4 range filter.

C Syntax

```

typedef struct ix_s_cc_stkdrv_ipv4_range_filter {
    ix_cc_stkdrv_ipv4_address_range* pDstAddr;
    ix_cc_stkdrv_port_range *pInPorts;
    ix_uint16 flag;
} ix_cc_stkdrv_ipv4_range_filter;

```

```

/* Definitions of flag field */
#define IX_CC_STKDRV_FILTER_ALL_CLASS      0x00
#define IX_CC_STKDRV_FILTER_DIP_CLASS     0x01
#define IX_CC_STKDRV_FILTER_INPORT_CLASS  0x02

```

Data Members

pDstAddr	Pointer to ix_cc_stkdrv_ipv4_address_range address range structure.
pInPorts	Pointer to ix_cc_stkdrv_port_range port range structure.

Data Members

flag	Selects one or both fields for classification. Legal values are: <ul style="list-style-type: none"> IX_CC_STKDRV_FILTER_ALL_CLASS: Use all classifications. IX_CC_STKDRV_FILTER_DIP_CLASS: Classify by destination IP. IX_CC_STKDRV_FILTER_INPORT_CLASS: Classify by input port ID.
------	--

27.4.1.7 ix_cc_stkdrv_ipv6_range_filter

Specifies an IPv6 range filter.

C Syntax

```
typedef struct ix_s_cc_stkdrv_ipv6_range_filter
{
    ix_cc_stkdrv_ipv6_address_range* pDstAddrs;
    ix_cc_stkdrv_port_range *pInPorts;
    ix_uint16 flag; /* selects one or both fields for classification */
} ix_cc_stkdrv_ipv6_range_filter;
```

Data Members

pDstAddrs	Pointer to ix_cc_stkdrv_ipv6_address_range address range structure.
pInPorts	Pointer to ix_cc_stkdrv_port_range port range structure.
flag	Selects one or both fields for classification. Legal values are: <ul style="list-style-type: none"> IX_CC_STKDRV_FILTER_ALL_CLASS: Use all classifications. IX_CC_STKDRV_FILTER_DIP_CLASS: Classify by destination IP. IX_CC_STKDRV_FILTER_INPORT_CLASS: Classify by input port ID.

27.4.1.8 ix_cc_stkdrv_filter

Represents a filter.

C Syntax

```
typedef struct ix_s_cc_stkdrv_filter {
    ix_cc_stkdrv_handler_id handlerID;
    ix_cc_stkdrv_packet_type packetType;
    ix_cc_stkdrv_filter_type filterType;
    ix_cc_stkdrv_ipv4_range_filter* pFilterCriteria;
    ix_cc_stkdrv_ipv6_range_filter* pV6FilterCriteria;
} ix_cc_stkdrv_filter;
```

Data Members

handlerID	The communication handler associated with this filter. See Section 27.1.2 .
-----------	---

Data Members

packetType	The type of packet associated with this filter. See Section 27.1.1 .
filterType	The type of filter. See Section 27.4.1.1 .
pFilterCriteria	The IPv4 filter criteria. See Section 27.4.1.6 .
pV6FilterCriteria	The IPv6 filter criteria. See Section 27.4.1.6 .

27.4.1.9 ix_cc_stkdrv_filter_index_node

Represents a node in a linked list of filter indices.

C Syntax

```
typedef struct ix_s_cc_stkdrv_filter_index_node ix_cc_stkdrv_filter_index_node;
struct ix_s_cc_stkdrv_filter_index_node
{
    ix_uint32 filterIndex;
    ix_cc_stkdrv_filter_index_node* pPrev;
    ix_cc_stkdrv_filter_index_node* pNext;
};
```

Data Members

filterIndex	The value of the filter index.
pPrev	A pointer to the previous index in the list.
pNext	A pointer to the next index in the list.

27.4.1.10 ix_cc_stkdrv_filter_handle

An opaque handle to a filter. The handler content is filled in when a filter is added and passed in when a filter is removed.

C Syntax

```
typedef ix_handle ix_cc_stkdrv_filter_handle;
```

27.4.1.11 ix_cc_stkdrv_filter_ctrl

Specifies the control structure for all filters. It is passed in as a context to all of the API calls described in [Section 27.4.2, “Core Component Infrastructure API.”](#)

C Syntax

```
typedef struct ix_s_cc_stkdrv_filter_ctrl
{
```

```

ix_cc_stkdrv_filter filterArray[IX_CC_STKDRV_NUM_FILTERS];
ix_cc_stkdrv_filter_index_node*
    pUsedFilterIndices[IX_CC_STKDRV_FILTER_PRIORITY_LAST];
ix_cc_stkdrv_filter_index_node* pFreeFilterIndices;
ix_cc_stkdrv_filter_index_node
    aAllFilterIndices[IX_CC_STKDRV_NUM_FILTERS];
ix_uint32 filterCount;
}ix_cc_stkdrv_filter_ctrl;

```

Data Members

filterArray	Array containing the actual filter data.
pUsedFilterIndices	Array of linked lists of indices for filters in use - one linked list for each priority.
pFreeFilterIndices	Linked list of free filter indices.
aAllFilterIndices	Array of all filter index nodes. Entries from here are placed in either the free index list of an appropriate priority index list.
filterCount	The number of filters in use.

27.4.2 Core Component Infrastructure API

This section describes the core component infrastructure API supported by the Stack Driver core component. This interface is summarized in [Table 27-8](#).

Table 27-8. Stack Driver Core Component Infrastructure API

Name	Description
<code>ix_cc_stkdrv_cb_filter_ops()</code>	Callback function from all asynchronous filter operations.
<code>ix_cc_stkdrv_init_filters()</code>	Creates an empty array of filter handlers and initializes each priority list to NULL.
<code>ix_cc_stkdrv_async_add_filter()</code>	Adds a filter to the array of filters and also places the filter index into the appropriate priority list.
<code>ix_cc_stkdrv_async_remove_filter()</code>	Removes a filter from the array of filters and from the appropriate priority list.
<code>ix_cc_stkdrv_async_remove_all_filters()</code>	Removes all filters from the array of filters and empties all priority lists.
<code>ix_cc_stkdrv_async_modify_filter()</code>	Modifies a filter with new input filter criteria and data.
<code>ix_cc_stkdrv_classify_pkt()</code>	Classifies a packet using the installed packet filters.

27.4.2.1 `ix_cc_stkdrv_cb_filter_ops()`

Callback function from all asynchronous filter operations.

C Syntax

```

typedef ix_error (*ix_cc_stkdrv_cb_filter_ops)(
    ix_error arg_Result,

```

```

    void *arg_pContext,
    ix_cc_stkdrv_filter_handle* arg_pFilterHandle
);

```

Input

arg_Result	Error code returned from the messaging operation.
arg_pContext	Pointer to user context.
arg_pFilterHandle	Pointer to where the filter handle is returned. NULL if the pointer is irrelevant.

Output/Returns

Return Value	Returns a valid ix_error. <ul style="list-style-type: none"> IX_SUCCESS—the operation succeeded. A valid ix_error—the operation failed
--------------	--

27.4.2.2 ix_cc_stkdrv_init_filters()

Creates an empty array of filter handlers and initializes each priority list to NULL.

C Syntax

```

ix_error ix_cc_stkdrv_init_filters(
    ix_uint32 arg_filterArraySize,
    ix_cc_stkdrv_filter_ctrl *arg_pFilterCtrl);

```

Input

arg_filterArraySize	The size of the filter array to allocate.
---------------------	---

Output/Returns

arg_pFilterCtrl	A pointer to the filter control structure to initialize.
Return Value	Returns a valid ix_error. <ul style="list-style-type: none"> IX_SUCCESS—the operation succeeded. A valid ix_error—the operation failed

27.4.2.3 ix_cc_stkdrv_fini_filters()

This function finalizes the filter control structure.

C Syntax

```
ix_error ix_cc_stkdrv_fini_filters(
    ix_cc_stkdrv_filter_ctrl** arg_ppFilterCtrl);
```

Input/Output

arg_ppFilterCtrl	Location of the pointer to the filter control structure.
------------------	--

27.4.2.4 ix_cc_stkdrv_classify_pkt()

Classifies a packet using the installed packet filters. Packets that don't match any classification will be treated as local legacy packets.

C Syntax

```
ix_error ix_cc_stkdrv_classify_pkt (
    ix_buffer_handle arg_hBuffer,
    ix_cc_stkdrv_filter_ctrl *arg_pFilterCtrl,
    ix_cc_stkdrv_handler_id *arg_pHandlerID,
    ix_cc_stkdrv_packet_type *arg_pPacketType);
```

Input

`arg_hBuffer` A handle to the packet to be classified.

`arg_pFilterCtrl` A pointer to the filter control structure.

Output/Returns

`arg_pHandlerID` A pointer to the handler ID which is associated with the input packet's classification. The classification function determines to which communication handler the packet is sent.

`arg_pPacketType` A pointer to the packet type which is associated with the input packet's classification. The classification function determines the type of a packet—that is, local or remote legacy stack or local or remote NPF stack.

Return Value Returns a valid `ix_error`.

- `IX_SUCCESS`—the operation succeeded.
- A valid `ix_error`—the operation failed

27.4.3 Message Helper API

Table 27-9 lists the Stack Driver message helper API.

Table 27-9. Stack Driver Message Helper API

Name	Description
<code>ix_cc_stkdrv_async_add_filter()</code>	Adds a filter to the array of filters.
<code>ix_cc_stkdrv_async_remove_filter()</code>	Removes a filter from the array of filters.
<code>ix_cc_stkdrv_async_remove_all_filters()</code>	Removes all filters from the array of filters.
<code>ix_cc_stkdrv_async_modify_filter()</code>	Modifies a filter with new input filter criteria and data.

27.4.3.1 ix_cc_stkdrv_async_add_filter()

Adds a packet filter to the array of filters.

C Syntax

```
ix_error ix_cc_stkdrv_async_add_filter(
    ix_cc_stkdrv_cb_filter_ops arg_pCallback,
    void* arg_pContext,
    ix_cc_stkdrv_filter* arg_pFilter,
    ix_cc_stkdrv_filter_priority arg_priority);
```

Input

arg_pCallback	A pointer to the ix_cc_stkdrv_cb_filter_ops() callback function to be invoked once the message returns.
arg_pContext	A pointer to the client context.
arg_pFilter	Pointer to the filter to be added.
arg_priority	The priority for this filter—either high, medium, or low. During classification the higher priority filters are checked first.

Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> IX_SUCCESS—the operation succeeded. A valid <code>ix_error</code>—the operation failed IX_CC_ERROR_OOM_SYSTEM
--------------	---

27.4.3.2 ix_cc_stkdrv_async_remove_filter()

Removes a packet filter from the array of filters.

C Syntax

```
ix_error ix_cc_stkdrv_async_remove_filter(
    ix_cc_stkdrv_cb_filter_ops arg_pCallback,
    void* arg_pContext,
    ix_cc_stkdrv_filter_handle arg_filterHandle);
```

Input

arg_pCallback	A pointer to the ix_cc_stkdrv_cb_filter_ops() callback function to be invoked once the message returns.
arg_pContext	A pointer to the client context.

Input (Continued)

<code>arg_filterHandle</code>	An opaque handle to the filter to be removed. This handle is obtained from the initial call to <code>ix_cc_stkdrv_async_add_filter()</code> .
-------------------------------	---

Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • A valid <code>ix_error</code>—the operation failed • <code>IX_CC_ERROR_OOM_SYSTEM</code>
--------------	---

27.4.3.3 ix_cc_stkdrv_async_remove_all_filters()

Removes all packet filters from the list of filters.

C Syntax

```
ix_error ix_cc_stkdrv_async_remove_all_filters(
    ix_cc_stkdrv_cb_filter_ops arg_pCallback,
    void* arg_pContext);
```

Input

<code>arg_pCallback</code>	A pointer to the <code>ix_cc_stkdrv_cb_filter_ops()</code> callback function to be invoked once the message returns.
<code>arg_pContext</code>	A pointer to the client context.

Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • A valid <code>ix_error</code>—the operation failed • <code>IX_CC_ERROR_OOM_SYSTEM</code>
--------------	---

27.4.3.4 ix_cc_stkdrv_async_modify_filter()

Modifies a filter with new input filter criteria and data.

C Syntax

```
ix_error ix_cc_stkdrv_async_modify_filter(
    ix_cc_stkdrv_cb_filter_ops arg_pCallback,
    void* arg_pContext,
    ix_cc_stkdrv_filter* arg_pFilter,
```

```
ix_cc_stkdrv_filter_priority arg_priority,
ix_cc_stkdrv_filter_handle arg_filterHandle);
```

Input

arg_pCallback	A pointer to the <code>ix_cc_stkdrv_cb_filter_ops()</code> callback function to be invoked once the message returns.
arg_pContext	A pointer to the client context.
arg_pFilter	A pointer to the filter to be added.
arg_priority	The filter priority—either high, medium, or low. During classification higher priority filters are checked first.
arg_filterHandle	The handle of the filter to be modified.

Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> IX_SUCCESS—the operation succeeded. A valid <code>ix_error</code>—the operation failed IX_CC_ERROR_OOM_SYSTEM
--------------	---

27.4.4 Library API

These internal functions are called by the stack driver message handler to perform the operations of adding, removing, and modifying filters. [Table 27-10](#) lists the Stack Driver library API.

Table 27-10. Stack Driver Library API

Name	Description
<code>ix_cc_stkdrv_add_filter()</code>	Adds a packet filter to the array of filters and places the filter index into the appropriate priority list.
<code>ix_cc_stkdrv_remove_filter()</code>	Removes a packet filter from the array of filters and the appropriate priority list.
<code>ix_cc_stkdrv_remove_all_filters()</code>	Removes all packet filters from the list of filters.
<code>ix_cc_stkdrv_async_modify_filter()</code>	Modifies a filter with new input filter criteria and data.

27.4.4.1 `ix_cc_stkdrv_add_filter()`

Adds a packet filter to the array of filters and places the filter index into the appropriate priority list.

C Syntax

```
ix_error _ix_cc_stkdrv_add_filter(
    ix_cc_stkdrv_filter* arg_pFilter,
    ix_cc_stkdrv_filter_priority arg_priority,
    ix_cc_stkdrv_filter_ctrl* arg_pFilterCtrl,
    ix_cc_stkdrv_filter_handle* arg_pFilterHandle);
```

Input

<code>arg_pFilter</code>	Pointer to the filter to be added.
<code>arg_priority</code>	The priority for this filter—either high, medium, or low. During classification the higher priority filters are checked first.
<code>arg_pFilterCtrl</code>	Pointer to the filter control structure.

Output/Returns

<code>arg_pFilterHandle</code>	Pointer to the handle of the added filter. The handle will be passed in to <code>ix_cc_stkdrv_remove_filter()</code> to remove the filter.
Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • A valid <code>ix_error</code>—the operation failed • <code>IX_CC_STKDRV_ERROR_OUT_OF_FILTERS</code> • <code>IX_CC_ERROR_RANGE</code> • <code>IX_CC_ERROR_INTERNAL</code>

27.4.4.2 `ix_cc_stkdrv_remove_filter()`

Removes a packet filter from the array of filters and the appropriate priority list.

C Syntax

```
ix_error _ix_cc_stkdrv_remove_filter(
    ix_cc_stkdrv_filter_handle arg_filterHandle,
    ix_cc_stkdrv_filter_ctrl* arg_pFilterCtrl);
```

Input

<code>arg_filterHandle</code>	An opaque handle to the filter to be removed. This handle is obtained from the initial call to <code>ix_cc_stkdrv_async_add_filter()</code> .
-------------------------------	---

Input (Continued)

`arg_pFilterCtrl` Pointer to the filter control structure.

Returns

Return Value Returns a valid `ix_error`.

- `IX_SUCCESS`—the operation succeeded.
- A valid `ix_error`—the operation failed
- `IX_CC_STKDRV_ERROR_INVALID_FILTER`
- `IX_CC_ERROR_INTERNAL`

27.4.4.3 `ix_cc_stkdrv_remove_all_filters()`

Removes all packet filters from the list of filters.

C Syntax

```
ix_error _ix_cc_stkdrv_remove_all_filters(
    ix_cc_stkdrv_filter_ctrl* arg_pFilterCtrl);
```

Input

`arg_pFilterCtrl` Pointer to the filter control structure.

Returns

Return Value Returns a valid `ix_error`.

- `IX_SUCCESS`—the operation succeeded.
- A valid `ix_error`—the operation failed

27.4.4.4 `ix_cc_stkdrv_async_modify_filter()`

Modifies a filter with new input filter criteria and data.

C Syntax

```
ix_error _ix_cc_stkdrv_modify_filter(
    ix_cc_stkdrv_filter* arg_pFilter,
    ix_cc_stkdrv_filter_priority arg_priority,
    ix_cc_stkdrv_filter_ctrl* arg_pFilterCtrl,
    ix_cc_stkdrv_filter_handle* arg_pFilterHandle);
```

Input

<code>arg_pFilter</code>	A pointer to the filter to be added.
<code>arg_priority</code>	The filter priority—either high, medium, or low. During classification higher priority filters are checked first.
<code>arg_pFilterCtrl</code>	Pointer to the filter control structure.

Returns

<code>arg_pFilterHandle</code>	Pointer to the handle of the added filter. The handle will be passed in to <code>ix_cc_stkdrv_remove_filter()</code> to remove the filter.
Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • A valid <code>ix_error</code>—the operation failed • <code>IX_CC_STKDRV_ERROR_OUT_OF_FILTERS</code> • <code>IX_CC_ERROR_RANGE</code> • <code>IX_CC_ERROR_INTERNAL</code> • <code>IX_CC_STKDRV_ERROR_INVALID_FILTER</code>

27.5 Outgoing Packet Classifier Design

The Outgoing Packet classifier is a logical block of the Stack Driver core component module that directs different type of packets to different core-components. For example, an outgoing IPv4 packet may be directed to IPv4 core-component and an IPv6 packet may be directed to IPv6 core-component. Outgoing packets can have multiple destinations. However, the current implementation shall support the following destination:

- IPv4 Core Component
- IPv6 Forwarder Core Component
- Queue Manager Core Component

27.5.1 Outgoing Packet Classifier Data Structures

Table 27-11 lists the data structures and data types used by the Outgoing Packet Classifier.

Table 27-11. Stack Driver Outgoing Packet Classifier Data Structures

Name	Description
<code>ix_cc_stkdrv_og_pkt_type</code>	Defines the packet type.
<code>ix_cc_stkdrv_og_filter_type</code>	Defines the filter type.
<code>ix_cc_stkdrv_og_cc_type</code>	Defines the outgoing core component type.
<code>ix_cc_stkdrv_port_range</code>	Defines the filter range.

Table 27-11. Stack Driver Outgoing Packet Classifier Data Structures

Name	Description
ix_cc_stkdrv_cb_og_chk_filter	Defines the function pointer for the actual filtering function.
ix_cc_stkdrv_og_filter	Defines the data structure for the filter.
ix_cc_stkdrv_og_comm_cc_map	Defines a map between the core component and communication ID.
ix_cc_stkdrv_og_filter_ctrl	Defines the control structure for the filters.

27.5.1.1 [ix_cc_stkdrv_og_pkt_type](#)

This enumeration defines the packet type.

C Syntax

```
typedef enum ix_e_cc_stkdrv_og_pkt_type
{
    IX_CC_STKDRV_OG_PKT_TYPE_FIRST = 0,
    IX_CC_STKDRV_OG_PKT_TYPE_IPV4,
    IX_CC_STKDRV_OG_PKT_TYPE_IPV6,
    IX_CC_STKDRV_OG_PKT_TYPE_UNAVAILABLE,
    IX_CC_STKDRV_OG_PKT_TYPE_UNDEFINED,
    IX_CC_STKDRV_OG_PKT_LAST
} ix_cc_stkdrv_og_pkt_type;
```

27.5.1.2 [ix_cc_stkdrv_og_filter_type](#)

This enumeration defines the filter_type.

Note: Any user-defined filter types should be added to this enumeration. Currently, a place holder has been provided to add one user-defined filter type.

C Syntax

```
typedef enum ix_e_cc_stkdrv_og_filter_type
{
    IX_CC_STKDRV_OG_FILTER_TYPE_FIRST = 0,
    IX_CC_STKDRV_OG_FILTER_TYPE_PKT,
    IX_CC_STKDRV_OG_FILTER_TYPE_USR_DEFINED_0,
    IX_CC_STKDRV_OG_FILTER_TYPE_UNDEFINED,
    IX_CC_STKDRV_OG_FILTER_LAST
} ix_cc_stkdrv_og_filter_type;
```

27.5.1.3 [ix_cc_stkdrv_og_cc_type](#)

This enumeration defines the outgoing core component type.

C Syntax

```
typedef enum ix_e_cc_stkdrv_og_cc_type
```

```

{
    IX_CC_STKDRV_OG_CC_TYPE_FIRST = 0,
    IX_CC_STKDRV_OG_CC_TYPE_IPV4_FWDER,
    IX_CC_STKDRV_OG_CC_TYPE_IPV6_FWDER,
    IX_CC_STKDRV_OG_CC_TYPE_Q_MGR,
    IX_CC_STKDRV_OG_CC_TYPE_UNDEFINED,
    IX_CC_STKDRV_OG_CC_TYPE_LAST
} ix_cc_stkdrv_og_cc_type;

```

27.5.1.4 ix_cc_stkdrv_port_range

This structure defines a port range.

C Syntax

```

typedef struct ix_s_cc_stkdrv_port_range
{
    ix_uint32 start;
    ix_uint32 end;
} ix_cc_stkdrv_port_range;

```

Data Members

start	The start port ID.
end	The end port ID.

27.5.1.5 ix_cc_stkdrv_cb_og_chk_filter

This prototype defines the function pointer for the actual filtering function.

C Syntax

```

typedef ix_error (*ix_cc_stkdrv_cb_og_chk_filter)(
    ix_buffer_handle arg_hBuffer,
    ix_uint8 arg_bufferType,
    ix_cc_stkdrv_physical_if_node *arg_pPhysicalIf,
    ix_cc_stkdrv_og_filter *arg_pFilter,
    ix_cc_stkdrv_og_cc_type *arg_pCcType,
    void **arg_ppData);

```

27.5.1.6 ix_cc_stkdrv_og_filter

Data structure for the filter.

C Syntax

```

typedef struct ix_s_cc_og_stkdrv_filter
{
    ix_cc_stkdrv_og_filter_type    filterType;

```

```

ix_cc_stkdrv_port_range      *pPortRange;
void                          *pCritereon;
ix_cc_stkdrv_cb_og_chk_filter aOgChkFunc;
}ix_cc_stkdrv_og_filter;

```

Data Members

filterType	The filter type value, see ix_cc_stkdrv_og_filter_type for details.
pPortRange	Associated port range for this filter.
pCritereon	The associated communication ID.
aOgChkFunc	Filter function pointer.

27.5.1.7 [ix_cc_stkdrv_og_comm_cc_map](#)

This structure defines a map between the core component and communication ID.

C Syntax

```

typedef struct ix_s_cc_stkdrv_og_comm_cc_map
{
    ix_cc_stkdrv_og_cc_type  ccType;
    ix_cc_communication_id   commId;
} ix_cc_stkdrv_og_comm_cc_map

```

Data Members

ccType	The type of core component.
commId	The associated communication ID.

27.5.1.8 [ix_cc_stkdrv_og_filter_ctrl](#)

This is the control structure for the filters.

C Syntax

```

typedef struct ix_s_cc_stkdrv_og_filter_ctrl
{
    ix_cc_stkdrv_og_filter  afilterArray[IX_CC_STKDRV_OG_FILTER_LAST];
    ix_cc_stkdrv_og_comm_cc_map aMap[IX_CC_STKDRV_OG_CC_TYPE_LAST];
    ix_uint32 filterCount;
}ix_cc_stkdrv_og_filter_ctrl;

```

Data Members

afilterArray	Array containing the actual filter data.
--------------	--

Data Members

aMap	Communication IDs in use in outgoing path.
filterCount	Number of filters in use.

27.5.2 Outgoing Packet Classifier Internal API Functions

This section lists API functions implemented to support outgoing packet classification. These API are exposed only to the stack driver module.

Table 27-12 lists the data structures and data types used by the Outgoing Packet Classifier.

Table 27-12. Stack Driver Outgoing Packet Classifier Internal API

Name	Description
<code>ix_cc_stkdrv_init_og_filters()</code>	Initializes the filter control structure with available filters.
<code>ix_cc_stkdrv_fini_og_filters()</code>	Clears the filter control structure.
<code>ix_cc_stkdrv_classify_output_id()</code>	Obtains the communicationId for the outgoing packet.

27.5.2.1 `ix_cc_stkdrv_init_og_filters()`

This function initializes the filter control structure with available filters, and communication ID information. This function shall be invoked from the stack-driver initialization routine,

C Syntax

```
ix_error ix_cc_stkdrv_init_og_filters(
    ix_cc_stkdrv_filter_ctrl **arg_ppFilterCtrl);
```

Output/Returns

arg_ppFilterCtrl The location of the pointer to the filter control structure.

Return Value Returns a valid `ix_error`.

- `IX_SUCCESS`—the operation succeeded.
- A valid `ix_error`—the operation failed
- `IX_CC_ERROR_INTERNAL`

27.5.2.2 `ix_cc_stkdrv_fini_og_filters()`

This function un-initializes the filter control structure. This function shall be invoked from the stack driver `ix_cc_stkdrv_fini()` function.

C Syntax

```
ix_error ix_cc_stkdrv_fini_og_filters(
    ix_cc_stkdrv_og_filter_ctrl** arg_ppFilterCtrl);
```

Input/Output

`arg_ppFilterCtrl` The location of the pointer to the filter control structure.

27.5.2.3 `ix_cc_stkdrv_classify_output_id()`

This function obtains the communication ID for the outgoing packet.

C Syntax

```
ix_error _ix_cc_stkdrv_classify_output_id(
    ix_buffer_handle      arg_hBuffer,
    ix_uint8              arg_pktType,
    ix_cc_communication_id *arg_pCommId,
    ix_cc_stkdrv_physical_if_node *arg_pPhysicalIf,
    void                  **arg_ppData)
```

Input

`arg_hBuffer` Handle to the packet to be classified.

`arg_pktType` Packet type. Legal values are IPv4 or IPv6.

`arg_pPhysicalIf` Handle to outgoing physical interface.

Output/Returns

`arg_pCommId` Pointer to outgoing communication ID. The classification function obtains the value of this communication ID.

`arg_ppData` Any data which may need to be returned back from the internal classification function.

Return Value Returns a valid `ix_error`.

- `IX_SUCCESS`—the operation succeeded.
- A valid `ix_error`—the operation failed
- `IX_CC_ERROR_INTERNAL`

27.6 VIDD for VxWorks*

The Virtual Interface Device Driver for VxWorks has the same design requirements as its counterpart for the Linux System. The goal is to expose the IXP ports as regular network interfaces to the Intel XScale® core OS TCP/IP stack. Essentially, the VIDD is a network device driver. There are two types of network drivers under VxWorks:

1. Enhanced Network Drivers
2. Traditional network drivers—that is, drivers supporting 4.3 BSD driver interface.

For details see *Intel® Internet Exchange Architecture (IXA) Software Building Blocks Developer's Manual*.

27.6.1 VIDD System Data Structures for VxWorks

The data structures described in this section are required—by the VxWorks kernel—to be implemented by any network driver. These data structures are summarized in [Table 27-13](#).

Note: See VxWorks documentation for more details on these data structures.

Table 27-13. Stack Driver VIDD System Data Structures

Name	Description
DEV_OBJ	The DEV_OBJ structure is the glue linking the device-generic END_OBJ structure with the device-specific data object referenced by pDevice.
END_ERR	This error data structure holds system and user defined errors and passes errors from virtual interface device driver to the control-plane protocol.
END_OBJ	Defines device-independent state that is maintained by all drivers and devices.
M2_INTERFACETBL	An M2_INTERFACETBL structure tracks the MIB-II variables used in the driver.

27.6.1.1 DEV_OBJ

The DEV_OBJ structure is the glue linking the device-generic END_OBJ structure, defined in [Section 27.6.1.3](#) with the device-specific data object referenced by pDevice. Defined by the VxWorks kernel.

C Syntax

```
typedef struct dev_obj
{
    char name[END_NAME_MAX];
    int unit;
    char description[END_DESC_MAX];
    void* pDevice;
} DEV_OBJ;
```

Data Members

name	The device name.
unit	The unit number of the interface.
description	A driver description.
pDevice	A pointer to the device data.

27.6.1.2 **END_ERR**

This error data structure holds system and user defined errors and is used to pass errors from virtual interface device driver to the control-plane protocol.

C Syntax

```
typedef struct end_err {
    IX_UINT32  errCode;
    char*  pMsg;
    void*  pSpare
} END_ERR;
```

The `errCode` member of the `END_ERR` structure The currently defined error codes are shown in [Table 27-14](#).

Data Members

errCode	A 32-bit error code. The lower 16 bits are reserved for system error messages. The upper 16 bits may be used for custom error messages.
pMsg	A NULL-terminated error message.
pSpare	A pointer to user-defined data.

Table 27-14. END_ERR Error Codes

END_ERR_INFO	This error is informational only.
END_ERR_WARN	A non-fatal error has occurred.
END_ERR_RESET	An error occurred that forced the device to reset itself, but the device has recovered.
END_ERR_DOWN	A fatal error occurred that forced the device to go down—that is, the device can no longer send or receive packets.
END_ERR_UP	The device was down but has now come up and may again send and receive packets.

The symbolic constants `OK` and `ERROR` are defined by VxWorks and appear throughout this section as return values.

27.6.1.3 END_OBJ

The core data structure for a device is the END object, or `END_OBJ`. This data structure defines device-independent state that is maintained by all drivers and devices. Each specific device derives from this object first and subsequently incorporates its own data structures. The driver allocates this structure and initializes some of its elements in the `endLoad()` function.

C Syntax

```
typedef struct end_object {
    NODE node;
    DEV_OBJ devObject;
    STATUS (*receiveRtn) ();
    struct net_protocol *outputFilter;
    void* pOutputFilterSpare;
    BOOL attached;
    SEM_ID txSem;
    long flags;
    struct net_funcs *pFuncTable;
    M2_INTERFACETBL mib2Tbl;
    LIST multiList;
    int nMulti;
    LIST protocols;
    int snarfCount;
    NET_POOL_ID pNetPool;
    M2_ID * pMib2Tbl;
} END_OBJ;
```

Data Members

<code>node</code>	The root of the device hierarchy.
<code>devObject</code>	The MUX sets the value of this member.
<code>receiveRtn</code>	The routine called on reception.
<code>outputFilter</code>	An optional output filter routine.
<code>pOutputFilterSpare</code>	A spare pointer to the output filter.
<code>attached</code>	Indicates the unit is attached.
<code>txSem</code>	The transmitter semaphore.
<code>flags</code>	Various flags.
<code>pFuncTable</code>	A pointer to the function table.
<code>mib2Tbl</code>	The MIBII counters.
<code>multiList</code>	The head of the multicast address list.
<code>nMulti</code>	The number of elements in the list.

Data Members (Continued)

<code>protocols</code>	The protocol node list.
<code>snarfCount</code>	The number of snarf protocols at the head of the list.
<code>pNetPool</code>	The memory cookie used by MUX buffering.
<code>pMib2Tbl</code>	The <i>RFC</i> 2233 MIB objects.

27.6.1.4 M2_INTERFACETBL

An `M2_INTERFACETBL` structure tracks the MIB-II variables used in the driver. The driver must initialize this structure, although the elements in the structure may be used and adjusted both by the driver and by the MUX.

C Syntax

```
typedef struct M2_INTERFACETBL_TAG {
    int ifIndex;
    char ifDescr[M2DISPLAYSTRSIZE];
    long ifType;
    long ifMtu;
    unsigned long ifSpeed;
    M2_PHYADDR ifPhysAddress;
    long ifAdminStatus;
    long ifOperStatus;
    unsigned long ifLastChange;
    unsigned long ifInOctets;
    unsigned long ifInUcastPkts;
    unsigned long ifInNUcastPkts;
    unsigned long ifInDiscards;
    unsigned long ifInErrors;
    unsigned long ifInUnknownProtos;
    unsigned long ifOutOctets;
    unsigned long ifOutUcastPkts;
    unsigned long ifOutNUcastPkts;
    unsigned long ifOutDiscards;
    unsigned long ifOutErrors;
    unsigned long ifOutQLen;
    M2_OBJECTID ifSpecific;
} M2_INTERFACETBL;
```

Data Members

<code>ifIndex</code>	The index.
<code>ifDescr</code>	A text description.
<code>ifType</code>	The type of the device, from <i>RFC 1158</i> .
<code>ifMtu</code>	The maximum transmission unit—that is, the maximum packet size.
<code>ifSpeed</code>	The speed in bits per second.
<code>ifPhysAddress</code>	The LLC address.
<code>ifAdminStatus</code>	Administration Status—this value is one of UP, DOWN, or TEST.
<code>ifOperStatus</code>	Operational Status—this value is one of UP, DOWN, or TEST.
<code>ifLastChange</code>	The last change of the interface.

Data Members (Continued)

<code>ifInOctets</code>	The number of octets received.
<code>ifInUcastPkts</code>	The number of unicast packets received.
<code>ifInNUcastPkts</code>	The number of broad and multicast packets received.
<code>ifInDiscards</code>	The number of input dicards.
<code>ifInErrors</code>	The number of input errors.
<code>ifInUnknownProtos</code>	The number of unknown packets.
<code>ifOutOctets</code>	The number of octets sent.
<code>ifOutUcastPkts</code>	The number of unicast packets sent.
<code>ifOutNUcastPkts</code>	The number of broad and multicast packets sent.
<code>ifOutDiscards</code>	The number of packets discarded.
<code>ifOutErrors</code>	The number of output errors.
<code>ifOutQLen</code>	The size of the output queue.
<code>ifSpecific</code>	This member is specific to the media used.

27.6.2 VIDD Local Data Structures for VxWorks

The Stack Driver VIDD local data structures are defined in this section. A summary of the VIDD local data structures is shown in [Table 27-15](#).

Table 27-15. Stack Driver VIDD Local Data Structures

Name	Description
<code>ix_cc_stkdrv_vidd_physical_if_node</code>	A list of data structures created upon initialization, which represents the hardware network interfaces of the Intel® IXDP2400 Advanced Development Platform base card.
<code>ix_cc_stkdrv_vidd_fp_node</code>	Represents a forwarding plane object.
<code>ix_cc_stkdrv_vidd_ctrl</code>	Represents the control information for interfaces and forwarding planes and is used as a context for all communication between the Stack Driver core component and the VIDD.

27.6.2.1 ix_cc_stkdrv_vidd_physical_if_node

A list of data structures created upon initialization by the VIDD, which represent the hardware network interfaces of the Intel® IXDP2400 Advanced Development Platform base card.

C Syntax

```
typedef struct ix_s_cc_stkdrv_vidd_physical_if_node {
    END_OBJ end;
    void* pMuxCookie;
    ix_cc_stkdrv_vidd_ctrl* pViddCtrl;
    void* pCCPktContext;
    ix_cc_stkdrv_physical_if_info* pPhysicalIfInfo;
    ix_cc_stkdrv_vidd_physical_if_node* pNextPhysicalIf;
} ix_cc_stkdrv_vidd_physical_if_node;
```

Data Members

end	The corresponding END_OBJ data structure.
pMuxCookie	A pointer to the cookie used for MUX communication.
pViddCtrl	A circular reference to the VIDD control structure.
pCCPktContext	A pointer to a context on the core-component side, used when passing VIDD packets to the core component.
pPhysicalIfInfo	A pointer to port information shared between the Stack Driver core component and the VIDD.
pNextPhysicalIf	The next interface structure in the linked list of structures.

27.6.2.2 ix_cc_stkdrv_vidd_fp_node

Represents a forwarding plane object. Its underlying design is a linked list which allows for future expansion of forwarding planes and points to the head of the list of virtual interfaces for each forwarding plane. This structure is created by the VIDD during initialization.

C Syntax

```
typedef struct ix_s_cc_stkdrv_vidd_fp_node ix_cc_stkdrv_vidd_fp_node;
struct ix_s_cc_stkdrv_vidd_fp_node {
    ix_uint32 fpId;
    ix_cc_stkdrv_vidd_physical_if_node* pPhysicalIfs;
    ix_uint32 numPorts;
    ix_cc_stkdrv_vidd_fp_node* pNextFp;
};
```

Data Members

<code>fp_Id</code>	The ID of the forwarding plane represented by the data structure.
<code>pPhysicalIfs</code>	The head node of the linked list of physical interfaces for the forwarding plane.
<code>numPorts</code>	The number of ports for the forwarding plane.
<code>pNextFp</code>	A pointer to the next forwarding plane in the list—in this release this pointer is always <code>NULL</code> .

27.6.3 ix_cc_stkdrv_vidd_ctrl

Represents the control information for interfaces and forwarding planes and is used as a context for all communication between the Stack Driver core component and the VIDD. This structure is defined by the driver and is created during initialization.

C Syntax

```
struct ix_s_cc_stkdrv_vidd_ctrl {
    ix_cc_stkdrv_vidd_fp_node* pFps;
    NET_POOL_ID pNetPool;
    void* pMclBlkCfg;
    void* pClustMem;
    ix_buffer_free_list_handle hFreeList;
};
```

Data Members

<code>pFps</code>	A pointer to the list of forwarding planes.
<code>pNetPool</code>	The memory cookie used by MUX buffering—this identifies the buffer pool that is shared among all VIDD interfaces.
<code>pMclBlkCfg</code>	A pointer to the <code>Mblk</code> memory area used to create the buffer pool. This pointer is stored at initialization and its memory must be freed at shutdown.
<code>pClustMem</code>	A pointer to the cluster memory area used to create the buffer pool. This pointer is stored at initialization and its memory must be freed at shutdown.
<code>hFreeList</code>	A handle to the <code>ix_buffer</code> freelist used to allocate buffers for transmit.

27.7 MUX Interface API

To support the MUX interface a VIDD driver must implement the system calls listed in [Table 27-16](#). The MUX data structure, `NET_FUNCS`, is defined in [Section 27.7.1](#).

Table 27-16. Required Functions for the Stack Driver MUX Interface

Function Name	Description
<code>nptLoad()</code>	Load a device into the MUX and associate a driver with the device.
<code>nptUnload()</code>	Release a device, or a port on a device, from the MUX.
<code>nptSend()</code>	Accept data from the MUX and send it on towards the physical layer.
<code>nptMCastAddrAdd()</code>	Add a multicast address to the list of those registered for the device.
<code>nptMCastAddrDel()</code>	Remove a registered multicast address from the list of those registered for the device.
<code>nptMCastAddrGet()</code>	Retrieve a list of multicast addresses registered for a device.
<code>nptPollSend()</code>	Send frames in polled mode rather than interrupt-driven mode.
<code>nptPollReceive()</code>	Receive frames in polled mode rather than interrupt-driven mode.
<code>nptStart()</code>	Connect device interrupts and activate the interface.
<code>nptStop()</code>	Stop or deactivate a network device or interface.
<code>nptIoctl()</code>	Support various ioctl commands.

Functions other than the *load* function are defined inside the `endLoad()` function by allocating the `NET_FUNC` data structure.

The MUX uses this structure to reference the functions implemented for a driver.

27.7.1 NET_FUNCS

The fields in this structure correspond with entry points for system calls for the VIDD driver.

Functions for `formAddress`, `packetDataGet`, and `addrGet` are not implemented for NPT drivers since NPTs deal with layer-3 packets and thus do not need to extract layer-2 headers from the packet, and so on.

C Syntax

```
typedef struct net_funcs {
    STATUS (* start)(END_OBJ*);
    STATUS (* stop)(END_OBJ*);
    STATUS (* unload)(END_OBJ*);
    int (* ioctl)(END_OBJ*, int, caddr_t);
    STATUS (* send)(END_OBJ*, M_BLK_ID);
    STATUS (* mCastAddrAdd) (END_OBJ*, char*);
    STATUS (* mCastAddrDel) (END_OBJ*, char*);
    STATUS (* mCastAddrGet) (END_OBJ*, MULTI_TABLE*);
    STATUS (* pollSend)(END_OBJ*, M_BLK_ID);
    STATUS (* pollRcv) (END_OBJ* pEND, M_BLK_ID pMblk,
        long* pNetSvc, long* pNetOffset,
        void* pSpareData)
    M_BLK_ID (* formAddress) (M_BLK_ID, M_BLK_ID, M_BLK_ID);
    STATUS (* packetDataGet) (M_BLK_ID, LL_HDR_INFO *);
    STATUS (* addrGet)(M_BLK_ID, M_BLK_ID, M_BLK_ID, M_BLK_ID, M_BLK_ID);
} NET_FUNCS;
```

The following entry points are defined for VIDD in `vidd.h`:

```
END_OBJ* ix_cc_stkdrv_vidd_npt_load(char*, void*);
STATUS ix_cc_stkdrv_vidd_npt_unload(END_OBJ*);
STATUS ix_cc_stkdrv_vidd_npt_start (END_OBJ*);
STATUS ix_cc_stkdrv_vidd_npt_stop (END_OBJ*);
int ix_cc_stkdrv_vidd_npt_ioctl(END_OBJ*, int, caddr_t);
STATUS ix_cc_stkdrv_vidd_npt_send(END_OBJ* pEnd, M_BLK_ID pMblk,
    char* dstMacAddr, long netType, void* pSpare);
STATUS ix_cc_stkdrv_vidd_npt_mCastAddrAdd (END_OBJ*, char*);
STATUS ix_cc_stkdrv_vidd_npt_mCastAddrDel (END_OBJ*, char*);
STATUS ix_cc_stkdrv_vidd_npt_mCastAddrGet (END_OBJ*, MULTI_TABLE*);
STATUS ix_cc_stkdrv_vidd_npt_pollSend (END_OBJ*, M_BLK_ID);
STATUS ix_cc_stkdrv_vidd_npt_pollRcv (END_OBJ*, M_BLK_ID);
```

27.7.2 VIDD System Function Calls

These function calls are implemented by the VIDD in order to conform to the MUX/END interface. These functions control all interactions between the VIDD and VxWorks:

- Configuration and initialization
- Shutdown processing
- Packet receiving
- Packet sending
- Memory allocation
- IOCTL handling

The addresses of these functions are in the `END_OBJ` structure so they can be called from the MUX library. The functions are summarized in [Table 27-17](#).

Table 27-17. VIDD System API

Name	Description
<code>ix_cc_stkdrv_vidd_npt_load()</code>	Creates a logical interface on the VIDD side and registers it with the VxWorks MUX layer.
<code>ix_cc_stkdrv_vidd_npt_unload()</code>	This function is called by the MUX to release the device.
<code>ix_cc_stkdrv_vidd_npt_start()</code>	Brings up the IXP network interface specified by the <code>END_OBJ</code> structure and makes the interface active and available to the OS.
<code>ix_cc_stkdrv_vidd_npt_stop()</code>	Brings down the IXP interface specified by the <code>END_OBJ</code> structure and deactivates the interface.
<code>ix_cc_stkdrv_vidd_npt_ioctl()</code>	Provides VIDD support for IOCTL commands.
<code>ix_cc_stkdrv_vidd_npt_send()</code>	The network protocol uses this function to send a network packet to the Stack Driver.
<code>ix_cc_stkdrv_vidd_npt_mCastAddrAdd()</code>	Adds a new link-layer multicast address to the table of multicast addresses for the IXP interface.
<code>ix_cc_stkdrv_vidd_npt_mCastAddrDel()</code>	Removes a previously added link-layer multicast address.
<code>ix_cc_stkdrv_vidd_npt_mCastAddrGet()</code>	Returns the list of all multicast addresses that are active on the interface.
<code>ix_cc_stkdrv_vidd_npt_pollSend()</code>	A polling mode equivalent to the <code>send()</code> routine.
<code>ix_cc_stkdrv_vidd_npt_pollRcv()</code>	The current implementation of the core component model does not support <code>PollReceive()</code> .

27.7.2.1 `ix_cc_stkdrv_vidd_npt_load()`

This function creates a logical interface on the VIDD side and registers it with the VxWorks MUX layer.

Note: Initially, one logical interface for each physical interface is supported. Future releases may support multiple logical interfaces per port. Application code does not call this function directly—the VxWorks MUX layer uses it as an entry point when adding a new physical interface.

This function is implemented as a two-pass algorithm. The MUX layer calls this function twice—once with an empty initialization string then a second time with the real initialization string.

C Syntax

```
END_OBJ* ix_cc_stkdrv_vidd_npt_load(char* initString, void* pBsp);
```

Input

<code>pBsp</code>	Contains optional BSP-specific information. This argument is used as a context—in this case a pointer to the VIDD control structure.
-------------------	--

Input/Output

<code>initString</code>	On the first pass of this function, <code>initString</code> is passed in as an empty allocated string, and the base name of the interface—for example, <code>eth</code> —is copied into it by the core component. On the second pass, <code>initString</code> contains the interface parameters for the logical interface being loaded—these interface parameters match the ones on the core component side. The function parses <code>initString</code> to fill in the logical interface data structure.
-------------------------	---

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • A valid <code>ix_error</code>—the operation failed
--------------	--

27.7.2.2 ix_cc_stkdrv_vidd_npt_unload()

This function is called by the MUX to *release* the device. The function is called for each port that has been activated by call to the `ix_cc_stkdrv_vidd_npt_load()`.

Note: In this release this function does not do anything.

C Syntax

```
STATUS ix_cc_stkdrv_vidd_npt_unload(END_OBJ* pEnd);
```

Input

pEnd	Pointer to END_OBJ data structure allocated by the Load() function.
------	---

Output/Returns

Return Value	The return value is one of: <ul style="list-style-type: none"> • OK • ERROR
--------------	---

27.7.2.3 ix_cc_stkdrv_vidd_npt_start()

Brings up the IXP network interface specified by the END_OBJ structure and makes the interface active and available to the operating system.

C Syntax

```
STATUS ix_cc_stkdrv_vidd_npt_start (END_OBJ* pEnd);
```

Input

pEnd	A pointer to END_OBJ data structure allocated by the Load() function.
------	---

Output/Returns

Return Value	The return value is one of: <ul style="list-style-type: none"> • OK • ERROR
--------------	---

27.7.2.4 ix_cc_stkdrv_vidd_npt_stop()

This function brings down the IXP interface specified by the END_OBJ structure and deactivates the interface.

C Syntax

```
STATUS ix_cc_stkdrv_vidd_npt_stop (END_OBJ* pEnd);
```

Input

pEnd A pointer to END_OBJ data structure allocated by the Load() function.

Output/Returns

Return Value The return value is one of:

- OK
- ERROR

27.7.2.5 ix_cc_stkdrv_vidd_npt_ioctl()

The VIDD needs to support IOCTL commands to keep the IOCTL interface with existing network protocols. [Table 27-18](#) gives the list of commonly-used IOCTL commands.

Table 27-18. IOCTL Commands

Command	Function	Data Type	Supported
SIOCGIFMTU	Get MTU.	char*	Yes
EIOCSFLAGS	Set device flags	int	Yes
EIOCGFLAGS	Get device flags	int	Yes
EIOCSADDR	Set device address	char*	No
EIOCGADDR	Get device address	char*	Yes
EIOCMULTIADD	Add multicast address	char*	No
EIOCMULTIDEL	Delete multicast address	char*	No
EIOCMULTIGET	Get multicast list	MULTI_TABLE*	No
EIOCPOLLSTART	Set device into polling mode	NULL	No
EIOCPOLLSTOP	Set device into interrupt mode	NULL	No
EIOCGFBUF	Get minimum first buffer for chaining	int	Yes
EIOCGHDRLEN	Get the size of the data link header	int	Yes
EIOCGNPT	Query a driver to determine whether it is a NPT driver	int	Yes
EIOCGMIB2233	retrieves the RFC2233 MIB II table	M2_ID *	Yes
EIOCGMIB2	Get the MIB-II counters from the driver	char*	Yes

In addition to the standard commands, which are in the range 0-128, the application or network protocol can define its own IOCTL commands to communicate with the driver.

C Syntax

```
int ix_cc_stkdrv_vidd_npt_ioctl(END_OBJ* pEnd, int command, caddr_t buffer);
```

Input/Output

<code>pEnd</code>	A pointer to the <code>END_OBJ</code> structure of the interface.
<code>command</code>	The IOCTL command
<code>buffer</code>	The character buffer holding the response from the command.

Output/Returns

Return Value	The return value is one of: <ul style="list-style-type: none"> • OK • ERROR
--------------	---

27.7.2.6 `ix_cc_stkdrv_vidd_npt_send()`

The MUX interface calls this function when the network protocol is sending a network packet. Network buffers are represented by a microblock chain in VxWorks. This function takes the packet stored in the microblock, performs all necessary checks on the packet, and stores the packet data in an `ix_buffer_handle` before calling the Stack Driver core component API `ix_cc_stkdrv_send_packet()` function.

C Syntax

```
STATUS ix_cc_stkdrv_vidd_npt_send(
    END_OBJ* pEnd,
    M_BLK_ID pMblk,
    char* dstMacAddr,
    long netType,
    void* pSpare);
```

Input

<code>pEnd</code>	A pointer to the <code>END_OBJ</code> structure identifying the transmit interface.
<code>pMblk</code>	A pointer to the microblock chain containing the network buffer. The data represents the full link-layer frame.
<code>dstMacAddr</code>	A destination MAC address from the OS stack. Ignored by this function as the IPv4 core component does a route lookup to obtain the destination MAC address.

Input (Continued)

<code>netType</code>	A network service type.
<code>pSpare</code>	A pointer to optional network service data.

Output/Returns

Return Value	The return value is one of: <ul style="list-style-type: none"> • OK • ERROR
--------------	---

27.7.2.7 `ix_cc_stkdrv_vidd_npt_mCastAddrAdd()`

This function adds a new link-layer multicast address to the table of multicast addresses for the IXP interface.

Note: In the initial release this call has no affect on the underlying hardware.

C Syntax

```
STATUS ix_cc_stkdrv_vidd_npt_mCastAddrAdd (
    END_OBJ * pEnd,
    char* pAddress);
```

Input

<code>pEnd</code>	A pointer to the <code>END_OBJ</code> structure to help identify the IXP port.
<code>pAddress</code>	A physical address to add to the Multicast table.

Output/Returns

Return Value	The return value is one of: <ul style="list-style-type: none"> • OK • ERROR
--------------	---

27.7.2.8 ix_cc_stkdrv_vidd_npt_mCastAddrDel()

The function removes a previously added link-layer multicast address

Note: In this release this function does not affect the underlying hardware.

C Syntax

```
STATUS ix_cc_stkdrv_vidd_npt_mCastAddrDel (
    END_OBJ* pEnd,
    char* pAddress);
```

Input

pEnd	A pointer to the END_OBJ structure to help identify the IXP port.
pAddress	The physical address to delete from the Multicast table.

Output/Returns

Return Value	The return value is one of: <ul style="list-style-type: none"> OK ERROR
--------------	---

27.7.2.9 ix_cc_stkdrv_vidd_npt_mCastAddrGet()

This function gets the list of all multicast addresses that are active on the interface.

C Syntax

```
STATUS ix_cc_stkdrv_vidd_npt_mCastAddrGet (
    END_OBJ * pEnd,
    MULTI_TABLE* pTable);
```

Input

pEnd	A pointer to the END_OBJ structure to help identify the IXP interface.
pTable	A pointer to the structure where the list is put.

Output/Returns

Return Value	The return value is one of: <ul style="list-style-type: none"> OK ERROR
--------------	---

27.7.2.10 ix_cc_stkdrv_vidd_npt_pollSend()

A polling mode equivalent to the `send()` routine. This routine transfers frames directly to the output core component or exits immediately if the core component is busy.

C Syntax

```
STATUS ix_cc_stkdrv_vidd_npt_pollSend (
    END_OBJ* pEND,
    M_BLK_ID pPkt,
    char* dstAddr,
    long netType,
    void * pSpareData);
```

Input

<code>pEnd</code>	A pointer to the <code>END_OBJ</code> structure. It identifies the transmit interface.
<code>pMblk</code>	A pointer to the microblock chain containing the network buffer. The data represents the full link-layer frame.
<code>dstMacAddr</code>	The destination MAC address from the OS stack. Ignored by this function as the VIDD does its own route table lookup to obtain the destination MAC address.
<code>netType</code>	The network service type.
<code>pSpare</code>	A pointer to optional network service data.

Output/Returns

Return Value	The return value is one of: <ul style="list-style-type: none"> ERROR
--------------	---

27.7.2.11 ix_cc_stkdrv_vidd_npt_pollRcv()

The current implementation of the core component model does not support `PollReceive()`.

C Syntax

```
STATUS ix_cc_stkdrv_vidd_npt_pollRcv (
    END_OBJ* pEND,
    M_BLK_ID pMblk,
    long* pNetSvc,
    long* pNetOffset,
    void* pSpareData);
```

Input

<code>pEnd</code>	A pointer to the <code>END_OBJ</code> structure. It identifies the transmit interface.
<code>pMblk</code>	A pointer to the microblock chain containing the network buffer. The data represents the full link layer frame.
<code>pNetSvc</code>	The payload/network frame type.
<code>pNetOffset</code>	The offset to the network frame.
<code>pSpare</code>	A pointer to optional network service data.

Output/Returns

Return Value	The return value is one of: <ul style="list-style-type: none"> • <code>ERROR</code>
--------------	--

27.7.3 MUX API used by the VIDD

These functions are exposed by MUX library and are called from the driver to the MUX interface to:

- Transfer data from driver to the protocol
- Report an error condition on the driver
- Restart network protocol to resume transmission, stopped previously because of error conditions

Table 27-19. VIDD MUX API

Name	Description
<code>muxTkReceive()</code>	Sends data to the control plane protocol atop the MUX interface.
<code>muxError()</code>	The VIDD calls this function to report an error condition to the network protocol.
<code>muxTxRestart()</code>	The VIDD calls this function to resume sending data from the network protocol—as, for example, when the protocol has stopped sending data when an error was returned from <code>muxSend()</code> .

27.7.3.1 `muxTkReceive()`

This function is called by the VIDD in order to send data to the control plane protocol on top of the MUX interface. If the function returns with an OK status, the driver can consider the data delivered and is not responsible for the associated memory buffers.

C Syntax

```
STATUS muxTkReceive (
    void* pCookie,
    M_BLK_ID pMblk,
    long netSvcOffset,
    long netSvcType,
    BOOL uniPromiscuous,
    void* pSpareData)
```

Input

<code>pEnd</code>	A pointer to the END data structure corresponding with the interface where packet came from.
<code>pMblk</code>	A pointer to the microblock chain containing the frame.
<code>netSvcOffset</code>	The offset to the network datagram in the packet.
<code>netSvcType</code>	The network service type.
<code>uniPromiscuous</code>	Set to <code>TRUE</code> when Driver is in promiscuous mode and <code>FALSE</code> otherwise.
<code>pSpareData</code>	A pointer to out-of-band data.

Output/Returns

Return Value	The return value is one of:
	<ul style="list-style-type: none"> • OK • ERROR

27.7.3.2 muxError()

The VIDDD calls this function to report an error condition to the network protocol. The MUX in turn calls the `stackErrorRtn()` callback of upper networking stack.

C Syntax

```
void muxError (END_OBJ* pEnd, END_ERR* pError)
```

Input

<code>pEnd</code>	A pointer to the END data structure.
<code>pError</code>	A pointer to the error information data structure.

27.7.3.3 muxTxRestart()

The VIDDD calls this function to resume sending data from the network protocol—as, for example, when the protocol has stopped sending data when an error was returned from `muxSend()`. The MUX in turn calls the `stackRestartRtn()` callback of the upper networking stack.

C Syntax

```
void muxTxRestart (END_OBJ* pEnd)
```

Input

<code>pEnd</code>	A pointer to the END data structure.
-------------------	--------------------------------------

27.8 VIDDD for Linux*

The local VIDDD implements all the functions called between the host's TCP/IP stack and the Stack Driver. These implementations are OS-specific and thus will include OS-specific data structures for net devices, ether devices, etc.

27.8.1 VIDD System Data Structures for Linux

Table 27-20 lists the data structures used by the VIDD from the Linux TCP/IP stack. These declarations only show the data members used in the VIDD.

Table 27-20. VIDD System Data Structures for Linux

Name	Description
<code>sk_buff</code>	The Linux socket buffer structure.
<code>net_device</code>	A container for the interactions between the Linux kernel and the VIDD.
<code>ifreq</code>	The interface request structure used for socket ioctl's.
<code>ix_cc_stkdrv_vidd_physical_if_node</code>	A VIDD proprietary structure which represents an interface on the VIDD.

27.8.1.1 `sk_buff`

This is the Linux socket buffer structure— packets are packaged in these before they are sent up the stack.

C Syntax

```
struct sk_buff {
    /* These two members must be first. */
    struct sk_buff * next;      /* Next buffer in list */
    struct sk_buff * prev;      /* Previous buffer in list */
    struct sk_buff_head * list; /* List we are on */
    struct sock * sk;           /* Socket we are owned by */
    struct timeval stamp;       /* Time we arrived */
    struct net_device * dev;    /* Device we arrived on/are leaving by */
    struct dst_entry * dst;
    /*
     * This is the control buffer. It is free to use for every
     * layer. Please put your private variables there. If you
     * want to keep them across layers you have to do a skb_clone()
     * first. This is owned by whoever has the skb queued ATM.
     */
    char cb[48];
    unsigned int len; /* Length of actual data */
    unsigned int data_len;
    unsigned int csum; /* Checksum */
    unsigned char __unused, /* Dead field, may be reused */
        cloned, /* head may be cloned (check refcnt to be sure). */
        pkt_type, /* Packet class */
        ip_summed; /* Driver fed us an IP checksum */
    __u32 priority; /* Packet queueing priority */
    atomic_t users; /* User count - see datagram.c, tcp.c */
    unsigned short protocol; /* Packet protocol from driver */
    unsigned short security; /* Security level of packet */
    unsigned int truesize; /* Buffer size */
    unsigned char * head; /* Head of buffer */
    unsigned char * data; /* Data head pointer */
    unsigned char * tail; /* Tail pointer */
    unsigned char * end; /* End pointer */
    void (*destructor)(struct sk_buff *); /* Destruct function */
};
```

27.8.1.2 net_device

This net device is a container for the interactions between the Linux kernel and the VIDD. It contains function pointers as data members. These functions are implemented in the VIDD.

C Syntax

```
typedef struct device net_device;
struct device
{
    /*
     * This is the first field of the "visible" part of
     * this structure. It is the name of the interface.
     */
    char *name;

    /* Low-level status flags. */
    volatile unsigned charstart; /* start an operation*/
    struct device*next;
    /* The device initialization function. Called only once. */
    int (*init)(struct device *dev);
    struct net_device_stats* (*get_stats)(struct device *dev);
    /*
     * This marks the end of the "visible" part of the
     * structure. All fields hereafter are internal to the
     * system, and may change at will (read: may be cleaned
     * up at will).
     */
    void *priv; /* pointer to private data*/

    unsigned chardev_addr[MAX_ADDR_LEN]; /*hw address*/
    unsigned charaddr_len; /*hardware address length*/
    /* Pointers to interface service routines.*/
    int (*open)(struct device *dev);
    int (*stop)(struct device *dev);
    int (*hard_start_xmit) (struct sk_buff *skb, struct device *dev);
#define HAVE_MULTICAST
    void (*set_multicast_list)(struct device *dev);
#define HAVE_PRIVATE_IOCTL
    int (*do_ioctl)(struct device *dev, struct ifreq *ifr, int cmd);
#define HAVE_SET_CONFIG
    int (*set_config)(struct device *dev, struct ifmap *map);
};
```

27.8.1.3 ifreq

This is the interface request structure used for socket ioctl's.

C Syntax

```
struct ifreq
{
#define IFHWADDRLEN 6
#define IFNAMSIZ 16
    union
    {
        char ifrn_name[IFNAMSIZ]; /* if name, e.g. "en0" */
    } ifr_ifrn;
};
```

```
union {
    struct sockaddr ifru_addr;
    struct sockaddr ifru_dstaddr;
    struct sockaddr ifru_broadaddr;
    struct sockaddr ifru_netmask;
    struct sockaddr ifru_hwaddr;
    short ifru_flags;
    int ifru_ival;
    int ifru_mtu;
    struct ifmap ifru_map;
    char ifru_slave[IFNAMSIZ]; /* Just fits the size */
    char ifru_newname[IFNAMSIZ];
    char * ifru_data;
} ifr_ifru;
};
```

27.8.1.4 ix_cc_stkdrv_vidd_physical_if_node

This data structure is proprietary to the VIDD and represents an interface on the VIDD.

C Syntax

```
typedef struct ix_s_cc_stkdrv_vidd_physical_if_node
ix_cc_stkdrv_vidd_physical_if_node;
struct ix_s_cc_stkdrv_vidd_physical_if_node
{
    /**
     * Corresponding net_device struct
     */
    struct net_device netdev;

    /* port statistics */
    struct net_device_stats stats;

    /* circular reference to VIDD control structure */
    ix_cc_stkdrv_vidd_ctrl *pViddCtrl;

    /* pointer to a context on the CC side, used when passing packets from the
    VIDD. */
    void *pCCPktContext;

    /* Pointer to port info shared between the CC Module and the VIDD. */
    ix_cc_stkdrv_physical_if_info *pPhysicalIfInfo;

    /* next interface structure in the list of the structures. */
    ix_cc_stkdrv_vidd_physical_if_node *pNextPhysicalIf;
};
```

27.8.2 VIDD System API for Linux

The driver functions listed in Table 27-21 are called by the Linux kernel to communicate with the driver module and are implemented in the VIDD.

Table 27-21. VIDD System API for Linux

Name	Description
<code>ix_cc_stkdrv_vidd_ifd_open</code>	An open function for the VIDD driver.
<code>ix_cc_stkdrv_vidd_ifd_stop</code>	A stop function for the VIDD driver.
<code>ix_cc_stkdrv_vidd_ifd_tx</code>	Sends a packet to the specified device.
<code>ix_cc_stkdrv_vidd_ifd_set_config</code>	Configures the VIDD driver.
<code>ix_cc_stkdrv_vidd_ifd_do_ioctl</code>	Sets the MAC address for each interface for private IOCTL calls.
<code>ix_cc_stkdrv_vidd_ifd_get_stats</code>	Retrieves interface statistics.
<code>ix_cc_stkdrv_vidd_ifd_set_multicast_list</code>	Sets multicast address and flag settings.
<code>ix_cc_stkdrv_vidd_ifd_init</code>	Initializes the VIDD driver.

27.8.2.1 `ix_cc_stkdrv_vidd_ifd_open`

Open function called by the kernel for this driver. This is for compatibility with older versions of the Linux operating system. This is useful if your development environment is using a slightly older version of Linux (for example, Red Hat 6.2) than the target Linux.

C Syntax

```
int ix_cc_stkdrv_vidd_ifd_open(struct net_device *arg_pDev)
```

Input

<code>arg_pDev</code>	A pointer to the <code>net_device</code> .
-----------------------	--

Output/Returns

Return Value	Zero.
--------------	-------

27.8.2.2 `ix_cc_stkdrv_vidd_ifd_stop`

Stop function called by the kernel for this driver. This is for compatibility with older versions. This is useful if your development environment is using a slightly older version of Linux (for example, Red Hat 6.2) than the target Linux.

C Syntax

```
int ix_cc_stkdrv_vidd_ifd_stop(struct net_device *arg_pDev)
```

Input

arg_pDev	A pointer to the net_device.
----------	------------------------------

Output/Returns

Return Value	Zero.
--------------	-------

27.8.2.3 ix_cc_stkdrv_vidd_ifd_tx

Function called by the kernel for this driver. The packet is sent to the output. (The output is typically bound to the IPv4 core component.) This function calls the `ix_cc_stkdrv_send_packet()` function implemented in the core component part.

C Syntax

```
int ix_cc_stkdrv_vidd_ifd_tx(
    struct sk_buff *arg_pSkb,
    struct net_device *arg_pDev)
```

Input

arg_pSkb	The sk_buffer containing the network buffer.
arg_pDev	A pointer to the net_device, identifying the transmit interface.

Output/Returns

Return Value	Returns zero to indicate success. Non-zero value indicates failure.
--------------	--

27.8.2.4 ix_cc_stkdrv_vidd_ifd_set_config

This function is the entry point for configuring the driver.

C Syntax

```
int ix_cc_skdrv_vidd_ifd_set_config(
    struct net_device *arg_pDev,
    struct ifmap *arg_pMap)
```

Input

<code>arg_pDev</code>	A pointer to the <code>net_device</code> .
<code>arg_pMap</code>	A pointer to the <code>ifmap</code> structure.

Output/Returns

Return Value	Zero.
--------------	-------

27.8.2.5 ix_cc_stkdrv_vidd_ifd_do_ioctl

This function is called whenever a private `ioctl` call is made for this driver. The only purpose it serves is to set the MAC address for each interface. It is recommended that the standard Linux `SIOCSIFHWADDR` `ioctl` call is used by developers.

C Syntax

```
int ix_cc_stkdrv_vidd_ifd_do_ioctl(
    struct net_device *arg_pDev,
    struct ifreq *arg_pReq,
    int arg_cmd)
```

Input

<code>arg_pDev</code>	A pointer to the <code>net_device</code> .
<code>arg_pReq</code>	A pointer to the <code>Ioctl</code> request structure.
<code>arg_cmd</code>	The private <code>ioctl</code> command value.

Output/Returns

Return Value	Returns zero to indicate success. Non-zero value indicates failure.
--------------	--

27.8.2.6 ix_cc_stkdrv_vidd_ifd_get_stats

This function is called whenever the kernel wants to get the statistics for a particular interface.

C Syntax

```
struct net_device_stats *ix_cc_stkdrv_vidd_ifd_get_stats(
    struct net_device *arg_pDev)
```


Input

arg_pDev A pointer to the net_device.

Output/Returns

Return Value Returns a pointer to the statistics structure.

27.8.2.7 ix_cc_stkdrv_vidd_ifd_set_multicast_list

This function is called whenever there are some changes in the flag settings for a particular interface. In future releases, when multicast support is fully implemented, this function will be called for changes in multicast addresses.

C Syntax

```
void ix_cc_stkdrv_vidd_ifd_set_multicast_list(  
    struct net_device *arg_pDev)
```

Input

arg_pDev A pointer to the net_device.

Output/Returns

Return Value None.

27.8.2.8 ix_cc_stkdrv_vidd_ifd_init

This is the initialization function called by the kernel to set up some default values.

C Syntax

```
int ix_cc_stkdrv_vidd_ifd_init(struct net_device *arg_pDev)
```

Input

arg_pDev A pointer to the net_device.

Output/Returns

Return Value Zero.

27.8.3 VIDD Linux Driver Support API

The driver functions listed in [Table 27-22](#) are called by the VIDD module to communicate with the OS kernel. Part of these procedures are called in response to the control and configuration messages received by the core component module.

Table 27-22. VIDD Linux Driver Support API

Name	Description
ix_cc_stkdrv_vidd_init	Initializes the local VIDD driver and registers entry points.
ix_cc_stkdrv_vidd_fini	Terminates the VIDD driver and frees any allocated memory.
ix_cc_stkdrv_vidd_if_devinet_ioctl	Wrapper function for protocol-dependent ioctl operation.
ix_cc_stkdrv_vidd_if_dev_ioctl	Wrapper function for protocol-independent ioctl operation.
ix_cc_stkdrv_vidd_if_up	Enables an interface.
ix_cc_stkdrv_vidd_if_down	Disables an interface.
ix_cc_stkdrv_vidd_receive_pkt	Transfers received packets to the TCP/IP stack.
ix_cc_stkdrv_set_ip_mask	Sets the IP and netmask for a given interface.

27.8.3.1 ix_cc_stkdrv_vidd_init

This function is called by the core component module to initialize the local VIDD and register the VIDD's entry points with the core component module.

C Syntax

```
ix_error ix_cc_stkdrv_vidd_init(
    ix_buffer_free_list_handle arg_hFreeList,
    ix_cc_stkdrv_fp_node *arg_pFP,
    void *arg_pInitData,
    ix_cc_stkdrv_handler_module *arg_pHandlerModule)
```

Input

arg_hFreeList	A handle to the freelist used to allocate buffers for passing packets down from the stack.
arg_pFP	A pointer to the FP structure used to get data for all ports.
arg_pInitData	A pointer to any additional initialization data that may be required by a specific handler module. Unused in this case.

Input (Continued)

`arg_pHandlerModule` The function will fill in this structure with its entry points and contexts.

Output/Returns

Return Value Returns a valid `ix_error`.

- `IX_SUCCESS`—the operation succeeded.
- A valid `ix_error`—the operation failed.
- `IX_CC_ERROR_OOM_SYSTEM`
- `IX_CC_STKDRV_VIDD_ERROR_KERNEL`

27.8.3.2 ix_cc_stkdrv_vidd_fini

This function is called by the core component module to shut down the local VIDD and free any memory allocated to it.

C Syntax

```
ix_error ix_cc_stkdrv_vidd_fini(void *arg_pContext)
```

Input

`arg_pContext` A pointer to context (in this case, the VIDD control structure).

Output/Returns

Return Value Returns a valid `ix_error`.

- `IX_SUCCESS`—the operation succeeded.
- A valid `ix_error`—the operation failed.

27.8.3.3 ix_cc_stkdrv_vidd_if_devinet_ioctl

This function is a wrapper for kernel mode IOCTL. If the stack driver is running in user mode, ioctl calls could be used to set various properties of the interface. The ioctl calls are divided in two classes by the Linux operating system. One is protocol-independent ioctl and other is protocol-dependent ioctl. This function is protocol-dependent ioctl, where the protocol is IP.

C Syntax

```
static int ix_cc_stkdrv_vidd_if_devinet_ioctl(
    unsigned int arg_cmd,
    struct ifreq *arg_pIfreq)
```

Input

<code>arg_cmd</code>	The ioctl command to be issued.
<code>arg_pIfreq</code>	A pointer to the interface request structure.

Output/Returns

Return Value	Returns zero to indicate success. Non-zero value indicates failure.
--------------	--

27.8.3.4 ix_cc_stkdrv_vidd_if_dev_ioctl

This function is a wrapper for kernel mode IOCTL. If the stack driver is running in user mode, ioctl calls could be used to set various properties of the interface. The ioctl calls are divided in two classes by the Linux operating system. One is protocol-independent ioctl and other is protocol-dependent ioctl. This function is protocol-independent ioctl.

C Syntax

```
static int ix_cc_stkdrv_vidd_if_dev_ioctl(
    unsigned int arg_cmd,
    struct ifreq *arg_pIfreq)
```

Input

<code>arg_cmd</code>	The ioctl command to be issued.
<code>arg_pIfreq</code>	A pointer to the interface request structure.

Output/Returns

Return Value	Returns zero to indicate success. Non-zero value indicates failure.
--------------	--

27.8.3.5 ix_cc_stkdrv_vidd_if_up

This function enables an interface.

C Syntax

```
int ix_cc_stkdrv_vidd_if_up(ix_cc_stkdrv_vidd_physical_if_node *arg_pIf)
```

Input

<code>arg_pIf</code>	A pointer to the interface.
----------------------	-----------------------------

Output/Returns

Return Value	Returns zero to indicate success. Non-zero value indicates failure.
--------------	--

27.8.3.6 `ix_cc_stkdrv_vidd_if_down`

This function disables an interface.

C Syntax

```
int ix_cc_stkdrv_vidd_if_down(
    ix_cc_stkdrv_vidd_physical_if_node *arg_pIf)
```

Input

<code>arg_pIf</code>	A pointer to the interface.
----------------------	-----------------------------

Output/Returns

Return Value	Returns zero to indicate success. Non-zero value indicates failure.
--------------	--

27.8.3.7 `ix_cc_stkdrv_vidd_receive_pkt`

This function transfers the received packets to the TCP/IP stack. The packet is packaged in `sk_buff` (it involves a copy) before sending it up to the TCP/IP stack.

C Syntax

```
ix_error ix_cc_stkdrv_vidd_receive_pkt(
    ix_buffer_handle arg_hBuffer,
    void* arg_pCtx,
    ix_cc_stkdrv_packet_type packetType
)
```

Input

<code>arg_hBuffer</code>	A handle to the input packet.
--------------------------	-------------------------------

Input

<code>arg_pCtx</code>	A pointer to the context, in this case, points to the VIDD control structure.
<code>packetType</code>	The type of packet being received—ignored for VIDD.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • A valid <code>ix_error</code>—the operation failed. • <code>IX_CC_ERROR_OOM_SYSTEM</code> • <code>IX_CC_ERROR_ENTRY_NOT_FOUND</code> • <code>IX_CC_ERROR_NULL</code> • <code>IX_CC_ERROR_INTERNAL</code>
--------------	--

27.8.3.8 ix_cc_stkdrv_set_ip_mask

This function sets the IP and netmask for a given interface.

C Syntax

```
int ix_cc_stkdrv_set_ip_mask(
    const ix_uint8 *arg_pName,
    int arg_addr,
    int arg_mask)
```

Input

<code>arg_pName</code>	The name of the interface.
<code>arg_addr</code>	The IP address to be set.
<code>arg_mask</code>	The netmask to be set.

Output/Returns

Return Value	Returns zero to indicate success. Non-zero value indicates failure.
--------------	--

27.9 Transport Module

The transport module sends packets to and receives packets from a remote TCP/IP stack.

27.9.1 Transport Data Structures

Table 27-23 lists the Transport Module data structure.

Table 27-23. Transport Module Data Structures

Name	Description
<code>ix_cc_stkdrv_tm_ctrl</code>	The control structure for the Transport module.

27.9.1.1 `ix_cc_stkdrv_tm_ctrl`

This is the control structure for the Transport module and it is used as a context in receiving packets from the forwarding plane module.

C Syntax

```
typedef struct ix_s_cc_stkdrv_tm_ctrl {
    ix_buffer_free_list_handle hFreeList;
} ix_cc_stkdrv_tm_ctrl;
```

Data Members

`hFreeList` A handle to free list from which to get `ix_buffer` for transmit. This can be expanded as necessary to increase transmit capacity.

27.9.2 Transport API

The Transport Module external API is summarized in Table 27-24.

Table 27-24. Transport Module External API

Name	Description
<code>ix_cc_stkdrv_tm_receive_pkt()</code>	Receives packets from the core component and passes them up to the forwarding plane module.
<code>ix_cc_stkdrv_tm_pkt_handler()</code>	Receives packets from the forwarding plane module and passes them down to the core component.

27.9.2.1 `ix_cc_stkdrv_tm_receive_pkt()`

Receives packets from the core component and passes them up to the forwarding plane module of the Intel® Control Plane Platform Development Kit. This function is called by the core component. Since the forwarding plane module makes use of buffer handles of type `ix_buffer_handle`, this function does not need to copy the packet data into another format.

C Syntax

```
ix_error ix_cc_stkdrv_tm_receive_pkt (  
    ix_buffer_handle arg_buffer,  
    void* arg_ctx,  
    ix_cc_stkdrv_packet_type arg_packetType);
```

Input

<code>arg_buffer</code>	A handle to the buffer to be received by the Transport module.
<code>arg_ctx</code>	A pointer to the context.
<code>arg_packetType</code>	The type of packet being received—the packet's destination stack type.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none">• <code>IX_SUCCESS</code>—the operation succeeded.• A valid <code>ix_error</code>—the operation failed.
--------------	--

27.9.2.2 ix_cc_stkdrv_tm_pkt_handler()

Receives packets from the forwarding plane module and passes them down to the core component. This function is called by the Forwarding Plane module. Since the Forwarding Plane module works with buffer handles of type `ix_buffer_handle`, this function does not need to copy the packet data into any other format.

C Syntax

```
ix_error ix_cc_stkdrv_tm_pkt_handler (
    ix_buffer_handle arg_hDataToken,
    ix_uint32 arg_UserData,
    void *arg_pComponentContext);
```

Input

<code>arg_hDataToken</code>	A handle to the buffer to be sent to the Core Component module.
<code>arg_UserData</code>	The user data for the packet.
<code>arg_pComponentContext</code>	A pointer to the Transport Module Core Component context.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. A valid <code>ix_error</code>—the operation failed
--------------	--

The following chapter is included in this section:

- [Chapter 28, “SoftSAR”](#)

The ATM SoftSAR Core Components cooperate together and include the following:

- The SAR Control Core Component is responsible for providing API for user application. It controls all core sub-components (ATM RX, ATM TX and TM4.1) so called SAR Control Plug-ins. It is responsible to for distributing user requests to appropriate sub-components in both single- and dual-IXP processor configurations.
- The ATM RX Core Component is responsible for symbol patching, data structures initialization and configuration of ATM RX microblock. Moreover, it must service exception messages with incoming AAL5 frames on any VC that has the exception flag set in the VC Reassembly Context.
- The TM4.1 Core Component is responsible for symbol patching, data structures initialization and configuration of TM4.1 shaper, scheduler and write-out microblocks. It does not service any exception messages.
- The ATM TX Core Component is responsible for symbol patching, data structures initialization and configuration of QM microblock. It does not service any exception messages.

This chapter includes the following core components APS:

- [Section 28.1, “SAR Core Components”](#)
- [Section 28.2, “ATM RX Core Components”](#)
- [Section 28.3, “ATM TX Core Components”](#)
- [Section 28.4, “TM4.1 Core Components”](#)

For more information on the ATM RX microblock and core component design see [Chapter 6, “ATM AAL5 RX Microblock.”](#) and [Chapter 65, “SoftSAR Core Components”](#) of *Intel® Internet Exchange Architecture Software Building Blocks Developer’s Manual*.

28.1 SAR Core Components

The SoftSAR core components provide the following functionalities:

- Utilization of SAR Plug-in API
- Provides front-end SAR Control API for user modules/applications
- Provides mechanism for registration of SAR Control plug-ins
- Controls SAR Control Agent (if it exists in the system)
- Distributes a user request to local SAR Control plug-ins and to SAR Control Agent (if it exists in the system)

28.1.1 Data Structures

[Table 28-1](#) lists the data structures and the data type definitions supported by AAL types for VC.

Table 28-1. SAR Data Structures and Data Type Definitions

Data Structures and Data Types	Description
VC-related Data Structures	Defines VC related data type
<code>ix_cc_atmsar_aal_t</code>	Defines VC related enumeration
<code>ix_cc_atmsar_vc_spec_t</code>	Defines VC identification parameters
<code>ix_cc_atmsar_service_class_t</code>	Defines supported service classes at VC creation
<code>ix_cc_atmsar_traffic_param_t</code>	Defines traffic parameters used during add VC
<code>ix_cc_atmsar_vc_descr_t</code>	Defines all VC parameters required to add VC
<code>ix_cc_atmsar_update_data_t</code>	Defines all VC parameters required for VC update
<code>ix_cc_atmsar_vc_handle_t</code>	Defines VC identification that is used in the system

Table 28-1. SAR Data Structures and Data Type Definitions

Data Structures and Data Types	Description
Port-Related Data Structures	Defines port related data structures
<code>ix_cc_atmsar_port_bandwidth_id_t</code>	Defines speed/bandwidth of port that can be set
<code>ix_cc_atmsar_port_descr_t</code>	Defines all port parameters required to add port

28.1.1.1 VC-related Data Structures

28.1.1.1.1 `ix_cc_atmsar_aal_t`

This is a VC-related data type.

C Syntax

```
enum {
    IX_CC_ATMSAR_AAL0,
    IX_CC_ATMSAR_AAL1,
    IX_CC_ATMSAR_AAL2,
    IX_CC_ATMSAR_AAL34,
    IX_CC_ATMSAR_AAL5,
    IX_CC_ATMSAR_AALRAW
} ix_cc_atmsar_aal_t;
```

28.1.1.1.2 `ix_cc_atmsar_vc_spec_t`

This data type defines VC identification parameters.

C Syntax

```
struct {
    ix_uint16 vpi;
    ix_uint16 vci;
    ix_uint16 portNo;
} ix_cc_atmsar_vc_spec_t;
```

28.1.1.1.3 `ix_cc_atmsar_service_class_t`

This data type defines support service classes at VC creation:

C Syntax

```
enum {
    IX_CC_ATMSAR_ATM_SERVICE_NONE = 0,
    IX_CC_ATMSAR_ATM_SERVICE_CBR,
    IX_CC_ATMSAR_ATM_SERVICE_GFR,
    IX_CC_ATMSAR_ATM_SERVICE_RT_VBR,
    IX_CC_ATMSAR_ATM_SERVICE_NRT_VBR,
    IX_CC_ATMSAR_ATM_SERVICE_ABR,
    IX_CC_ATMSAR_ATM_SERVICE_UBR,
} ix_cc_atmsar_service_class_t;
```

28.1.1.1.4 **ix_cc_atmsar_traffic_param_t**

This data type defines traffic parameters used during VC creation:

C Syntax

```
struct {
    ix_uint16    serviceClass;        // see: ix_cc_atmsar_service_class_t
    ix_uint8     bstEffortReq;        // best effort requested
    ix_cc_atmsar_tp_direction_t bwd; // backward direction
    ix_cc_atmsar_tp_direction_t fwd; // forward direction
} ix_cc_atmsar_traffic_param_t;
```

28.1.1.1.5 **ix_cc_atmsar_vc_descr_t**

This data type defines all VC parameters that are required for VC adding operation:

C Syntax

```
struct {
    ix_cc_atmsar_vc_spec_t spec;
    ix_cc_atmsar_traffic_param_t trafficParams;
    ix_cc_atmsar_oam_specific_t oam;
    ix_uint8 aalType;
    ix_uint8 hdrType;
    ix_uint16 freeListId;
    ix_uint16 l2port;
    ix_uint16 outputQueueNum;
#define IX_CC_ATMSAR_PACKET_PATH_REGULAR 0

#define IX_CC_ATMSAR_PACKET_PATH_EXCEPTION 1
    ix_uint8 packetPath;} ix_cc_atmsar_vc_descr_t;
} ix_cc_atmsar_vc_descr_t;
```

28.1.1.1.6 **ix_cc_atmsar_update_data_t**

This data type defines all VC parameters that are required for VC update operations:

C Syntax

```
struct {
    ix_uint32 lw0;
    ix_uint32 lw1;
} ix_cc_atmsar_update_data_t;
```

The contents of lw0 and lw1 is application specific. It may be used to specify entities like l2-port number or header type bound with this VC.

28.1.1.1.7 **ix_cc_atmsar_vc_handle_t**

This data type defines VC identification that is used in whole system:

C Syntax

```
typedef ix_uint16 ix_cc_atmsar_vc_handle_t;
```

The memPoolNum parameter defines a pool of buffers used for the VC. The outputQueueNum parameter is used to define output queue from SoftSAR to other microblocks or core components. VCs that service IP traffic have output queues other than VCs that service voice or signaling traffic.

28.1.1.2 Port-Related Data Structures

28.1.1.2.1 ix_cc_atmsar_port_bandwidth_id_t

This data type defines the speed/bandwidth options available for a port:

C Syntax

```
enum {  
    IX_CC_ATMSAR_PORT_BANDWIDTH_ID1=0,  
    IX_CC_ATMSAR_PORT_BANDWIDTH_ID2,  
    IX_CC_ATMSAR_PORT_BANDWIDTH_ID3,  
    IX_CC_ATMSAR_PORT_BANDWIDTH_ID4,  
    IX_CC_ATMSAR_PORT_BANDWIDTH_ID5,  
    IX_CC_ATMSAR_PORT_BANDWIDTH_ID6,  
    IX_CC_ATMSAR_PORT_BANDWIDTH_ID7,  
    IX_CC_ATMSAR_PORT_BANDWIDTH_ID8,  
    IX_CC_ATMSAR_PORT_BANDWIDTH_ID9,  
    IX_CC_ATMSAR_PORT_BANDWIDTH_ID10,  
    IX_CC_ATMSAR_PORT_BANDWIDTH_ID11,  
    IX_CC_ATMSAR_PORT_BANDWIDTH_ID12  
} ix_cc_atmsar_port_bandwidth_id_t;
```


C Syntax

```
#define IX_CC_ATMSAR_PORT_BANDWIDTH_DS0
        IX_CC_ATMSAR_PORT_BANDWIDTH_ID1
#define IX_CC_ATMSAR_PORT_BANDWIDTH_OC3
        IX_CC_ATMSAR_PORT_BANDWIDTH_ID8
#define IX_CC_ATMSAR_PORT_BANDWIDTH_OC12
        IX_CC_ATMSAR_PORT_BANDWIDTH_ID10
#define IX_CC_ATMSAR_PORT_BANDWIDTH_OC48
        IX_CC_ATMSAR_PORT_BANDWIDTH_ID12
```

Note: Limitations for port bandwidth settings are explained in detail in the [Chapter 23, “TM4.1 Shaper and Scheduler Microblock”](#) of *Intel® Internet Exchange Architecture Software Building Blocks Developer’s Manual*.

28.1.1.2.2 ix_cc_atmsar_port_descr_t

This data type defines all port parameters that required for a port adding operation:

C Syntax

```
struct {
    ix_uint16 portId;    // port ID
    ix_cc_atmsar_bandwidth_id_t bandwidth; // port bandwidth
} ix_cc_atmsar_port_descr_t;
```

28.1.2 Core Component Infrastructure API

Table 28-2 lists the SAR core component supported Core Component Infrastructure APIs.:

Table 28-2. SAR Core Component Infrastructure API

Name	Description
ix_cc_atmsar_init()	Initialize the core component
ix_cc_atmsar_fini()	Terminate the core component
ix_cc_atmsar_msg_handler()	Message handler for processing rule add/remove requests
ix_cc_atmsar_pkt_handler()	Packet handler for processing exception packets

28.1.2.1 ix_cc_atmsar_init()

This function initializes the core component. It should be called and returned successfully before any other function in the core component is called. This function performs the following tasks:

- Registers a packet handler [ix_cc_atmsar_pkt_handler\(\)](#) for inputs:
 - IX_CC_ATMSAR_ATMSAR_EXEPTION_PKT_INPUT (single source mode)
- Registers a message handler [ix_cc_atmsar_msg_handler\(\)](#) for inputs:
 - IX_CC_ ATMSAR_MSG_INPUT (multiple sources mode)
- Allocates a control block, and returns a pointer to this block in `arg_ppContext`.

C Syntax

```
ix_error ix_cc_atmsar_init(
    ix_cc_handle arg_hCcHandle,
    void **arg_ppContext);
```

I

Input

`arg_hCcHandle` Handle of the core component.

Input/Output

`arg_ppContext`

- As Input, it is a pointer to a generic init context which is used to extract system configuration information.
- As output, it is location where a pointer to the control block allocated by the core component is stored. The control block is internal to the core component and contains variables and internal data structures. This pointer is used later, and passed on to the [ix_cc_atmsar_fini\(\)](#) function by the Core Component Infrastructure to free memory when the core component is being terminated.

Output/Returns

Returns

Returns a valid `ix_error`.

- `IX_SUCCESS`—Operation completed successfully.
- `IX_CC_ERROR_FAILED_MEMORY_ALLOCATION`—operation failed, DRAM/ SRAM memory allocation failure.
- `IX_CC_ERROR_CCI`—operation failed, failure from the Core Component Infrastructure.

28.1.2.2 [ix_cc_atmsar_fini\(\)](#)

This function is called to terminate services from the core component. It frees memory allocated during initialization as well as used resources, including 64 bit counters.

Note: Calling any core component function after calling this function is not allowed.

C Syntax

```
ix_error ix_cc_atmsar_fini(
    ix_cc_handle arg_hCcHandle,
    void *arg_pContext);
```

Inputs

<code>arg_hCcHandle</code>	Handle of the core component.
<code>arg_pContext</code>	Pointer to the control block memory allocated earlier in <code>ix_cc_atmsar_init()</code> function.

Output/Returns

Returns	<p>Returns a valid <code>ix_error</code>.</p> <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—Operation completed successfully. • <code>IX_CC_ATMSAR_ERROR_INVALID_INPUT_PARAM</code>—operation failed, invalid input parameter(s). • <code>IX_CC_ERROR_OOM</code>—operation failed, memory freeing failure • <code>IX_CC_ERROR_OOM_64BIT_COUNTER</code>—operation failed, failure of 64 bit counter deletion. • <code>IX_CC_ATMSAR_ERROR_RTM</code>—operation failed, failure from RTM • <code>IX_CC_ERROR_CCI</code>—operation failed, failure from the Core Component Infrastructure
---------	--

28.1.2.3 ix_cc_atmsar_msg_handler()

This function is the message handler for the core component. The core component receives messages from other core component through this function, and internally it calls appropriate library function to process the message. This message handle is used to update dynamic properties.

C Syntax

```
ix_error ix_cc_atmsar_msg_handler(
    ix_buffer_handle arg_hDataToken,
    ix_uint32 arg_UserData,
    void *arg_pContext);
```

Input

<code>arg_hDataToken</code>	Buffer handle embedding information on the message passed in <code>arg_UserData</code> .
<code>arg_UserData</code>	Indicates the message type.
<code>arg_pContext</code>	Pointer to the control block memory allocated in the <code>ix_cc_atmsar_init()</code> function.

Outputs/Returns

Returns	<p>Returns a valid <code>ix_error</code>.</p> <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—Operation completed successfully • <code>IX_CC_ERROR_CCI</code>—operation failed, failure from the Core Component Infrastructure • <code>IX_CC_ERROR_NULL</code>—operation failed, null argument • <code>IX_CC_ERROR_UNDEFINED_MSG</code>—operation failed, unsupported message • <code>IX_CC_ATMSAR_ERROR_MSG_LIBRARY</code>—operation failed, error from message support buffer • <code>IX_CC_ATMSAR_ERROR_BUFFER_FREE</code>—operation failed, error from RM for freeing buffer
---------	---

28.1.2.4 `ix_cc_atmsar_pkt_handler()`

This is the only registered function to receive exception packets from high priority queue of ATM RX microblock. This function sends the packet to stack driver (i.e., receives only local delivery packets).

C Syntax

```
ix_error ix_cc_atmsar_pkt_handler(
    ix_buffer_handle arg_hBuffer,
    ix_uint32 arg_exceptionCode,
    void *arg_pContext);
```

Input

<code>arg_hBuffer</code>	Handle to a buffer which contains exception packets from ATM RX microblock.
<code>arg_exceptionCode</code>	Is ignored.
<code>arg_pContext</code>	Pointer to the control block memory allocated in the <code>ix_cc_atmsar_init()</code> function.

Output/Returns

Returns	<p>Returns a valid <code>ix_error</code>.</p> <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—Operation completed successfully • <code>IX_CC_ERROR_CCI</code>—operation failed, failure from the Core Component Infrastructure
---------	---

28.1.3 Messaging API

The Messaging API uses the Message Helper wrapper library that facilitates sending messages to the SoftSAR core component. One-to-one mapping exists between Message Helper APIs and the message types supported by a core component. All API functions are asynchronous; a core component reports operation status in a callback routine. [Table 28-3](#) lists the SAR Messaging API.

Table 28-3. SAR Messaging API

API Function	Description
ix_cc_atmsar_async_vc_create()	Adds VC
ix_cc_atmsar_async_vc_update()	Updates VC
ix_cc_atmsar_async_vc_remove()	Deletes VC
ix_cc_atmsar_async_port_create()	Adds port
ix_cc_atmsar_async_port_remove()	Deletes port
ix_cc_atmsar_async_get_vc_stats()	Gets VC statistics
ix_cc_atmsar_async_get_port_stats()	Gets port statistics

28.1.3.1 [ix_cc_atmsar_async_vc_create\(\)](#)

This function implements the messaging API for adding VC. It is an asynchronous function and the result of its operation is processed in a callback routine when the operation completes. The Core Component Infrastructure calls the callback routine and passes the status of operation and the VC handle assigned for the requested VC. The RXC VC entry is left with “valid bit not set” (it must be set by the update VC operation, see below.)

C Syntax

```
ix_error ix_cc_atmsar_async_vc_create(
    ix_cc_atmsar_instance_t arg_sarInstance,
    ix_cc_atmsar_vc_descr_t *arg_pVcDescr,
    ix_cc_atmsar_cb_vc_create_t arg_Callback,
    void *arg_pUserContext);
```

Input

<code>arg_sarInstance</code>	Instance of the core component
<code>arg_pVcDescr</code>	The pointer to the data structure describing the requested VC
<code>arg_Callback</code>	The pointer to the callback routine. See ix_cc_atmsar_cb_vc_create .
<code>arg_pUserContext</code> <code>t</code>	Pointer to the calling application-defined context.

Output/Returns

Returns	<p>Returns a valid <code>ix_error</code>.</p> <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—Operation completed successfully. • <code>IX_CC_ERROR_SEND_FAIL</code>—operation failed, failure sending a message. • <code>IX_CC_ATMSAR_ERROR_BUSY</code>—operation failed, the SAR CTRL does not have resources for the operation performed. • <code>IX_CC_ERROR_UNINIT</code>—operation failed, the core component has not been initialized. • <code>IX_CC_ATMSAR_ERROR_INVALID_PARAM</code>—operation failed, invalid pointer in input parameters. • <code>IX_NO_RESOURCES</code>—operation failed, no resources to add the entry.
---------	---

28.1.3.1.1 `ix_cc_atmsar_cb_vc_create`

The function prototype for the callback function used to notify the calling application of the results of a call to `ix_cc_atmsar_async_vc_create()`.

C Syntax

```
ix_error (*ix_cc_atmsar_cb_vc_create)(
    ix_error arg_Result,
    ix_cc_atmsar_vc_handle_t arg_hVc,
    void *arg_pUserContext);
```

Input

<code>arg_hVc</code>	The VC handle of the newly created VC
----------------------	---------------------------------------

Output/Returns

<code>arg_result</code>	The error code returned by SAR Control core component for the result of the VC update request.
Returns	<p>Returns a valid <code>ix_error</code>.</p> <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—returned if the operation succeeds • A valid <code>ix_error</code>—returned if the operation fails.

28.1.3.2 `ix_cc_atmsar_async_vc_update()`

This function implements the messaging API for updating a previously added VC. It is an asynchronous function and the result of operation is processed in a callback routine when the operation completes. The Core Component Infrastructure calls the callback routine and passes status of operation and the VC handle assigned for the requested VC. It checks if the RXC VC allocated bit is set. After the update is completed, the valid bit in the RXC entry is set. The function is used mainly to specify binding between ATM and forwarding layers.

C Syntax

```
ix_error ix_cc_atmsar_async_vc_update(
    ix_cc_atmsar_instance_t arg_sarInstance,
    ix_cc_atmsar_vc_handle_t arg_hVc,
    ix_cc_atmsar_update_data_t* arg_pUpdateData,
    ix_cc_atmsar_cb_vc_update arg_Callback,
    void *arg_pUserContext);
```

Input

<code>arg_sarInstance</code>	Instance of the core component.
<code>arg_hVc</code>	Handle to the VC.
<code>arg_pUpdateData</code>	Pointer to the data structure describing the updated VC.
<code>arg_Callback</code>	Pointer to the callback routine. See <code>ix_cc_atmsar_cb_vc_update</code> .
<code>arg_pUserContext</code>	Pointer to the calling application-defined context.

Output/Returns

Returns	<p>Returns a valid <code>ix_error</code>.</p> <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—returned if the operation succeeds. • <code>IX_CC_ERROR_SEND_FAIL</code>—operation failed, failure sending a message • <code>IX_CC_ATMSAR_ERROR_BUSY</code>—operation failed, the SAR Control core component does not have resources for the operation performed. • <code>IX_CC_ERROR_UNINIT</code>—operation failed, the core components has not been initialized. • <code>IX_CC_ATMSAR_ERROR_INVALID_PARAM</code>—operation failed, due to invalid pointer in input parameters
---------	--

28.1.3.2.1 `ix_cc_atmsar_cb_vc_update`

The function prototype for the callback function used to notify the calling application of the results of a call to `ix_cc_atmsar_async_vc_update()`.

C Syntax

```
ix_error (*ix_cc_atmsar_cb_vc_update)(
```

```
ix_error arg_Result,
void *arg_pUserContext);
```

Input

<code>arg_pUserContext</code> <code>t</code>	Pointer to the user provided context. The handle is valid if result of operation is <code>IX_SUCCESS</code> .
---	---

Output/Returns

<code>arg_result</code>	The error code returned by SAR Control core component for the result of the VC update request.
Returns	<p>Returns a valid <code>ix_error</code>.</p> <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—returned if the operation succeeds • A valid <code>ix_error</code>—returned if the operation fails.

28.1.3.3 `ix_cc_atmsar_async_vc_remove()`

This function implements the messaging API for deleting VC. It is an asynchronous function and the result of operation is processed in callback routine, when the operation is completed. The Core Component Infrastructure calls the callback routine and passes status of operation.

Syntax

```
ix_error ix_cc_atmsar_async_vc_remove(
    ix_cc_atmsar_instance_t arg_sarInstance,
    ix_cc_atmsar_vc_handle_t arg_hVc
    ix_cc_atmsar_cb_vc_remove_t arg_Callback,
    void *arg_pUserContext);
```

Input

<code>arg_sarInstance</code>	Instance of the core component.
<code>arg_hVc</code>	Handle to the VC.
<code>arg_Callback</code>	The pointer to the callback routine. See ix_cc_atmsar_cb_vc_remove .
<code>arg_pUserContext</code> <code>t</code>	Pointer to user-defined context.

Outputs/Returns

Returns	<p>Returns a valid <code>ix_error</code>.</p> <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—Operation completed successfully. • <code>IX_CC_ATMSAR_ERROR_INVALID_PARAM</code>—operation failed, invalid pointer in input parameters. • <code>IX_CC_ERROR_MSG_LIBRARY</code>—operation failed, failure on sending a message. • <code>IX_CC_ERROR_UNINIT</code>—operation failed, the core component has not been initialized.
---------	---

28.1.3.3.1 `ix_cc_atmsar_cb_vc_remove`

The function prototype for the callback function used to notify the calling application of the results of a call to `ix_cc_atmsar_async_vc_remove()`.

C Syntax

```
ix_error (*ix_cc_atmsar_cb_vc_remove)(
    ix_error arg_Result,
    void *arg_pUserContext);
```

Input

<code>arg_pUserContext</code> t	Pointer to the calling application-provided context. The handle is valid if result of operation is <code>IX_SUCCESS</code> .
------------------------------------	--

Output/Returns

<code>arg_result</code>	The error code returned by SAR Control core component for the result of the VC remove request.
Returns	<p>Returns a valid <code>ix_error</code>.</p> <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—returned if the operation succeeds • A valid <code>ix_error</code>—returned if the operation fails.

28.1.3.4 `ix_cc_atmsar_async_port_create()`

This function implements the messaging API for adding port. It is an asynchronous function and the result of operation is processed in callback routine, when the operation is completed. The Core Component Infrastructure calls the callback routine and passes status of operation and port handle assigned for the requested port.

Syntax

```
ix_error ix_cc_atmsar_async_port_create(
    ix_cc_atmsar_instance_t arg_sarInstance,
    ix_cc_atmsar_port_descr_t *arg_pPortDescr,
    ix_cc_atmsar_cb_add_port arg_Callback,
    void *arg_pUserContext);
```

Input

<code>arg_sarInstance</code>	Instance of the core component.
<code>arg_pPortDescr</code>	The pointer to data structure describing the requested port.
<code>arg_Callback</code>	The pointer to the callback routine. See ix_cc_atmsar_cb_add_port .
<code>arg_pUserContext</code> <code>t</code>	Pointer to user-defined context.

Output/Returns

Returns	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—Operation completed successfully. <code>IX_NO_RESOURCES</code>—operation failed, no resources to add the entry. <code>IX_CC_ATMSAR_ERROR_INVALID_PARAM</code>—operation failed, invalid pointer in input parameters. <code>IX_CC_ERROR_MSG_LIBRARY</code>—operation failed, failure sending a message. <code>IX_CC_ERROR_UNINIT</code>—operation failed, the core component has not been initialized.
---------	---

28.1.3.4.1 `ix_cc_atmsar_cb_add_port`

The function prototype for the callback function used to notify the calling application of the results of a call to `ix_cc_atmsar_async_port_create()`.

C Syntax

```
ix_error (*ix_cc_atmsar_cb_add_port)(
    ix_error arg_Result,
    ix_cc_atmsar_port_handle_t arg_hPort
    void *arg_pUserContext);
```

Input

<code>arg_hport</code>	Port handle to the newly created port
<code>arg_pUserContext</code> <code>t</code>	Pointer to the calling application-provided context. The handle is valid if result of operation is <code>IX_SUCCESS</code> .

Output/Returns

<code>arg_result</code>	The error code returned by SAR Control core component for the result of the port add request.
Returns	<p>Returns a valid <code>ix_error</code>.</p> <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—returned if the operation succeeds • A valid <code>ix_error</code>—returned if the operation fails.

28.1.3.5 `ix_cc_atmsar_async_port_remove()`

This function implements the messaging API for deleting a port. It is an asynchronous function and the result of operation is processed in callback routine, when the operation is completed. The Core Component Infrastructure calls the callback routine and passes status of operation.

Syntax

```
ix_error ix_cc_atmsar_async_port_remove(
    ix_cc_atmsar_instance_t arg_sarInstance,
    ix_cc_atmsar_port_handle_t arg_hPort
    ix_cc_atmsar_cb_del_port arg_Callback,
    void *arg_pUserContext);
```

Input

<code>arg_sarInstance</code>	Instance of the core component.
<code>arg_hPort</code>	Handle to data structure describing the requested port.
<code>arg_Callback</code>	The pointer to the callback routine. See ix_cc_atmsar_cb_del_port .
<code>arg_pUserContext</code> <code>t</code>	Pointer to user-defined context.

Output/Returns

Returns	<p>Returns a valid <code>ix_error</code>.</p> <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—Operation completed successfully. • <code>IX_CC_ATMSAR_ERROR_INVALID_PARAM</code>—operation failed, invalid pointer in input parameters. • <code>IX_CC_ERROR_MSG_LIBRARY</code>—operation failed, failure on sending a message. • <code>IX_CC_ERROR_UNINIT</code>—operation failed, the core component has not been initialized.
---------	---

28.1.3.5.1 `ix_cc_atmsar_cb_del_port`

The function prototype for the callback function used to notify the calling application of the results of a call to `ix_cc_atmsar_async_port_remove()`.

C Syntax

```
ix_error (*ix_cc_atmsar_cb_del_port)(
    ix_error arg_Result,
    void *arg_pUserContext);
```

Input

<code>arg_pUserContext</code>	Pointer to the calling application-provided context. The handle is valid if result of operation is <code>IX_SUCCESS</code> .
-------------------------------	--

Output/Returns

<code>arg_result</code>	The error code returned by SAR Control core component for the result of the port remove request.
Returns	<p>Returns a valid <code>ix_error</code>.</p> <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—returned if the operation succeeds • A valid <code>ix_error</code>—returned if the operation fails.

28.1.3.6 `ix_cc_atmsar_async_get_vc_stats()`

This function implements the messaging API for getting VC statistics. It is an asynchronous function and the result of operation is processed in callback routine, when the operation is completed. The Core Component Infrastructure calls the callback routine and passes status of operation.

Syntax

```
ix_error ix_cc_atmsar_async_get_vc_stats(
    ix_cc_atmsar_instance_t arg_sarInstance,
    ix_cc_atmsar_vc_handle_t arg_hVc
    ix_cc_atmsar_cb_del_vc arg_Callback,
    void *arg_pUserContext);
```

Inputs

<code>arg_sarInstance</code>	Instance of the core component.
<code>arg_hVc</code>	Handle to the VC.
<code>arg_Callback</code>	The pointer to the callback routine. See ix_cc_atmsar_cb_get_vc_stats .
<code>arg_pUserContext</code> <code>t</code>	Pointer to calling application-defined context.

Output/Returns

Returns	<p>Returns a valid <code>ix_error</code>.</p> <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—Operation completed successfully. • <code>IX_CC_ATMSAR_ERROR_INVALID_PARAM</code>—operation failed, invalid pointer in input parameters. • <code>IX_CC_ERROR_MSG_LIBRARY</code>—operation failed, failure on sending a message. • <code>IX_CC_ERROR_UNINIT</code>—operation failed, the core component has not been initialized.
---------	---

28.1.3.6.1 `ix_cc_atmsar_cb_get_vc_stats`

The function prototype for the callback function used to notify the calling application of the results of a call to `ix_cc_atmsar_async_get_vc_stats()`.

C Syntax

```
ix_error (*ix_cc_atmsar_cb_get_vc_stats)(
    ix_error arg_Result,
    ix_cc_atmsar_vc_stats_t *arg_pLookupResult,
    void *arg_pUserContext);
```

Input

<code>*arg_pLookupResult</code>	The pointer to the statistics buffer area
<code>arg_pUserContext</code>	Pointer to the calling application-provided context. The handle is valid if result of operation is <code>IX_SUCCESS</code> .

Output/Returns

<code>arg_result</code>	The error code returned by SAR Control core component for the result of the VC get stat request.
Returns	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—returned if the operation succeeds • A valid <code>ix_error</code>—returned if the operation fails.

28.1.3.7 `ix_cc_atmsar_async_get_port_stats()`

This function implements the messaging API for getting port statistics. It is an asynchronous function and the result of operation is processed in callback routine, when the operation is completed. The Core Component Infrastructure calls the callback routine and passes status of operation.

Syntax

```
ix_error ix_cc_atmsar_async_get_port_stats(
    ix_cc_atmsar_instance_t arg_sarInstance,
    ix_cc_atmsar_port_handle_t arg_hPort
    ix_cc_atmsar_cb_del_port arg_Callback,
    void *arg_pUserContext);
```

Inputs

<code>arg_sarInstance</code>	Instance of the core component.
<code>arg_hPort</code>	Handle to data structure describing the requested port.
<code>arg_Callback</code>	The pointer to the callback routine. See ix_cc_atmsar_cb_get_port_stats .
<code>arg_pUserContext</code> <code>t</code>	Pointer to user-defined context.

Output/Returns

Returns	<p>Returns a valid <code>ix_error</code>.</p> <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—Operation completed successfully. • <code>IX_CC_ATMSAR_ERROR_INVALID_PARAM</code>—operation failed, invalid pointer in input parameters. • <code>IX_CC_ERROR_MSG_LIBRARY</code>—operation failed, failure on sending a message. • <code>IX_CC_ERROR_UNINIT</code>—operation failed, the core component has not been initialized.
---------	---

28.1.3.7.1 `ix_cc_atmsar_cb_get_port_stats`

The function prototype for the callback function used to notify the calling application of the results of a call to `ix_cc_atmsar_async_get_port_stats()`.

C Syntax

```
ix_error (*ix_cc_atmsar_cb_get_port_stats)(
    ix_error arg_Result,
    ix_cc_atmsar_port_stats_t *arg_pLookupResult,
    void *arg_pUserContext);
```

Input

<code>arg_pLookupResult</code>	Pointer to the statistics buffer area
<code>arg_pUserContext</code>	Pointer to the calling application-provided context. The handle is valid if result of operation is <code>IX_SUCCESS</code> .

Output/Returns

<code>arg_result</code>	The error code returned by SAR Control core component for the result of the port get statistics request.
Returns	<p>Returns a valid <code>ix_error</code>.</p> <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—returned if the operation succeeds • A valid <code>ix_error</code>—returned if the operation fails.

28.1.4 Library API

The Library API provides direct C-style calls to the SAR Control core component. The following library functions are provided. [Table 28-4](#) lists the SAR library APIs.

Table 28-4. SAR Library API

API Function	Description
ix_cc_atmsar_vc_create()	Add VC
ix_cc_atmsar_vc_update()	Update VC
ix_cc_atmsar_vc_remove()	Delete VC
ix_cc_atmsar_port_create()	Add port
ix_cc_atmsar_port_remove()	Delete port
ix_cc_atmsar_get_vc_stats()	Get VC statistics
ix_cc_atmsar_get_port_stats()	Get port statistics

28.1.4.1 [ix_cc_atmsar_vc_create\(\)](#)

This function implements adding VC. It is a synchronous function. On return, the routine passes status of the operation and VC handle assigned for the requested VC. The RXC entry is left with Valid bit not set.

Syntax

```
ix_error ix_cc_atmsar_vc_create(
    ix_cc_atmsar_vc_descr_t *arg_pVcDescr,
    ix_cc_atmsar_vc_handle_t *arg_phVc,
    void *arg_pContext);
```

Input

`arg_pVcDescr` Pointer to data structure describing the requested VC.

`arg_pContext` Pointer to core component context.

Output/Returns

`arg_phVc` Pointer to the handle describing the requested VC.

Returns Returns a valid `ix_error`.

- `IX_SUCCESS`—Operation completed successfully.
- `IX_CC_ERROR_UNINIT`—operation failed, the core component has not been initialized.
- `IX_CC_ATMSAR_ERROR_INVALID_PARAM`—operation failed, invalid pointer in input parameters.
- `IX_NO_RESOURCES`—operation failed, no resources to add the entry.

28.1.4.2 ix_cc_atmsar_vc_update()

This function implements update of previously added VC. It is a synchronous function. On return, the routine passes status of the operation. It checks if RXC VC Allocated bit is set. After the update is completed, the Valid bit in the RXC entry is set. The function is used mainly to specify binding between ATM and forwarding layers.

Syntax

```
ix_error ix_cc_atmsar_vc_update(
    ix_cc_atmsar_vc_handle_t arg_hVc,
    ix_cc_atmsar_update_data_t* arg_pUpdateData,
    void *arg_pContext);
```

Input

arg_hVc	Handle to the VC.
arg_pUpdateData	Pointer to data structure describing the updated VC
arg_pContext	Pointer to core component context.

Output/Returns

Returns	Returns a valid ix_error. <ul style="list-style-type: none"> IX_SUCCESS—Operation completed successfully. IX_CC_ATMSAR_ERROR_BUSY—operation failed, the SAR CRTL does not have resources for the operation performed. IX_CC_ERROR_UNINIT—operation failed, the core component has not been initialized. IX_CC_ATMSAR_ERROR_INVALID_PARAM—operation failed, invalid pointer in input parameters.
---------	---

28.1.4.3 ix_cc_atmsar_vc_remove()

This function implements deleting VC. It is a synchronous function. On return, the routine passes status of the operation.

Syntax

```
ix_error ix_cc_atmsar_vc_remove(
    ix_cc_atmsar_vc_handle_t arg_hVc
    void *arg_pContext);
```

Input

arg_hVc	Handle to the VC.
arg_pContext	Pointer to core component context.

Output/Returns

Returns	<p>Returns a valid <code>ix_error</code>.</p> <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—Operation completed successfully. • <code>IX_CC_ATMSAR_ERROR_INVALID_PARAM</code>—operation failed, invalid pointer in input parameters. • <code>IX_CC_ERROR_UNINIT</code>—operation failed, the core component has not been initialized.
---------	---

28.1.4.4 `ix_cc_atmsar_port_create()`

This function implements adding a port. It is a synchronous function. On return, the routine passes status of the operation and port handle assigned for the requested port.

C Syntax

```
ix_error ix_cc_atmsar_port_create(
    ix_cc_atmsar_port_descr_t *arg_pPortDescr,
    ix_cc_atmsar_port_handle_t *arg_phPort
    void *arg_pContext);
```

Input

<code>arg_pPortDescr</code>	The pointer to data structure describing the requested port.
<code>arg_pContext</code>	Pointer to core component context.

,

Output/Returns

<code>arg_phPort</code>	The pointer to the handle describing the requested port
Returns	<p>Returns a valid <code>ix_error</code>.</p> <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—Operation completed successfully. • <code>IX_NO_RESOURCES</code>—operation failed, no resources to add the entry. • <code>IX_CC_ATMSAR_ERROR_INVALID_PARAM</code>—invalid pointer in input parameters. • <code>IX_CC_ERROR_UNINIT</code>—operation failed, the core component has not been initialized.

28.1.4.5 ix_cc_atmsar_port_remove()

This function implements removing port. It is a synchronous function. On return, the routine passes status of the operation.

C Syntax

```
ix_error ix_cc_atmsar_port_remove(  
    ix_cc_atmsar_port_handle_t arg_hPort  
    void *arg_pContext);
```

Input

arg_hPort	Handle to the data structure describing the requested port.
arg_pContext	Pointer to the core component context.

Output/Returns

Returns	Returns a valid ix_error. <ul style="list-style-type: none">• IX_SUCCESS—Operation completed successfully.• IX_CC_ATMSAR_ERROR_INVALID_PARAM—operation failed, invalid pointer in input parameters.• IX_CC_ERROR_UNINIT—operation failed, the core component has not been initialized.
---------	--

28.1.4.6 ix_cc_atmsar_get_vc_stats()

This function implements getting VC statistics. It is a synchronous function. On return, the routine passes status of the operation and statistics for the requested VC.

C Syntax

```
ix_error ix_cc_atmsar_get_vc_stats(  
    ix_cc_atmsar_vc_handle_t arg_hVc  
    ix_cc_atmsar_vc_stats_t *arg_pLookupResult,  
    void *arg_pContext);
```

Input

arg_hVc	Handle to the VC.
arg_pContext	Pointer to core component context.

Output/Returns

Output	arg_pLookupResult - pointer to the VC stats buffer.
Returns	Returns a valid ix_error. <ul style="list-style-type: none"> • IX_SUCCESS—Operation completed successfully. • IX_CC_ATMSAR_ERROR_INVALID_PARAM—operation failed, invalid pointer in input parameters. • IX_CC_ERROR_UNINIT—operation failed, core component has not been initialized.

28.1.4.7 ix_cc_atmsar_get_port_stats()

This function retrieves port statistics. It is a synchronous function. On return, the routine passes the status of the operation and port statistics for the requested port.

C Syntax

```
ix_error ix_cc_atmsar_get_port_stats(
    ix_cc_atmsar_port_handle_t arg_hPort
    ix_cc_atmsar_port_stats_t *arg_pLookupResult,
    void *arg_pContext);
```

Input

arg_hPort	Handle to the data structure describing the requested port.
arg_pContext	Pointer to core component context.

Output/Returns

arg_pLookupResult	Pointer to the VC statistics buffer.
Returns	Returns a valid ix_error. <ul style="list-style-type: none"> • IX_SUCCESS—Operation completed successfully. • IX_CC_ATMSAR_ERROR_INVALID_PARAM—operation failed, invalid pointer in input parameters. • IX_CC_ERROR_UNINIT—operation failed, the core component has not been initialized.

28.1.5 Plug-in API

Table 28-5 lists the APIs used between SoftSAR Control core component and SoftSAR sub-components (ATM RX, ATM TX and ATM TM4.1) to establish communication with the plug-in core components. All API functions are synchronous. On return, they report operation status in the return code.

Table 28-5. SAR Control Plug-in API

API Function	Description
<code>ix_cc_atmsar_plugin_get_cfg_params()</code>	Gets configuration parameters for the SAR instance
<code>ix_cc_atmsar_plugin_reg_vc_service()</code>	Registers plug-in for servicing VCs
<code>ix_cc_atmsar_plugin_reg_port_service()</code>	Registers plug-in for servicing ports
<code>ix_cc_atmsar_plugin_reg_vc_handle_service()</code>	Registers plug-in for servicing VC handles
<code>ix_cc_atmsar_plugin_reg_port_handle_service()</code>	Registers plug-in for servicing port handles
<code>ix_cc_atmsar_plugin_reg_done()</code>	Confirms that plug-in registering is completed

28.1.5.1 `ix_cc_atmsar_plugin_get_cfg_params()`

The function gets configuration parameters for a SAR instance. The values are read from registry or from configuration header (selected as compile-time option). The configuration parameters describe general behavior and functionality of the SAR instance.

C Syntax

```
ix_error ix_cc_atmsar_plugin_get_cfg_params(
    ix_cc_atmsar_instance_t arg_sarInstance,
    ix_cc_atmsar_plugin_cfg_params_t *arg_pCfgParams)
```

Input

`arg_sarInstance` ID of SAR Instance for that the action is performed.

Output/Returns

`arg_pCfgParams` Pointer to structure where configuration parameters are stored.

Returns a valid `ix_error`.

- `IX_SUCCESS`—returned if the operation succeeds.
- `IX_CC_ERROR_INTERNAL`—operation failed, the calling plug-in works in the wrong thread. Check description of the ATM SAR configuration tables and XML config description.

28.1.5.2 `ix_cc_atmsar_plugin_reg_vc_service()`

The function registers ATM SAR plug-in into ATM SAR core component. The input parameters - callback routines may be NULL, indicating that the plug-in is not interested in servicing that operation.

C Syntax

```
ix_error ix_cc_atmsar_plugin_reg_vc_service(
    ix_uint16 arg_coreComponentId,
    ix_cc_atmsar_instance_t arg_sarInstance,
    ix_cc_atmsar_plugin_vc_create_t arg_pCbVcCreate,
    ix_cc_atmsar_plugin_vc_update_t arg_pCbVcUpdate,
    ix_cc_atmsar_plugin_vc_remove_t arg_pCbVcRemove,
    ix_cc_atmsar_plugin_get_vc_stats_t arg_pCbVcStats);
```

Input

<code>arg_coreComponentId</code>	ID of core component that registers the service.
<code>arg_sarInstance</code>	ID of SAR Instance for that the action is performed.
<code>arg_pCbVcCreate</code>	Pointer to the callback routine servicing “VC CREATE” operation internal in plug-in.
<code>arg_pCbVcUpdate</code>	Pointer to the callback routine servicing “VC UPDATE” operation internal in plug-in.
<code>arg_pCbVcRemove</code>	Pointer to the callback routine servicing “VC REMOVE” operation internal in plug-in.
<code>arg_pCbVcStats</code>	Pointer to the callback routine servicing “GET VC STATS” operation internal in plug-in.

Outputs/Returns

Returns	<p>Returns a valid <code>ix_error</code>.</p> <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—returned if the operation succeeds. <code>IX_CC_ERROR_ENTRY_NOT_FOUND</code>—operation failed, the ATM SAR is not ready for registering the core component for the specific ID. <code>IX_CC_ERROR_DUPLICATE_ENTRY</code>—operation failed, the ATM SAR already registered core component for the specific ID. <code>IX_CC_ERROR_INTERNAL</code>—operation failed, the calling plug-in works in the wrong thread. Check description of ATM SAR configuration tables and XML config description.
---------	---

28.1.5.3 `ix_cc_atmsar_plugin_reg_port_service()`

The function registers ATM SAR plug-in into ATM SAR core component. The input parameters - callback routines may be NULL, indicating that the plug-in is not interested in servicing that operation.

Syntax

```
ix_error ix_cc_atmsar_plugin_reg_port_service(
    ix_uint16 arg_coreComponentId,
    ix_cc_atmsar_instance_t arg_sarInstance,
    ix_cc_atmsar_plugin_port_create_t arg_pCbPortCreate,
    ix_cc_atmsar_plugin_port_remove_t arg_pCbPortRemove,
    ix_cc_atmsar_plugin_get_port_stats_t arg_pCbPortStats);
```

Input

<code>arg_coreComponentId</code>	ID of core component that register service.
<code>arg_sarInstance</code>	ID of SAR Instance for that the action is performed.
<code>arg_pCbPortCreate</code>	Pointer to the callback routine servicing “PORT CREATE” operation internal in plug-in.
<code>arg_pCbPortRemove</code>	Pointer to the callback routine servicing “PORT REMOVE” operation internal in plug-in.
<code>arg_pCbPortStats</code>	Pointer to the callback routine servicing “GET PORT STATS” operation internal in plug-in.

Output/Returns

Returns	<p>Returns a valid <code>ix_error</code>.</p> <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—returned if the operation succeeds. <code>IX_CC_ERROR_ENTRY_NOT_FOUND</code>—The ATM SAR is not ready for registering the core component for the specific ID. <code>IX_CC_ERROR_DUPLICATE_ENTRY</code>—The ATM SAR already registered core component for the specific ID. <code>IX_CC_ERROR_INTERNAL</code>—The calling plug-in works in the wrong thread. Check description of ATM SAR configuration tables and XML config description.
---------	---

28.1.5.4 `ix_cc_atmsar_plugin_reg_vc_handle_service()`

The function registers ATM SAR plug-in into ATM SAR core component. The registered functions are used for allocating and freeing of VC handles.

Syntax:

```
ix_error ix_cc_atmsar_plugin_reg_vc_handle_service(
    ix_cc_atmsar_instance_t arg_sarInstance,
    void* arg_pInstance,
    ix_cc_atmsar_plugin_vc_alloc_t arg_pCbVcAlloc,
    ix_cc_atmsar_plugin_vc_free_t arg_pCbVcFree);
```

:

Input

<code>arg_sarInstance</code>	ID of SAR Instance for that the action is performed.
<code>arg_pInstance</code>	Pointer to structure describing local data instance for the <code>arg_sarInstance</code> .
<code>arg_pCbVcAlloc</code>	Pointer to the callback routine allocating VC handle.
<code>arg_pCbVcFree</code>	Pointer to the callback routine freeing VC handle.

Output/Returns

Returns	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—returned if the operation succeeds. <code>IX_CC_ERROR_INTERNAL</code>—operation failed, the calling plug-in works in the wrong thread. Check description of ATM SAR configuration tables and XML config description.
---------	---

28.1.5.5 `ix_cc_atmsar_plugin_reg_port_handle_service()`

The function registers ATM SAR plug-in into ATM SAR core component. The registered functions are used for allocating and freeing of port handles.

Syntax

```
ix_error ix_cc_atmsar_plugin_reg_port_handle_service(
    ix_cc_atmsar_instance_t arg_sarInstance,
    void* arg_pInstance,
    const ix_cc_atmsar_plugin_port_alloc_t arg_pCbPortAlloc,
    const ix_cc_atmsar_plugin_port_free_t arg_pCbPortFree);
```


Input

<code>arg_sarInstance:</code>	ID of SAR Instance for that the action is performed.
<code>arg_pInstance</code>	Pointer to structure describing local data instance for the <code>arg_sarInstance</code> .
<code>arg_pCbPortAlloc</code>	Pointer to the callback routine allocating port handle.
<code>arg_pCbPortFree</code>	Pointer to the callback routine freeing port handle.

Outputs/Returns

Returns	<p>Returns a valid <code>ix_error</code>.</p> <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—returned if the operation succeeds. <code>IX_CC_ERROR_INTERNAL</code>—operation failed, the calling plug-in works in the wrong thread. Check description of ATM SAR configuration tables and XML config description.
---------	---

28.1.5.6 `ix_cc_atmsar_plugin_reg_done()`

This function notifies ATM SAR that the plug-in already registered all requested services.

C Syntax

```
ix_error ix_cc_atmsar_plugin_reg_done(
    ix_uint16 arg_coreComponentId,
    ix_cc_atmsar_instance_t arg_sarInstance,
    void* arg_pInstance)
```

Input

<code>arg_coreComponentId</code>	ID of the core component that register service.
<code>arg_sarInstance</code>	ID of the SAR instance for which the action is performed.
<code>arg_pInstance</code>	Pointer to structure describing local data instance for the <code>arg_sarInstance</code>

Outputs/Returns

Returns	<p>Returns a valid ix_error.</p> <ul style="list-style-type: none"> IX_SUCCESS—returned if the operation succeeds. IX_CC_ERROR_ENTRY_NOT_FOUND—The ATM SAR is not ready for registering the core component for the specific ID. IX_CC_ERROR_DUPLICATE_ENTRY—The ATM SAR already registered core component for the specific ID. IX_CC_ERROR_INTERNAL—The calling plug-in works in the wrong thread. Check description of ATM SAR configuration tables and XML config description.
---------	--

28.2 ATM RX Core Components

This section describes the high level design for the ATM Receive Core Component. The ATM RX Core Component runs on the ingress side and performs the following functions:

- Performs initialization/configuration of ATM RX microblock and patch symbols.
- Provides an interface to a system application for setting and retrieving configuration and statistical parameters.
- Keeps track of the Virtual Circuit (VC) establish/teardown. This information is used to maintain the RXC table and the hash lookup table for looking up the Receive Context (RXC) for a particular VC.
- Handles exception packets from the ATM RX microblock.
- Supports port (up to 2048) and VCQ#.

For more information on the ATM RX microblock and core component design see [Chapter 6, “ATM AAL5 RX Microblock.”](#) and [Chapter 65, “SoftSAR Core Components”](#) of *Intel® Internet Exchange Architecture Software Building Blocks Developer's Manual*.

28.2.1 Core Component Infrastructure API

Table 28-6 lists the Core Component Infrastructure API provided by ATM RX core components.

Table 28-6. Core Component Infrastructure API

API	Description
<code>ix_cc_atmrx_init()</code>	Initializes the core component
<code>ix_cc_atmrx_fini()</code>	Terminates the core component
<code>ix_cc_atmrx_msg_handler()</code>	Handles messages for interface state and statistics
<code>ix_cc_atmrx_pkt_handler()</code>	Packet handler for processing exception packets

28.2.1.1 `ix_cc_atmrx_init()`

The function initializes the core component. It is called and returned successfully before any other function in the core component can be called. It performs static configuration for the microblock by allocating, and patching required variables and memory block by calling the Resource Manager APIs. When the function returns, ATM RX interface is enabled.

C Syntax

```
ix_error ix_cc_atmrx_init(
    ix_cc_handle arg_CcHandle,
    void **arg_ppContext);
:
```

Input

`arg_CcHandle` handle to the core component.

:

Input/Output

`arg_ppContext`

- As Input, it is a pointer to a generic init context which is used to extract system configuration information.
- As Output, it is location where the pointer to the control block allocated by the core component will be stored. The control block is internal to the core component and will contain Receive Context memory and other variables and internal data structures. This pointer will be use later to be passed into the `ix_cc_atmrx_fini()` function by the Core Component Infrastructure for memory freeing when the core component is being disabled.

.

Output/Returns

Return Value Returns a valid `ix_error`.

- `IX_SUCCESS`—returned if the operation succeeds.
- A valid `ix_error`—returned if the operation fails.

28.2.1.2 `ix_cc_atmrx_fini()`

The function terminates the core component. It is executed when the execution engine running the core component is being shut down. This function frees all allocated memory and resources obtained in `ix_cc_atmrx_init()` function. Before retuning, the function disables the ATM RX interface.

C Syntax

```
ix_error ix_cc_atmrx_fini(
    ix_cc_handle arg_CcHandle,
    void *arg_pContext);
```

Input

`arg_CcHandle` Handle to the core component.

`arg_pContext` Pointer to the control block memory allocated earlier in `ix_cc_atmrx_init()`. The termination routine uses it to de-allocate the control block memory.

Output/Returns

Return Value Returns a valid `ix_error`.

- `IX_SUCCESS`—returned if the operation succeeds.
- A valid `ix_error`—returned if the operation fails.

28.2.1.3 ix_cc_atmrx_msg_handler()

This is the message handler function for ATM RX core component.

C Syntax

```
ix_error ix_cc_atmrx_msg_handler(
    ix_buffer_handle arg_Msg,
    ix_uint32 arg_UserData
    void *arg_pContext);
```

Input

`arg_Msg` Buffer handle embedding information for the invocation of associated library API function.

`arg_UserData` Message type. Table 28-7 shows the defined messages.

`arg_pContext` Handle to the core component created internal context structure.

Table 28-7. Message Type defined in ATM RX Core Components

Message Type	Description
<code>IX_CC_ATM_RX_MSG_GET_INTERFACE_STATE</code>	Obtain the interface state
<code>IX_CC_ATM_RX_MSG_GET_STATISTICS_INFO</code>	Obtain the statistics info.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—returned if the operation succeeds. A valid <code>ix_error</code>—returned if the operation fails.
--------------	---

28.2.1.4 `ix_cc_atmrx_pkt_handler()`

This function handles the exception packets sent “up” by the ATM Rx microblock. These packets are sent to the output defined for this component. Any application/component that needs to capture these packets can “bind” to this output in “bindings.h” file.

C Syntax

```
ix_error ix_cc_atmrx_pkt_handler(
    ix_buffer_handle arg_hBuffer,
    ix_uint32 arg_ExceptionCode,
    void *arg_pContext);
```

Input

<code>arg_hBuffer</code>	Handle for the packet buffer
<code>arg_ExceptionCode</code>	The following values are valid: <ul style="list-style-type: none"> 0— for an unsupported AALx type 1—for a packet with CRC error
<code>arg_pContext</code>	Pointer to a context (may be a calling application-defined structure) that is passed to the core component when a packet arrives. This context is defined by the core component and passed to the framework through the <code>ix_cci_cc_add_packet_handler</code> function called in the init function of the core component's execution engine.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—returned if the operation succeeds. A valid <code>ix_error</code>—returned if the operation fails.
--------------	---

28.3 ATM TX Core Components

The ATM TX Core Component performs the following functions:

- Initializes and configures the ATM TX microblock through patching symbols
- Provides interfaces for setting and retrieving the ATM TX interface state and statistical parameters
- Initializes and configures the ATM framer device

For SoftSAR Core Components design [Chapter 65, “SoftSAR Core Components”](#) of the *Intel® Internet Exchange Architecture Software Building Blocks Developer’s Manual*.

28.3.1 Core Component Infrastructure API

Table 28-8 lists the Core Component Infrastructure API provided by the ATM TX core component.

Table 28-8. ATM TX Core Component Infrastructure API

API	Description
ix_cc_atmtx_init()	Initializes the core component
ix_cc_atmtx_fini()	Terminates the core component

28.3.1.1 [ix_cc_atmtx_init\(\)](#)

The function initializes the core component. It is called and returned successfully before calling any other function in the core component. Based on channel mode specified in the static data, the function performs the following:

- Performs Static configuration for either ATM TX microblock or POS TX microblock by allocating and patching required variables and memory block into the microblock through calling Resource Manager's APIs, [ix_rm_ueng_patch_symbols\(\)](#)
- Registers message handler for the defined message inputs
- Initializes and configures the ATM/POS framer device by calling the framer device driver based on the static configuration data obtained

C Syntax

```
ix_error ix_cc_atmtx_init(
    ix_cc_handle arg_CcHandle,
    void **arg_ppContext);
```

Input

`arg_CcHandle` Handle to the core component.

Input/Output

<code>arg_ppContext</code>	<ul style="list-style-type: none"> • As Input, it is a pointer to a generic init context which is used to extract system configuration information. • As Output, it is a location where the pointer to the control block allocated by the core component is stored. The control block is internal to the core component and contains Transmit Context memory, other variables and internal data structures. This pointer is used later to be passed into the <code>ix_cc_atmtx_fini()</code> function by the Core Component Infrastructure to free memory when the core component is being terminated.
----------------------------	--

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—returned if the operation succeeds. • A valid <code>ix_error</code>—returned if the operation fails.
--------------	---

28.3.1.2 `ix_cc_atmtx_fini()`

The function terminates the core component and is executed when the execution engine running the core component is being shut down. This function frees all allocated memory and resources allocated by `ix_cc_atmtx_init()` function as well as other functions of the core component.

C Syntax

```
ix_error ix_cc_atmtx_fini(
    ix_cc_handle arg_CcHandle,
    void *arg_pContext);
```

Input

<code>arg_cc_handle</code>	Handle to the core component.
<code>arg_pContext</code>	Pointer to the control block memory allocated earlier in <code>ix_cc_atmtx_init()</code> . The termination routine uses it to de-allocate the control block memory.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—returned if the operation succeeds. • A valid <code>ix_error</code>—returned if the operation fails.
--------------	---

28.4 TM4.1 Core Components

This section describes the high level design of the TM4.1 core component. The following functionalities are supported:

- Provides an API interface to add and remove ports and VCs. VCs description and ports description are stored in tables shared between the core component and the microblocks.
- Defines algorithms for setting TM4.1 microblocks data that provide even cell transmit for ports and for VC with real-time related conformance parameters.

For information on the TM4.1 microblock, see [Chapter 23, “TM4.1 Shaper and Scheduler Microblock”](#) of the *Intel® Internet Exchange Architecture Software Building Blocks Developer’s Manual*.

28.4.1 Core Component Infrastructure API

Table 28-9 lists the TM4.1 core component supported the Core Component Infrastructure APIs

Table 28-9. TM4.1 Core Component supported Core Component Infrastructure API

API function	Description
<code>ix_cc_atmtm41_init()</code>	Initializes the core component
<code>ix_cc_atmtm41_fini()</code>	Terminates the core component
<code>ix_cc_atmtm41_msg_handler()</code>	Handles messages for processing rule add/remove requests
<code>ix_cc_atmtm41_pkt_handler()</code>	Handles packets for processing exception packets

28.4.1.1 `ix_cc_atmtm41_init()`

The function initializes the core component. It is called and returned successfully before calling any other function in the core component. The API performs the following functions:

- Allocates DRAM/SRAM memory for the hash table and initializes it with empty entries (all zeros)
- Patches TM4.1 microcode with imported variables. The function gets the variable values from the system repository properties
- Registers a packet handler `ix_cc_atmtm41_pkt_handler()` for inputs:
 - `IX_CC_ATMSAR_TM41_EXCEPTION_PKT_INPUT` (single source mode)
 - `IX_CC_ATMSAR_TM41_LOCAL_PKT_INPUT` (single source mode)
- Registers a message handler `ix_cc_atmtm41_msg_handler()` for inputs:
 - `IX_CC_ATMSAR_TM41_MSG_INPUT` (multiple sources mode)
- Allocates a control block, and returns a pointer to this block in `arg_ppContext`

C Syntax

```
ix_error ix_cc_atmtm41_init(
    ix_cc_handle arg_CcHandle,
    void **arg_ppContext);
:
```

Input

`arg_CcHandle` Handle to the core component.

Input/Output

`arg_ppContext`

- As Input, it is a pointer to a generic init context which is used to extract system configuration information.
- As output, it is a location of a pointer where the control block allocated by the core component is stored. The control block is internal to the core component and contains variables and internal data structures. This pointer is used later, and passed on to the `ix_cc_atmtm41_fini()` function by the Core Component Infrastructure to free memory when the core component is being terminated.

Output/Returns

Return Value

Returns a valid `ix_error`.

- `IX_SUCCESS`—returned if the operation succeeds.
- `IX_CC_ERROR_FAILED_MEMORY_ALLOCATION`—operation failed, DRAM/ SRAM memory allocation failure
- `IX_CC_ERROR_CCI`—operation failed, failure from the Core Component Infrastructure

28.4.1.2 `ix_cc_atmtm41_fini()`

The function terminates the core component. It is executed when the execution engine running the core component is being shut down. This function frees all allocated memory and resources obtained in `ix_cc_atmtm41_init()` function

C Syntax

```
ix_error ix_cc_atmtm41_fini(
    ix_cc_handle arg_CcHandle,
    void *arg_pContext);
```

Input

<code>arg_CcHandle</code>	Handle to the core component.
<code>arg_pContext</code>	Pointer to the control block memory allocated earlier in <code>ix_cc_atmtm41_init()</code> . The termination routine uses it to de-allocate the control block memory.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—returned if the operation succeeds. A valid <code>ix_error</code>—returned if the operation fails.
--------------	---

28.4.1.3 `ix_cc_atmtm41_msg_handler()`

The function is the message handler for TM4.1 core component.

C Syntax

```
ix_error ix_cc_atmtm41_msg_handler(
    ix_buffer_handle arg_Msg,
    ix_uint32 arg_UserData
    void *arg_pContext);
```

Input

<code>arg_Msg</code>	Buffer handle embedding information for the calling the associated library APIs.
<code>arg_UserData</code>	Indicates the message type.
<code>arg_pContext</code>	Handle to the core component created internal context structure.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> .
	<ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—returned if the operation succeeds. • A valid <code>ix_error</code>—returned if the operation fails.

28.4.1.4 `ix_cc_atmtm41_pkt_handler()`

This function handles the exception packets sent by the TM4.1 microblock. These packets are sent to the output defined for this component. Any application/component that needs to capture these packets can “bind” to this output in `bindings.h` file.

C Syntax

```
ix_error ix_cc_atmtm41_pkt_handler(
    ix_buffer_handle arg_hBuffer,
    ix_uint32 arg_ExceptionCode, void *arg_pContext);
```

Input

<code>arg_hBuffer</code>	Handle for the packet buffer
<code>arg_ExceptionCode</code>	A value “0” indicates the code associated with the packet
<code>arg_pContext</code>	Pointer to a context (may be a calling application-defined structure) that is passed to the core component when a packet arrives. This context is defined by the core component and passed to the framework through the <code>ix_cci_cc_add_packet_handler</code> function called in the <code>init</code> function of the core component's execution engine.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> .
	<ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—returned if the operation succeeds. • A valid <code>ix_error</code>—returned if the operation fails.

MPLS Core Component

The following chapter is included in this section:

- [Chapter 29, “MPLS Forwarder”](#)

The MPLS Forwarder core component receives packets and messages from the MPLS Forwarder microblocks (ILM Forwarder and FTN Forwarder).

The MPLS Forwarder core component performs the following functions on behalf of the ILM Forwarder and FTN Forwarder microblocks:

- Initializes and maintains the data structures for the ILM Forwarder and FTN Forwarder microblocks
- Configures the ILM Forwarder and FTN Forwarder microblocks (static configuration)
- Provides message and packet handlers to receive messages from the control plane and packets from the ILM Forwarder and FTN Forwarder microblocks
- Handles fragmentation of MPLS packets
- Implements "slow path" forwarding for fragmented MPLS packets
- Handles packets with special MPLS label values

The MPLS Forwarder core component receives packets and messages from the MPLS Forwarder microblocks (ILM Forwarder and FTN Forwarder).

The MPLS Forwarder core component performs the following functions on behalf of the ILM Forwarder and FTN Forwarder microblocks:

- Initializes and maintains the data structures for the ILM Forwarder and FTN Forwarder microblocks
- Configures the ILM Forwarder and FTN Forwarder microblocks (static configuration)
- Provides message and packet handlers to receive messages from the control plane and packets from the ILM Forwarder and FTN Forwarder microblocks
- Handles fragmentation of MPLS packets
- Implements "slow path" forwarding for fragmented MPLS packets
- Handles packets with special MPLS label values

For complete details see [Chapter 66, “MPLS Forwarder Core Component”](#) of the *Intel® Internet Exchange Architecture Software Building Blocks Developer’s Manual*.

29.1 Data Structures and Types

[Figure 29-1](#) shows the main tables maintained by the MPLS Forwarder Core Component, together with variables indexing their entries. An overview of the tables is provided below; full details on the tables are given later in this chapter.

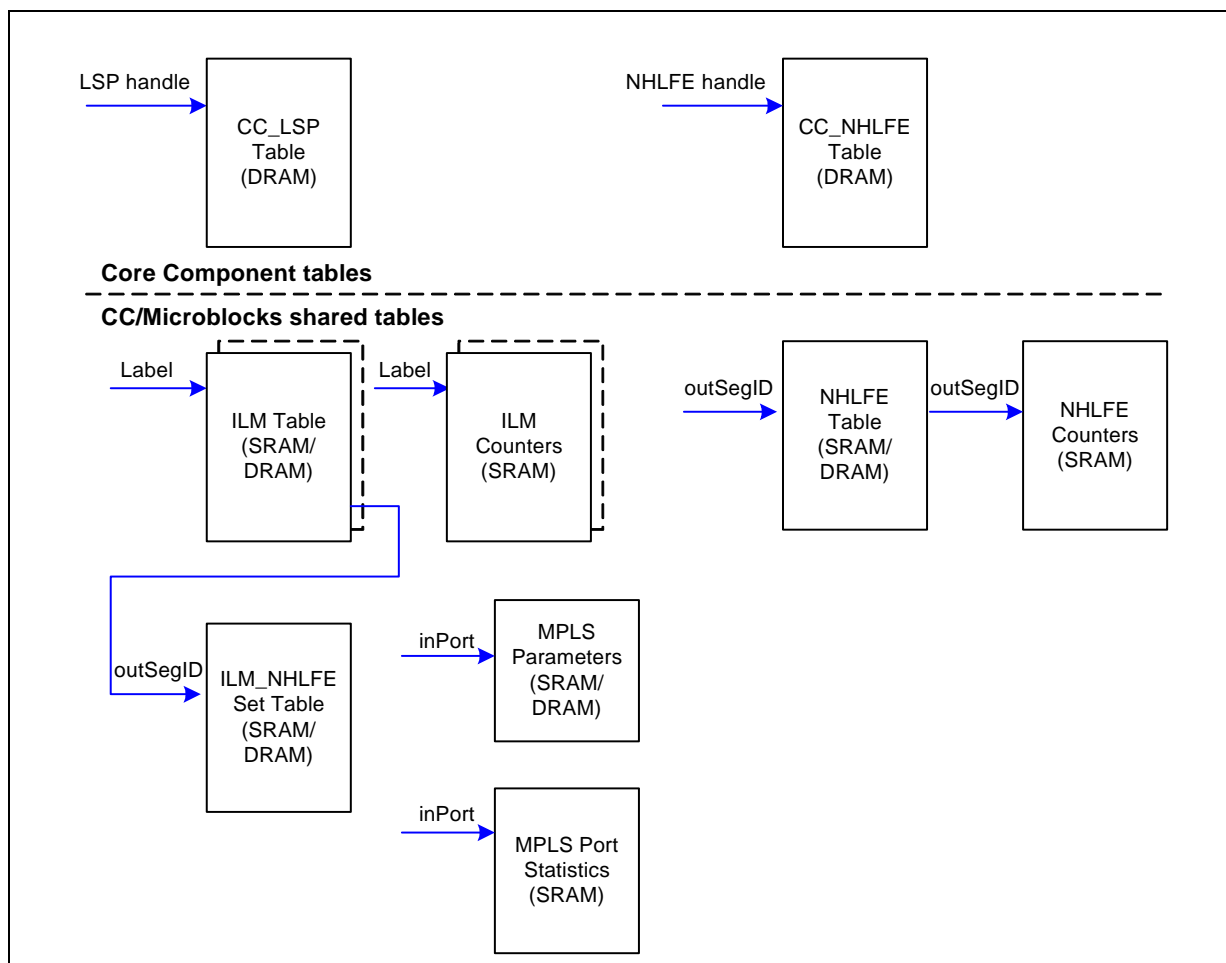
- CC_LSP Table is used by the core component for storing LSP descriptions (FEC to NHLFE or ILM to NHLFE mappings). It is indexed by LSP handle values.
- CC_NHLFE Table is used by the core component for tracking individual NHLFE usage by different LSPs and accessing the NHLFE data. It is indexed by NHLFE handle values.
- ILM Table is maintained by the MPLS core component and used by the ILM Forwarder microblock for packet forwarding. It is indexed by the top label value of the received MPLS packet. The ILM table can contain regular entries (used directly for forwarding) and set entries (pointing to regular entries stored in the ILM_NHLFE Set table). The layout of this table is defined in [Section 36.5.2, “ILM Table” on page 624](#) of the *Intel® Internet Exchange Architecture (IXA) Software Building Blocks Developer’s Manual*.

- ILM Counters are per-ILM entry structures accommodating various LSP statistics. They are compiled conditionally.

Note: There are individual per-input port ILM tables and ILM Counters tables in PER_INTERFACE_LABEL_SPACE mode.

- ILM_NHLFE Set Table stores forwarding entries belonging to ILM sets. It is indexed by the outSegID values contained in ILM table set entries.

Figure 29-1. MPLS Core Component Data Structures



- NHLFE Table is maintained by the MPLS core component and used by the FTN Forwarder microblock for packet forwarding. It is indexed by the outSegID (also called the FEC ID) value obtained as the result of the LPM lookup performed on the received packet's IP destination address by the IPv4 Forwarder microblock. There is a 1:1 correspondence of OutSegID values to NHLFE handles. The layout of this table is defined in [Section 35.5.2, “NHLFE Table” on page 606](#) of the *Intel® Internet Exchange Architecture (IXA) Software Building Blocks Developer's Manual*.
- NHLFE Counters are per-NHLFE entry structures accumulating various NHLFE statistics. They are compiled conditionally.
- MPLS Parameters comprise the flag determining per-platform or per-interface label space for the device, label ranges for input ports, and predefined EXP to PHB and PHB to EXP mappings used by DiffServ over MPLS application. These parameters are read-only; they are set by the core component at system initialization time and copied to local memory during microcode initialization.
- MPLS Port Statistics are per-input port counters accumulating the number of received MPLS packets and bytes on a given local port. It is assumed that output port MPLS statistics will be collected by the L2 Encapsulation microblock on the output blade.

These structures are shared between the MPLS core component, the ILM Forwarder and the MPLS Marker microblocks. Additionally, the MPLS Forwarder core component operates on several specific data types. Table 29-1 lists the data structures and types described in the following sections.

Table 29-1. MPLS Forwarder Core Component Data Types and Structures

Data Types and Structures	Description
ix_cc_mpls_fec	Specifies an MPLS Forwarding Equivalence Class
ix_cc_mpls_label	Specifies the control-plane level view of an MPLS label
ix_cc_mpls_ilm	Specifies the ILM key
ix_cc_mpls_params	Specifies the control-plane level view of MPLS parameters
ix_cc_mpls_nhlfe	Specifies the control-plane level view of an NHLFE table entry
ix_cc_mpls_nhlfe_handle	Defines the layout of an NHLFE handle
ix_cc_mpls_cc_nhlfe	Defines the layout of the CC_NHLFE table entry
ix_cc_mpls_nhlfe_stats	Specifies the control-plane view of per-NHLFE table entry counters
ix_cc_mpls_cc_lsp	Defines the layout of a CC_LSP table entry
ix_cc_mpls_lsp_param	Specifies the control-plane level view of LSP parameters
ix_cc_mpls_lsp	Specifies the control-plane level view of an LSP
ix_cc_mpls_lsp_stats	Specifies the control-plane view of a per-LSP statistics
ix_cc_mpls_port_stats	Specifies the control-plane view of an MPLS Ports Statistics Table entry

29.1.1 ix_cc_mpls_fec

This type specifies an MPLS Forwarding Equivalence Class.

C Syntax

```
typedef struct ix_s_cc_mpls_fec
{
    #define IX_CC_MPLS_FEC_TYPE_IPV4          1

    /*! FEC type: IPv4, IPv6, 6-tuple, etc. */
    ix_uint32 type;
    union
    {
        struct ix_s_ip4fec
        {
            ix_uint32      ipAddress;
            ix_uint32      netMask;
            ix_cc_rtmv4_nhid nexthopId;
        } ip4fec;

        /* other FECs to be defined */
    } fec;
} ix_cc_mpls_fec;
```

Data Members

type	The type of data stored in union FEC - currently the only supported value is IX_CC_MPLS_FEC_TYPE_IPV4.
ipAddress	The IPv4 address value for the FEC.
netMask	The network mask value for the FEC.
nexthopId	The next hop ID value identifying the IPv4 next hop database entry for this FEC.

29.1.2 ix_cc_mpls_label

This type specifies the control-plane level view of an MPLS label.

C Syntax

```
typedef struct ix_s_cc_mpls_label
{
    ix_uint32    label        :20,
                reserved      :12;
} ix_cc_mpls_label;
```

29.1.3 ix_cc_mpls_ilm

This type specifies the ILM table key (MPLS label and incoming port) and operation to be performed on the incoming label.

C Syntax

```
typedef struct ix_s_cc_mpls_ilm
{
    ix_cc_mpls_label label;
    ix_uint16    in_port;    /* 0 for per-platform label space */

    /* LSP operation field definitions */
#define IX_MPLS_LABELOP_SWAP      0
#define IX_MPLS_LABELOP_SWAP_PUSH 1
#define IX_MPLS_LABELOP_POP      2
#define IX_MPLS_LABELOP_POP_FORWARD 3
#define IX_MPLS_LABELOP_PUSH     4

    ix_uint8    operation;    /* Label operation */
} ix_cc_mpls_ilm;
```

29.1.4 ix_cc_mpls_params

This type specifies the control-plane level view of MPLS parameters.

C Syntax

```
/* ix_cc_mpls_lm_exp2phb Data Type */
/**
 * Data structure of EXP to PHB mapping.
 * Used in ix_cc_mpls_params
 */
typedef struct ix_s_cc_mpls_lm_exp2phb
{
    ix_uint32          color_id      :2,
                      class_id      :6,
                      reserved      :18,
                      phbid         :6;
} ix_cc_mpls_lm_exp2phb;

/*
 * ix_cc_mpls_lm_phb2exp
 */
/**
 * Data structure of EXP to PHB mapping.
 * Used in ix_cc_mpls_params
 */
typedef struct ix_s_cc_mpls_lm_phb2exp
{
    ix_uint32          reserved3     :5,
                      exp_3         :3,
                      reserved2     :5,
                      exp_2         :3,
                      reserved1     :5,
                      exp_1         :3,
                      reserved0     :5,
                      exp_0         :3;
} ix_cc_mpls_lm_phb2exp;

typedef struct ix_s_cc_mpls_params
{
    ix_uint32 labelSpaceMode; /* per platform (0),
                             * per-interface(1) */

    struct
    {
        ix_cc_mpls_label min_label;
        ix_cc_mpls_label max_label;
    } labelRange[MPLS_MAX_LOCAL_PORTS] ;
    ix_cc_mpls_lm_exp2phb preconfExp2Phb;
    ix_cc_mpls_lm_phb2exp preconfPhb2Exp2[MPLS_MAX_PHB_NUM];
} ix_cc_mpls_params;
```

During initialization in per-interface label space mode the MPLS Core Component creates a corresponding MPLS parameters table in SRAM, containing the minimum and maximum label values for each port together with offsets to the beginnings of ILM and statistics tables from a common base. This table looks as follows:

```

port0:    LW0    MIN_LABEL0
           LW1    MAX_LABEL0
           LW2    ilm_offset0
           LW3    ilm_stats_offset0    (conditionally compiled)

port1:    LW4    MIN_LABEL1
           LW5    MAX_LABEL1
           LW6    ilm_offset1
           LW7    ilm_stats_offset1    (conditionally compiled)

```

And so on...

For efficiency reasons the ILM Forwarder microblock can write this table to local memory.

29.1.5 ix_cc_mpls_nhlfe

This type specifies the control-plane level view of an NHLFE table entry. It is used by the MPLS CC API.

C Syntax

```

/* NHLFE data type */
typedef struct ix_s_cc_mpls_nhlfe
{

    ix_uint8  pushCount;          /* number of labels to push, 1 to 4 */
    ix_uint8  l3Protocol;         /* type of the MPLS payload
    *   MPLS_L3_UNKNOWN
    *   MPLS_L3_IP_V4
    *   MPLS_L3_IP_V6
    */
    ix_uint32 nextHopId;          /* index to the L2 table on the egress blade
    */

    ix_uint16 bladeId;            /* outgoing blade ID */
    ix_uint16 outPortId;          /* outgoing port ID */
    ix_uint8  outPortType;        /* outgoing port type - Eth, POS, ATM */

    ix_cc_mpls_label topLabel;    /* label pushed on the top of stack */
    ix_cc_mpls_label label1;      /* label pushed below the top of stack */
    ix_cc_mpls_label label2;      /* label pushed below label1 */
    ix_cc_mpls_label label3;      /* label pushed below label2 */

    ix_uint32 mplsPolicy;         /* criteria for choosing set element */

} ix_cc_mpls_nhlfe;

```

29.1.6 ix_cc_mpls_nhlfe_handle

This type defines the layout of an NHLFE handle which is used to index the CC_NHLFE table.

C Syntax

```
typedef struct ix_s_cc_mpls_nhlfe_handle {
    ix_uint32 bladeID : 8,
               outsegID: 24; /* index to NHLFE table */
} ix_cc_mpls_nhlfe_handle;
```

29.1.7 ix_cc_mpls_cc_nhlfe

This type specifies the layout of a CC_NHLFE table entry.

C Syntax

```
/* CC_NHLFE table entry */
typedef struct ix_s_cc_mpls_cc_nhlfe
{
    ix_uint16 flags; /* bits 0:1
                     * - 00 entry is vacant
                     * - 01 entry is taken by stand-alone NHLFE ,
                     * - 10 entry points at NHLFE SET
                     * - 11 entry is used in NHLFE SET
                     */
    ix_uint16 refCount; /* number of LSPs referencing this NHLFE */
    void*ilmList; /* list of referencing ILM entries */
} ix_cc_mpls_cc_nhlfe;
```

29.1.8 ix_cc_mpls_nhlfe_stats

This type specifies the control-plane view of per-NHLFE table entry counters.

C Syntax

```
typedef struct ix_s_cc_mpls_nhlfe_stats
{
    ix_uint64 mplsNhlfeOctets; /* octets received */
    ix_uint32 mplsNhlfePkts; /* packets received */

    /* octets not sent because of TTL expired or required
     * fragmentation but had the DF bit set in the IP header
     */
    ix_uint64 mplsNhlfeErrorOctets;
    ix_uint32 mplsNhlfeErrorPkts; /* packets as above */

    /* deltas between received packets and sent fragmented packets */
    ix_uint32 mplsNhlfeFragmOctetsDelta;
    ix_uint32 mplsNhlfeFragmPktsDelta;
```

```
} ix_cc_mpls_nhlfe_stats;
```

29.1.9 ix_cc_mpls_cc_lsp

This type specifies the layout of a CC_LSP table entry.

C Syntax

```
/* CC_LSP table entry */
typedef struct ix_s_cc_mpls_cc_lsp
{
    ix_uint16          flags;      /* bit 0 - 0 entry is free
                                   *          1 entry holds valid
                                   *          LSP entry
                                   */

    ix_uint16          type;      /* LSP_FEC(1), LSP_ILM(2) */
    ix_uint32          inHandle; /* MPLS label or FEC ID */
    ix_cc_mpls_nhlfe_handle outHandle; /* NHLFE handle */
} ix_cc_mpls_cc_lsp;
```

29.1.10 ix_cc_mpls_lsp_param

This type specifies the control-plane view of LSP parameters.

C Syntax

```
/* LSP flags field definitions */
#define MPLS_FLAG_TUNNEL_PIPE          0x00
#define MPLS_FLAG_TUNNEL_UNIFORM      0x10
#define MPLS_FLAG_TUNNEL_SHORT_PIPE  0x20
#define MPLS_FLAG_TUNNEL_SHORT_PIPE_WITH_PHP 0x30

#define MPLS_FLAG_LSP                  0x00 /* non-Diffserv */
#define MPLS_FLAG_E_LSP                0x08
#define MPLS_FLAG_L_LSP                0x0C

#define MPLS_FLAG_TC_ENABLED           0x02 /* traffic conditioning
*/
#define MPLS_FLAG_PRECONF_EXP_MARK    0x01 /* perform EXP marking */

typedef struct ix_s_cc_mpls_lsp_param
{
    ix_uint32 flags; /* LSP flags */
    ix_uint16 maxLabPktSize; /* LSP MTU */
    ix_uint8 ttl; /* ttl decrement value */
    ix_uint32 flowID; /* flow ID for DiffServ */
    ix_uint16 exp2phbIdx; /* index into the EXP2PHB table */
    ix_uint16 phb2expIdx; /* index into the PHB2EXP table */
} ix_cc_mpls_lsp_param;
```

29.1.11 ix_cc_mpls_lsp

This type specifies the control-plane level view of an LSP.

C Syntax

```
typedef struct ix_s_cc_mpls_lsp
{
    #define IX_CC_MPLS_LSP_FEC_TYPE          1
    #define IX_CC_MPLS_LSP_ILM_TYPE          2

    ix_uint8 lspType;          /* LSP_FEC(1), LSP_ILM(2) */

    union
    {
        ix_cc_mpls_fec fec;
        ix_cc_mpls_ilm ilm;
    } inSeg;

    #define IX_CC_MPLS_LSP_NHLFE_SET_HANDLE_TYPE 1
    #define IX_CC_MPLS_LSP_NHLFE_HANDLE_TYPE 2
    #define IX_CC_MPLS_LSP_NHLFE_DATA_TYPE 3

    ix_uint8          nhlfeInfoType; /* NHLFE_SET_HANDLE_TYPE(1),
                                        * NHLFE_HANDLE_TYPE(2),
                                        * NHLFE_DATA_TYPE(3)
                                        */

    ix_cc_mpls_nhlfe_handle nhlfeSetHandle;

    ix_uint16          nhlfe_num;      /* number of nhlfes in the table
    */
    union
    {
        ix_cc_mpls_nhlfe_handle nhlfeHandle[MPLS_MAX_NHLFE_IN_SET];
        ix_cc_mpls_nhlfe          *nhlfeData[MPLS_MAX_NHLFE_IN_SET];
    } outSeg;

    ix_cc_mpls_lsp_param      lspParams;
} ix_cc_mpls_lsp;
```

29.1.12 ix_cc_mpls_lsp_stats

This type specifies the control-plane view of a per-LSP statistics.

C Syntax

```
typedef struct ix_s_cc_mpls_lsp_stats
{
    ix_uint64 mplsRxPkts;
    ix_uint64 mplsRxOctets;
```

```

ix_uint64 mplsErrorPkts;
ix_uint64 mplsErrorOctets;
ix_uint64 mplsTxPkts;
ix_uint64 mplsTxOctets;
} ix_cc_mpls_lsp_stats;

```

29.1.13 ix_cc_mpls_port_stats

This type specifies the control-plane view of an MPLS Ports Statistics Table entry.

C Syntax

```

typedef structure ix_s_cc_mpls_port_stats
{
    ix_uint32 mplsInIfcRxPkts;
    ix_uint64 mplsInIfcRxOctets;
    ix_uint32 mplsInIfcFailedLookupPkts;
    ix_uint32 mplsInIfcErrorPkts;
} ix_cc_mpls_port_stats;

```

29.2 Core Component Infrastructure API

Table 29-2 lists the MPLS core component infrastructure API.

Table 29-2. MPLS Core Component Infrastructure API

API	Description
<code>ix_cc_mpls_init()</code>	Initialize the core component
<code>ix_cc_mpls_fini()</code>	Terminate the core component
<code>ix_cc_mpls_msg_handler()</code>	Message handler for forwarding tables maintenance, MPLS ports management, and statistics
<code>ix_cc_mpls_microblock_pkt_handler()</code>	Packet handler for processing exception packets from MPLS microblocks

29.2.1 ix_cc_mpls_init()

The function initializes the MPLS core component and is called and returned successfully before any other function in the core component can be called. This function should be called only once for initialization. The calling application needs to initialize the IXA software framework and the Core Components Infrastructure before calling this function.

The function performs the following:

- Allocates memory for symbols to be patched
- Creates 64 bit counters
- Registers packet and message handlers
- Allocates and initializes internal data structures

- Initializes and configures the ILM and FTN Forwarder microblocks

C Syntax

```
ix_error ix_cc_mpls_init (
    ix_cc_handle arg_hCcHandle,
    void **arg_ppContext);
```

Input

arg_hCcHandle Handle to the core component, created by the core component infrastructure; this will be used later to get other services from the core component infrastructure.

:

Output/Returns

arg_ppContext The handle of the buffer free list and dynamic property data, given by the system application during initialization.

Upon successful completion, it represents the location where the pointer to the control block allocated by MPLS forwarder core component is stored. The control block is internal to the MPLS core component and it contains information about MPLS internal data structures, allocated memory, and other relevant information. Later this pointer is passed to the [ix_cc_mpls_fini\(\)](#) function for freeing memory and other house-keeping operations when the MPLS core component is terminated.

Return Value Returns a valid `ix_error`.

- `IX_SUCCESS`—the operation succeeded.
- `IX_CC_ERROR_NULL`—operation failed due to an invalid pointer
- `IX_CC_ERROR_OOM`—operation failed due to memory allocation failure
- `IX_CC_ERROR_FAILED_PATCHING`—operation failed due to patching failures
- `IX_CC_ERROR_64BIT_COUNTER`—operation failed due to a 64-bit counter creation failure
- `IX_CC_ERROR_CCI`—operation failed due to failure from Core Component Infrastructure

29.2.2 `ix_cc_mpls_fini()`

The function terminates the MPLS core component. It is executed when the execution engine running the core component is being shut down. This function frees all allocated memory and resources allocated by the `ix_cc_mpls_init()` function and other functions of the core component.

The calling application must stop the microengines before calling this function. If the calling application requests MPLS services after calling this function, then the behavior is undefined.

C Syntax

```
ix_error ix_cc_mpls_fini (
    ix_cc_handle arg_hCcHandle,
    void *arg_pContext);
```

Input

<code>arg_hCcHandle</code>	Handle to the core component.
<code>arg_pContext</code>	Pointer to the control block memory allocated earlier by the call to ix_cc_mpls_init() . The termination routine uses it to de-allocate the control block memory.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. <code>IX_CC_ERROR_NULL</code>—operation failed due to an invalid pointer <code>IX_CC_ERROR_OOM</code>—operation failed due to memory free failure <code>IX_CC_ERROR_OOM_64BIT_COUNTER</code>—operation failed due to a 64-bit counter deletion failure <code>IX_CC_ERROR_CCI</code>—operation failed due to failure from Core Component Infrastructure
--------------	---

29.2.3 ix_cc_mpls_msg_handler()

This is the message handler function for the MPLS core component. The MPLS core component receives messages from the MPLS microblocks through this message handler function, and internally calls the appropriate library functions to process the message.

C Syntax

```
ix_error ix_cc_mpls_msg_handler (
    ix_buffer_handle arg_hDataToken,
    ix_uint32 arg_UserData
    void* arg_pContext);
```

Input

<code>arg_hDataToken</code>	buffer handle embedding information for the message passed in <code>arg_UserData</code> .
<code>arg_UserData</code>	message type. The supported messages are listed in the Table 29-3 .
<code>arg_pContext</code>	pointer to the MPLS Forwarder core component context that is passed to the core component when a message arrives. This context was defined by the core component and passed to core components infrastructure through the ix_cc_mpls_init() function.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. <code>IX_CC_ERROR_UNDEFINED_MSG</code>—operation failed due to unsupported message <code>IX_CC_ERROR_NULL</code>—operation failed due to an invalid pointer <code>IX_CC_MPLS_ERROR_INVALID_BUF_HANDLE</code>—operation failed due to unusable buffer handle
--------------	---

Table 29-3 lists the supported messages.

Table 29-3. Messages Supported by MPLS Forwarder Core Components

Messages	Description
<code>IX_CC_MPLS_MSG_LSP_CREATE</code>	This message creates an LSP.
<code>IX_CC_MPLS_MSG_LSP_DELETE</code>	This message deletes an LSP.
<code>IX_CC_MPLS_MSG_LSP_MODIFY</code>	This message modifies an LSP.
<code>IX_CC_MPLS_MSG_LSP_QUERY</code>	This message reads an LSP description.
<code>IX_CC_MPLS_MSG_LSP_STATS_QUERY</code>	This message reads LSP statistics.
<code>IX_CC_MPLS_MSG_LSP_PURGE</code>	This message deletes all LSPs.
<code>IX_CC_MPLS_MSG_NHLFE_CREATE</code>	This message creates an NHLFE table entry.
<code>IX_CC_MPLS_MSG_NHLFE_DELETE</code>	This message deletes an NHLFE table entry.
<code>IX_CC_MPLS_MSG_NHLFE_QUERY</code>	This message reads an NHLFE table entry.
<code>IX_CC_MPLS_MSG_NHLFE_STATS_QUERY</code>	This message reads NHLFE table entry statistics.
<code>IX_CC_MPLS_MSG_NHLFE_PURGE</code>	This message deletes all NHLFE table entries.
<code>IX_CC_MPLS_MSG_NHLFE_SET_CREATE</code>	This message creates an NHLFE set entry.
<code>IX_CC_MPLS_MSG_NHLFE_SET_DELETE</code>	This message deletes an NHLFE set entry.
<code>IX_CC_MPLS_MSG_NHLFE_SET_MODIFY</code>	This message modifies an NHLFE set entry.
<code>IX_CC_MPLS_MSG_NHLFE_SET_QUERY</code>	This message reads an NHLFE set entry.
<code>IX_CC_MPLS_MSG_PARAM_QUERY</code>	This message reads the MPLS parameters table.
<code>IX_CC_MPLS_MSG_PORT_STATS_QUERY</code>	This message reads MPLS port statistics.

29.2.4 `ix_cc_mpls_microblock_pkt_handler()`

This is the registered function to receive exception packets from the ILM Forwarder and FTN Forwarder microblocks. This function internally calls different functions and performs various operations based on the exception code for a given packet. According to the results of exception processing, the packet handler should send the generated packets through one of the MPLS core component outputs.

The exception code can be one of the following:

Exception Code	Description
LABELED_PACKET_TOO_BIG	<p>After (possibly initial) labeling, the resulting MPLS packet's length exceeds the "Effective Maximum Frame Payload Size for Labeled Packets" (maxLabPktSize) of the output port.</p> <p>For a LABELED_PACKET_TOO_BIG exception, the MPLS core component must implement algorithms specified in sections 3.4 of [RFC3032] for IPv4 datagrams. This implies the necessity of implementing IP fragmentation routines and a full "slow path" version of the MPLS forwarder, covering both the ILM and FTN Forwarder functionality. If a packet cannot be fragmented, for example, it has the DF bit of its IP header set, an appropriate ICMP message "Destination Unreachable because fragmentation needed and DF set" is generated and processed in the same way as for the TTL_EXPIRED exception.</p>
ROUTER_ALERT	<p>The top MPLS label has the value of 1.</p> <p>In case of a ROUTER_ALERT exception, the packet is processed by the "slow path" of the MPLS core component in the way determined by the next label on the stack. If the packet is forwarded further, the "Router Alert Label" is pushed back onto the top of the label stack.</p>
TTL_EXPIRED	<p>The "outgoing" TTL field value is lower than 1.</p> <p>For a TTL_EXPIRED exception, an appropriate ICMP message "Time Exceeded" is generated, encapsulated with the label stack copied from the original packet, and sent to the original packet destination. For more details on this technique see [RFC3032], section 2.3.2.</p>
TOO_MANY_POPS	<p>After popping 3 labels from the packet's label stack, the next ILM entry indicates the POP operation (such a packet should be forwarded by the slow path to avoid excessive performance degradation).</p>

C Syntax

```
ix_error ix_cc_mpls_microblock_pkt_handler (
    ix_buffer_handle arg_hBuffer,
    ix_uint32 arg_ExceptionCode,
    void* arg_pContext)
```

Input

<code>arg_hBuffer</code>	Handle to a buffer which contains exception packets from the ILM and FTN Forwarder microblocks.
<code>arg_ExceptionCode</code>	Exception codes generated by the ILM and FTN Forwarder microblocks.
<code>arg_pContext</code>	Pointer to the MPLS core component context that is passed to the core component when a packet arrives. This context was defined by the core component and passed to core components infrastructure through the <code>ix_cc_mpls_init()</code> function.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. <code>IX_CC_ERROR_INVALID_EXCEP</code>—operation failed due to an unsupported exception code <code>IX_CC_ERROR_NULL</code>—operation failed due to an invalid pointer <code>IX_CC_MPLS_ERROR_INVALID_BUF_HANDLE</code>—operation failed due to an unusable buffer handle
--------------	--

29.3 Message Helper API

Table 29-4 lists the functions of the MPLS Forwarder Core Component Messaging API. Each Message Helper API function internally calls a corresponding MPLS core component Library API function. Refer to [Section 29.4, “Library API”](#) on page 719 for details.

Table 29-4. MPLS Forwarder Core Component Message Helper API

API	Description
<code>ix_cc_mpls_cb_general()</code>	Generic callback function
<code>ix_cc_mpls_async_lsp_create()</code>	Creates an LSP
<code>ix_cc_mpls_async_lsp_delete()</code>	Deletes an LSP
<code>ix_cc_mpls_async_lsp_modify()</code>	Modifies an LSP
<code>ix_cc_mpls_async_lsp_query()</code>	Reads an LSP table entry
<code>ix_cc_mpls_async_lsp_stats_query()</code>	Reads LSP statistics
<code>ix_cc_mpls_async_lsp_purge()</code>	Deletes all LSPs
<code>ix_cc_mpls_async_nhlfe_create()</code>	Creates an NHLFE
<code>ix_cc_mpls_async_nhlfe_delete()</code>	Deletes an NHLFE
<code>ix_cc_mpls_async_nhlfe_query()</code>	Reads an NHLFE table entry
<code>ix_cc_mpls_async_nhlfe_stats_query()</code>	Reads NHLFE statistics
<code>ix_cc_mpls_async_nhlfe_purge()</code>	Deletes all NHLFEs

Table 29-4. MPLS Forwarder Core Component Message Helper API (Continued)

API	Description
<code>ix_cc_mpls_async_nhlfe_set_create()</code>	Creates an NHLFE set
<code>ix_cc_mpls_async_nhlfe_set_delete()</code>	Deletes an NHLFE set
<code>ix_cc_mpls_async_nhlfe_set_modify()</code>	Modifies an NHLFE set
<code>ix_cc_mpls_async_nhlfe_set_query()</code>	Reads an NHLFE set
<code>ix_cc_mpls_async_param_query()</code>	Reads MPLS parameters area
<code>ix_cc_mpls_async_port_stats_query()</code>	Reads MPLS port statistics

29.3.1 `ix_cc_mpls_cb_general()`

This is a generic callback function type used by MPLS message helper functions that do not require data to be returned in the callback, and where only the result and context are relevant.

C Syntax

```
ix_error (*ix_cc_mpls_cb_general) (
    ix_error   arg_Result,
    void       *arg_pContext);
```

Input/Output

<code>arg_Result</code>	Indicates error conditions for the call.
<code>arg_pContext</code>	Pointer to the user provided context that was sent by API call. Used by the caller to identify the instance of the request.

29.3.2 `ix_cc_mpls_async_lsp_create()`

This message helper function creates an LSP description defining a FEC to NHLFE, or ILM to NHLFE mapping, and creates corresponding IP Routing table, ILM and NHLFE entries (if necessary). See `ix_cc_mpls_lsp_create()` for a detailed description.

C Syntax

```
ix_error ix_cc_mpls_async_lsp_create(
    ix_cc_mpls_lsp      *arg_pLsp,
    ix_cc_mpls_cb_lsp_create arg_Callback,
    void                 *arg_pUserContext);
```

Input

<code>arg_pLsp</code>	Pointer to data defining an LSP.
<code>arg_Callback</code>	Pointer to the <code>ix_cc_mpls_cb_lsp_create</code> callback function.
<code>arg_pUserContext</code>	Pointer to the client defined context that will be passed to the callback function.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. <code>IX_CC_ERROR_NULL</code>—operation failed due to an invalid pointer <code>IX_CC_ERROR_SEND_FAIL</code>—operation failed, failure from Message Support Library
--------------	---

29.3.2.1 `ix_cc_mpls_cb_lsp_create()`

This is the function prototype for the callback function used to return message-specific data (in this case, information on the LSP that was created) to the calling application.

C Syntax

```
ix_error (*ix_cc_mpls_cb_lsp_create) (
    ix_error          arg_Result,
    void              *arg_pContext,
    ix_cc_mpls_nhlfe_handle arg_nhlfeSetHandle,
    ix_uint16          arg_nhlfeHandles_num,
    ix_cc_mpls_nhlfe_handle arg_nhlfeHandles[MPLS_MAX_NHLFE_IN_SET],
    ix_uint32          arg_LspHandle);
```

Input/Output

<code>arg_Result</code>	Indicates error conditions for the call.
<code>arg_pContext</code>	Pointer to the user provided context that was sent by API call. Used by the caller to match the asynchronous response with the request.
<code>arg_nhlfeHandles_num</code>	If greater than 1, it means that created entry is associated with NHLFE set. In such case, it denotes the number of member NHLFE in a set.

Input/Output (Continued)

<code>arg_pnhlfeHandles</code>	Pointer to NHLFE associated with the current LSP entry. If LSP entry is associated with NHLFE SET, this is the pointer to the table with handles to all members of NHLFE set.
<code>arg_nhlfeSetHandle</code>	If created entry is associated with NHLFE set, this is the handle to that NHLFE set.
<code>arg_LspHandle</code>	Handle identifying the created LSP.

29.3.3 `ix_cc_mpls_async_lsp_delete()`

This message helper function deletes an LSP description. See `ix_cc_mpls_lsp_delete()` for a detailed description.

C Syntax

```
ix_error ix_cc_mpls_async_lsp_delete(
    ix_cc_mpls_lsp_handle    arg_LspHandle,
    ix_cc_mpls_cb_general    arg_Callback
    void                    *arg_pContext);
```

Input

<code>arg_LspHandle</code>	Handle identifying an LSP.
<code>arg_Callback</code>	Pointer to the client provided <code>ix_cc_mpls_cb_general</code> callback function.
<code>arg_pUserContext</code>	Pointer to the client defined context that will be passed to the callback function.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. <code>IX_CC_ERROR_NULL</code>—operation failed due to an invalid pointer <code>IX_CC_ERROR_SEND_FAIL</code>—operation failed, failure from Message Support Library
--------------	---

29.3.4 `ix_cc_mpls_async_lsp_modify()`

This message helper function modifies the NHLFE set mapping for a given LSP. See `ix_cc_mpls_lsp_modify()` for a detailed description.

C Syntax

```
ix_error ix_cc_mpls_async_lsp_modify(
    ix_cc_mpls_lsp_handle      arg_LspHandle,
    ix_uint8                   arg_nhlfeInfoType,
    ix_cc_mpls_nhlfe_handle    arg_nhlfeSetHandle,
    ix_uint16                   arg_nhlfeHandles_num,
    ix_cc_mpls_nhlfe_handle    arg_nhlfeHandles[MPLS_MAX_NHLFE_IN_SET],
    ix_cc_mpls_lsp_param       arg_lspParams[MPLS_MAX_NHLFE_IN_SET],
    ix_cc_mpls_cb_lsp_modify   arg_Callback,
    void                        *arg_pUserContext);
```

Input

arg_LspHandle	Handle identifying an LSP.
arg_nhlfeInfoType	Specifies whether the function should use a handle to a predefined NHLFE set or the table of single NHLFEs.
arg_nhlfeSetHandle	If LSP entry is to be associated with an existing NHLFE set, this is the handle to that set.
arg_nhlfeHandles_num	Number of valid NHLFE handles in the arg_nhlfeHandles array.
arg_nhlfeHandles	Array handles identifying NHLFEs to be associated with the LSP entry.
arg_lspParams	Array of LSP parameters for each element of the arg_nhlfeHandles array.
arg_Callback	Pointer to the client provided ix_cc_mpls_cb_lsp_modify callback function.
arg_pUserContext	Pointer to the client defined context that will be passed to the callback function.

Output/Returns

Return Value	Returns a valid ix_error. <ul style="list-style-type: none"> IX_SUCCESS—the operation succeeded. IX_CC_ERROR_NULL—operation failed due to an invalid pointer IX_CC_ERROR_SEND_FAIL—operation failed, failure from Message Support Library
--------------	--

29.3.4.1 ix_cc_mpls_cb_lsp_modify()

This is the function prototype for the callback function used to return message-specific data (in this case, information on the LSP that was changed) to the calling application.

C Syntax

```
typedef ix_error (*ix_cc_mpls_cb_lsp_modify) (
```

```

ix_error          arg_Result,
void              *arg_pContext,
ix_cc_mpls_nhlfe_handle arg_nhlfeSetHandle);

```

Input/Output

<code>arg_Result</code>	Indicates error conditions for the call.
<code>arg_pContext</code>	Pointer to the user provided context that was sent by API call. Used by the caller to identify the instance of the request.
<code>arg_nhlfeSetHandle</code>	If during the LSP modification an NHLFE set has been created, this is the handle to that set.

29.3.5 `ix_cc_mpls_async_lsp_query()`

This message helper function returns an LSP description.

C Syntax

```

ix_error ix_cc_mpls_async_lsp_query(
    ix_uint32          arg_LspHandle,
    ix_cc_mpls_cb_lsp_query arg_Callback,
    void              *arg_pUserContext);

```

Input

<code>arg_LspHandle</code>	Handle identifying an LSP
<code>arg_Callback</code>	Pointer to the client provided <code>ix_cc_mpls_cb_lsp_query</code> callback function.
<code>arg_pUserContext</code> <code>t</code>	Pointer to the client defined context that will be passed to the callback function.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. <code>IX_CC_ERROR_NULL</code>—operation failed due to an invalid pointer <code>IX_CC_ERROR_SEND_FAIL</code>—operation failed, failure from Message Support Library
--------------	---

29.3.5.1 `ix_cc_mpls_cb_lsp_query()`

This is the function prototype for the callback function used to return message-specific data (in this case, information on the LSP) to the calling application.

C Syntax

```
ix_error (*ix_cc_mpls_cb_lsp_query) (
    ix_error          arg_Result,
    void              *arg_pContext,
    ix_cc_mpls_lsp    *arg_Lsp);
```

Input/Output

arg_Result	Indicates error conditions for the call.
arg_pContext	Pointer to the user provided context that was sent by API call. Used by the caller to identify the instance of the request.
arg_Lsp	Pointer to the buffer storing the LSP information returned by the function.

29.3.6 ix_cc_mpls_async_lsp_stats_query()

This message helper function reads LSP statistics.

C Syntax

```
ix_error ix_cc_mpls_async_lsp_stats_query(
    ix_uint32          arg_LspHandle,
    ix_cc_mpls_cb_lsp_stats_query arg_Callback,
    void*              arg_pUserContext);
```

Input

arg_LspHandle	Handle identifying an LSP
arg_Callback	Pointer to the client provided ix_cc_mpls_cb_lsp_stats_query callback function.
arg_pUserContext	Pointer to the client defined context that will be passed to the callback function.

Output/Returns

Return Value	Returns a valid ix_error. <ul style="list-style-type: none"> IX_SUCCESS—the operation succeeded. IX_CC_ERROR_NULL—operation failed due to an invalid pointer IX_CC_ERROR_SEND_FAIL—operation failed, failure from Message Support Library
--------------	--

29.3.6.1 ix_cc_mpls_cb_lsp_stats_query()

This is the function prototype for the callback function used to return message-specific data (in this case, LSP statistics) to the calling application.

C Syntax

```
ix_error (*ix_cc_mpls_cb_lsp_stats_query) (
    ix_error          arg_Result,
    void              *arg_pContext,
    ix_cc_mpls_lsp_stats *arg_LspStatistics);
```

Input/Output

<code>arg_Result</code>	Indicates error conditions for the call.
<code>arg_pContext</code>	Pointer to the user provided context that was sent by API call. Used by the caller to identify the instance of the request.
<code>arg_LspStatistics</code>	Pointer to the buffer storing the LSP statistics information returned by the function.

29.3.7 `ix_cc_mpls_async_lsp_purge()`

This message helper function deletes all LSP definitions, together with its corresponding IP routing table and ILM entries.

C Syntax

```
ix_error ix_cc_mpls_async_lsp_purge (
    ix_cc_mpls_cb_general arg_Callback,
    void                  *arg_pUserContext);
```

Input

<code>arg_LspHandle</code>	Handle identifying an LSP.
<code>arg_Callback</code>	Pointer to the client provided <code>ix_cc_mpls_cb_general</code> callback function.
<code>arg_pUserContext</code> <code>t</code>	Pointer to the client defined context that will be passed to the callback function.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • <code>IX_CC_ERROR_NULL</code>—operation failed due to an invalid pointer • <code>IX_CC_ERROR_SEND_FAIL</code>—operation failed, failure from Message Support Library
--------------	---

29.3.8 ix_cc_mpls_async_nhlfe_create()

This message helper function creates an NHLFE table entry. See [ix_cc_mpls_nhlfe_create\(\)](#) for a detailed description.

C Syntax

```
ix_error ix_cc_mpls_async_nhlfe_create(
    ix_cc_mpls_nhlfe          *arg_pNhlfe,
    ix_cc_mpls_cb_nhlfe_create arg_Callback,
    void                      *arg_pUserContext);
```

Input

<code>arg_pNhlfe</code>	Pointer to the data defining an NHLFE.
<code>arg_Callback</code>	Pointer to the client provided <code>ix_cc_mpls_cb_nhlfe_create</code> callback function
<code>arg_pUserContext</code>	Pointer to the client defined context that will be passed to the callback function.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. <code>IX_CC_ERROR_NULL</code>—operation failed due to an invalid pointer <code>IX_CC_ERROR_SEND_FAIL</code>—operation failed, failure from Message Support Library
--------------	---

29.3.8.1 ix_cc_mpls_cb_nhlfe_create()

This is the function prototype for the callback function used to return message-specific data (in this case, information on the NHLFE table entry that was created) to the calling application.

C Syntax

```
ix_error (*ix_cc_mpls_cb_nhlfe_create) (
    ix_error          arg_Result,
    void              *arg_pContext,
    ix_cc_mpls_nhlfe_handle arg_NhlfeHandle);
```

Input/Output

<code>arg_Result</code>	Indicates error conditions for the call.
<code>arg_pContext</code>	Pointer to the user provided context that was sent by API call. Used by the caller to identify the instance of the request.
<code>arg_NhlfeHandle</code>	Pointer to the handle identifying the created NHLFE returned by the function.

29.3.9 `ix_cc_mpls_async_nhlfe_delete()`

This message helper function deletes an NHLFE entry. See `ix_cc_mpls_nhlfe_delete()` for a detailed description.

C Syntax

```
ix_error ix_cc_mpls_async_nhlfe_delete(
    ix_cc_mpls_nhlfe_handle arg_NhlfeHandle,
    ix_cc_mpls_cb_general   arg_Callback,
    void                    *arg_pUserContext);
```

Input

<code>arg_NhlfeHandle</code>	Handle identifying an NHLFE.
<code>arg_Callback</code>	Pointer to the client provided callback function
<code>arg_pUserContext</code> <code>t</code>	Pointer to the client defined context that will be passed to the callback function.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. <code>IX_CC_ERROR_NULL</code>—operation failed due to an invalid pointer <code>IX_CC_ERROR_SEND_FAIL</code>—operation failed, failure from Message Support Library
--------------	---

29.3.10 `ix_cc_mpls_async_nhlfe_query()`

This message helper function reads a regular NHLFE entry.

C Syntax

```
ix_error ix_cc_mpls_async_nhlfe_query(
```

```
ix_cc_mpls_nhlfe_handle    arg_NhlfeHandle,
ix_cc_mpls_cb_nhlfe_query  arg_Callback,
void                      *arg_pUserContext);
```

Input

arg_NhlfeHandle	Handle identifying an NHLFE.
arg_Callback	Pointer to the client provided ix_cc_mpls_cb_nhlfe_query callback function.
arg_pUserContext	Pointer to the client defined context that will be passed to the callback function.

29.3.10.1 ix_cc_mpls_cb_nhlfe_query()

This is the function prototype for the callback function used to return message-specific data (in this case, information on the NHLFE entry) to the calling application.

C Syntax

```
ix_error (ix_cc_mpls_cb_nhlfe_query) (
    ix_error      arg_Result,
    void          *arg_pContext,
    ix_cc_mpls_nhlfe *arg_Nhlfe);
```

Input/Output

arg_Result	Indicates error conditions for the call.
arg_pContext	Pointer to the user provided context that was sent by API call. Used by the caller to identify the instance of the request.
arg_Nhlfe	Pointer to the buffer storing the NHLFE information returned by the function.

29.3.11 ix_cc_mpls_async_nhlfe_stats_query()

This message helper function reads NHLFE statistics.

C Syntax

```
ix_error ix_cc_mpls_async_nhlfe_stats_query(
    ix_cc_mpls_nhlfe_handle    arg_NhlfeHandle,
    ix_cc_mpls_cb_nhlfe_stats_query  arg_Callback,
    void                      *arg_pUserContext);
```

Input

<code>arg_NhlfeHandle</code>	Handle identifying an NHLFE.
<code>arg_Callback</code>	Pointer to the client provided <code>ix_cc_mpls_cb_nhlfe_stats_query</code> callback function.
<code>arg_pUserContext</code> <code>t</code>	Pointer to the client defined context that will be passed to the callback function.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. <code>IX_CC_ERROR_NULL</code>—operation failed due to an invalid pointer <code>IX_CC_ERROR_SEND_FAIL</code>—operation failed, failure from Message Support Library
--------------	---

29.3.11.1 `ix_cc_mpls_cb_nhlfe_stats_query()`

This is the function prototype for the callback function used to return message-specific data (in this case, information on NHLFE statistics) to the calling application.

C Syntax

```
ix_error (*ix_cc_mpls_cb_nhlfe_stats_query) (
    ix_error          arg_Result,
    void              *arg_pContext,
    ix_cc_mpls_nhlfe_stats *arg_NhlfeStats);
```

Input/Output

<code>arg_Result</code>	Indicates error conditions for the call.
<code>arg_pContext</code>	Pointer to the user provided context that was sent by API call. Used by the caller to identify the instance of the request.
<code>arg_NhlfeStats</code>	Pointer to the buffer storing the LSP statistics information returned by the function.

29.3.12 `ix_cc_mpls_async_nhlfe_purge()`

This message helper function deletes all NHLFE table entries.

C Syntax

```
ix_error ix_cc_mpls_async_nhlfe_purge (
    ix_cc_mpls_cb_general    arg_Callback,
    void                     *arg_pUserContext);
```

Input

<code>arg_Callback</code>	Pointer to the client provided callback function.
<code>arg_pUserContext</code> <code>t</code>	Pointer to the client defined context that will be passed to the callback function.

:

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. <code>IX_CC_ERROR_NULL</code>—operation failed due to an invalid pointer <code>IX_CC_ERROR_SEND_FAIL</code>—operation failed, failure from Message Support Library
--------------	---

29.3.13 ix_cc_mpls_async_nhlfe_set_create()

This message helper function creates an NHLFE table entry containing an NHLFE set. See [ix_cc_mpls_nhlfe_set_create\(\)](#) for a detailed description.

C Syntax

```
ix_error ix_cc_mpls_async_nhlfe_set_create(
    ix_uint16                arg_nhlfeSetSize,
    ix_cc_mpls_nhlfe_handle  arg_nhlfeSetHandles[MPLS_MAX_NHLFE_IN_SET],
    ix_cc_mpls_cb_nhlfe_set_create  arg_Callback,
    void                     *arg_pUserContext);
```

Input

<code>arg_nhlfeSetSize</code>	Number of elements in the NHLFE set.
<code>arg_nhlfeSetHandles</code>	Array of handles defining the new NHLFE set.
<code>arg_Callback</code>	Pointer to the client provided <code>ix_cc_mpls_cb_nhlfe_set_create</code> callback function.
<code>arg_pUserContext</code>	Pointer to the client defined context that will be passed to the callback function.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> .
	<ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • <code>IX_CC_ERROR_NULL</code>—operation failed due to an invalid pointer • <code>IX_CC_ERROR_SEND_FAIL</code>—operation failed, failure from Message Support Library

29.3.13.1 `ix_cc_mpls_cb_nhlfe_set_create()`

This is the function prototype for the callback function used to return message-specific data (in this case, information on the NHLFE table entry that was created) to the calling application.

C Syntax

```
ix_error (*ix_cc_mpls_cb_nhlfe_set_create) (
    ix_error      arg_Result,
    void          *arg_pContext,
    ix_cc_mpls_nhlfe_handle arg_NhlfeSetHandle);
```

Input

<code>arg_Result</code>	Indicates error conditions for the call.
<code>arg_pContext</code>	Pointer to the user provided context that was sent by API call. Used by the caller to identify the instance of the request.
<code>arg_NhlfeSetHandle</code>	Handle identifying the created NHLFE set entry.

29.3.14 `ix_cc_mpls_async_nhlfe_set_delete()`

This message helper function deletes an NHLFE entry containing an NHLFE set. See [ix_cc_mpls_nhlfe_set_delete\(\)](#) for a detailed description.

C Syntax

```
ix_error ix_cc_mpls_async_nhlfe_set_delete(
    ix_cc_mpls_nhlfe_handle arg_NhlfeSetHandle,
    ix_cc_mpls_cb_general   arg_Callback,
    void                   *arg_pUserContext);
```

Input

<code>arg_NhlfeSetHandle</code>	Handle identifying an NHLFE set entry.
<code>arg_Callback</code>	Pointer to the client provided callback function.
<code>arg_pUserContext</code>	Pointer to the client defined context that will be passed to the callback function.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. <code>IX_CC_ERROR_NULL</code>—operation failed due to an invalid pointer <code>IX_CC_ERROR_SEND_FAIL</code>—operation failed, failure from Message Support Library
--------------	---

29.3.15 `ix_cc_mpls_async_nhlfe_set_modify()`

This message helper function modifies the NHLFE table entry containing an NHLFE set. See [ix_cc_mpls_nhlfe_set_modify\(\)](#) for a detailed description.

C Syntax

```
ix_error ix_cc_mpls_async_nhlfe_set_modify(
    ix_cc_mpls_nhlfe_handle arg_NhlfeSetHandle,
    ix_uint16               arg_nhlfeSetSize,
    ix_cc_mpls_nhlfe_handle arg_nhlfeHandles[MPLS_MAX_NHLFE_IN_SET],
    ix_cc_mpls_cb_general   arg_Callback,
    void                    *arg_pUserContext);
```

Input

<code>arg_NhlfeSetHandle</code>	Handle identifying an NHLFE entry containing an NHLFE set.
<code>arg_nhlfeSetSize</code>	Number of elements in the new NHLFE set.
<code>arg_nhlfeHandles</code>	Array of handles identifying the new NHLFEs for the set.
<code>arg_Callback</code>	Pointer to the client provided callback function.
<code>arg_pUserContext</code>	Pointer to a client defined context that will be passed to the callback function.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> .
	<ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • <code>IX_CC_ERROR_NULL</code>—operation failed due to an invalid pointer • <code>IX_CC_ERROR_SEND_FAIL</code>—operation failed, failure from Message Support Library

29.3.16 `ix_cc_mpls_async_nhlfe_set_query()`

This message helper function reads an NHLFE entry containing an NHLFE set.

C Syntax

```
ix_error ix_cc_mpls_async_nhlfe_set_query(
    ix_cc_mpls_nhlfe_handle      arg_NhlfeSetHandle,
    ix_cc_mpls_cb_nhlfe_set_query arg_Callback,
    void                          *arg_pUserContext);
```

Input

<code>arg_NhlfeSetHandle</code>	Handle identifying an NHLFE entry containing an NHLFE set.
<code>arg_Callback</code>	Pointer to the client provided <code>ix_cc_mpls_cb_nhlfe_set_query</code> callback function.
<code>arg_pUserContext</code>	Pointer to the client defined context that will be passed to the callback function.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> .
	<ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • <code>IX_CC_ERROR_NULL</code>—operation failed due to an invalid pointer • <code>IX_CC_ERROR_SEND_FAIL</code>—operation failed, failure from Message Support Library

29.3.17 `ix_cc_mpls_async_param_query()`

This message helper function reads MPLS parameters.

C Syntax

```
ix_error ix_cc_mpls_async_param_query (
    ix_cc_mpls_cb_param_query    arg_Callback,
    void                          *arg_pUserContext);
```

Input

<code>arg_Callback</code>	Pointer to the client provided <code>ix_cc_mpls_cb_param_query</code> callback function
<code>arg_pUserContext</code>	Pointer to the client defined context that will be passed to the callback function.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. <code>IX_CC_ERROR_NULL</code>—operation failed due to an invalid pointer <code>IX_CC_ERROR_SEND_FAIL</code>—operation failed, failure from Message Support Library
--------------	---

29.3.17.1 `ix_cc_mpls_cb_param_query()`

This is the function prototype for the callback function used to return message-specific data (in this case, information on MPLS parameters) to the calling application.

C Syntax

```
ix_error (*ix_cc_mpls_cb_param_query) (
    ix_error          arg_Result,
    void              *arg_pContext,
    ix_cc_mpls_params *arg_pMplsParameters);
```

Input/Output

<code>arg_Result</code>	Indicates error conditions for the call.
<code>arg_pContext</code>	Pointer to the user provided context that was sent by API call. Used by the caller to identify the instance of the request.
<code>arg_pMplsParameters</code>	Pointer to the buffer storing the MPLS parameters returned by the function.

29.3.18 `ix_cc_mpls_async_port_stats_query()`

This message helper function reads MPLS statistics for a given input port.

C Syntax

```
ix_error ix_cc_mpls_async_port_stats_query (
    ix_uint16          arg_PortIndex,
```

```
ix_cc_mpls_cb_port_stats_query    arg_Callback,
void                             *arg_pUserContext);
```

Input

arg_PortIndex	Local port index.
arg_Callback	Pointer to the client provided ix_cc_mpls_cb_port_stats_query callback function
arg_pUserContext t	Pointer to the client defined context that will be passed to the callback function.

Output/Returns

Return Value	Returns a valid ix_error. <ul style="list-style-type: none"> IX_SUCCESS—the operation succeeded. IX_CC_ERROR_NULL—operation failed due to an invalid pointer IX_CC_ERROR_SEND_FAIL—operation failed, failure from Message Support Library
--------------	--

29.3.18.1 ix_cc_mpls_cb_port_stats_query()

This is the function prototype for the callback function used to return message-specific data (in this case, MPLS port statistics information) to the calling application.

C Syntax

```
ix_error (*ix_cc_mpls_cb_port_stats_query) (
    ix_error      arg_Result,
    void          *arg_pContext,
    ix_cc_mpls_port_stats *arg_pMplsPortStatistics);
```

Input/Output

arg_Result	Indicates error conditions for the call.
arg_pContext	Pointer to the user provided context that was sent by API call. Used by the caller to identify the instance of the request.
arg_pMplsPortStatistics	Pointer to the buffer storing the MPLS port statistics returned by the function.

29.4 Library API

Table 29-5 lists the functions of the MPLS Forwarder Core Component Library API.

Table 29-5. MPLS Forwarder Core Component Library API

API	Description
<code>ix_cc_mpls_lsp_create()</code>	Creates an LSP
<code>ix_cc_mpls_lsp_delete()</code>	Deletes an LSP
<code>ix_cc_mpls_lsp_modify()</code>	Modifies an LSP
<code>ix_cc_mpls_lsp_query()</code>	Reads an LSP table entry
<code>ix_cc_mpls_lsp_stats_query()</code>	Reads LSP statistics
<code>ix_cc_mpls_lsp_purge()</code>	Deletes all LSPs
<code>ix_cc_mpls_nhlfe_create()</code>	Creates an NHLFE
<code>ix_cc_mpls_nhlfe_delete()</code>	Deletes an NHLFE
<code>ix_cc_mpls_nhlfe_query()</code>	Reads an NHLFE
<code>ix_cc_mpls_nhlfe_stats_query()</code>	Reads NHLFE statistics
<code>ix_cc_mpls_nhlfe_purge()</code>	Deletes all NHLFEs
<code>ix_cc_mpls_nhlfe_set_create()</code>	Creates an NHLFE set
<code>ix_cc_mpls_nhlfe_set_delete()</code>	Deletes an NHLFE set
<code>ix_cc_mpls_nhlfe_set_modify()</code>	Modifies an NHLFE set
<code>ix_cc_mpls_nhlfe_set_query()</code>	Reads an NHLFE set
<code>ix_cc_mpls_param_query()</code>	Reads MPLS parameters
<code>ix_cc_mpls_port_stats_query()</code>	Reads MPLS port statistics

29.4.1 `ix_cc_mpls_lsp_create()`

This function creates an LSP description that is a mapping of an incoming segment to an outgoing segment, and sets up appropriate MPLS forwarding table entries.

From the MPLS Forwarder Core Component, there are two types of LSPs:

- ingress LSP (LSP_FEC_TYPE), on the ingress edge of an MPLS domain, receiving IP traffic, and transmitting MPLS traffic. In this case, the incoming segment is defined by a forwarding equivalence class (FEC), that is an IP address and mask. The outgoing segment is determined by one or more NHLFE table entries.
- transit LSP (LSP_ILM_TYPE), inside or on the egress edge of an MPLS domain, receiving MPLS traffic, and transmitting either MPLS or IP traffic. In this case, the incoming segment is defined by an MPLS label and incoming interface. The outgoing segment is determined by one or more NHLFE table entries.

For an ingress LSP (LSP_FEC_TYPE), the function checks whether an IPv4 routing table entry for the given FEC exists; if so, an error is returned because the FEC belongs to the IPv4 Forwarder. Otherwise, the outgoing segment specification is processed.

The outgoing segment can be defined in the following mutually exclusive ways:

- explicit NHLFE data for one or more NHLFE table entries
- one or more NHLFE handles
- an NHLFE set handle

For explicit NHLFE data, the function creates corresponding NHLFE table entries, and the NHLFE set comprising those entries in the case of multiple NHLFEs. The NHLFE set is also created in the second case for multiple NHLFE handles. Both the new NHLFE handles and the NHLFE set handle will be returned to the caller.

Next, the IPv4 next hop and routing table entries for the given FEC are added. Finally, a CC_LSP table entry linking the FEC ID with the NHLFE outSeg ID is created, and the LSP handle is returned to the caller.

For a transit LSP (LSP_ILM_TYPE), the function checks whether an ILM table entry for the given label and input port exists; if so, an error is returned. Otherwise, the outgoing segment specification is processed in the same way as described for an ingress LSP.

Next, the ILM table entry for the given label and input port is added. Finally, a CC_LSP table entry linking the ILM entry with the NHLFE outSeg ID is created, and the LSP handle is returned to the caller.

Note: The following items should be noted:

1. An NHLFE or NHLFE set can only belong to one ingress LSP (LSP_FEC_TYPE), whereas at the same time, it can belong to many transit LSPs (LSP_ILM_TYPE).
2. NHLFE and NHLFE set entries created implicitly by the `ix_cc_mpls_lsp_create()` function must be deleted by explicit calls to the `ix_cc_mpls_nhlfe_delete()` function.
3. Each NHLFE and NHLFE set entry has an associated reference count storing the number of LSPs it belongs to. It is incremented by `ix_cc_mpls_lsp_create()` and decremented by `ix_cc_mpls_lsp_delete()`.

C Syntax

```
ix_error ix_cc_mpls_lsp_create(
    ix_cc_mpls_lsp          *arg_pLsp,
    ix_cc_mpls_lsp_handle   *arg_pLspHandle,
    ix_cc_mpls_context      *arg_pContext);
```

Input

`arg_pContext` Pointer to the MPLS core component context.

Input/Output

<code>arg_pLsp</code>	Pointer to an <code>ix_cc_mpls_lsp</code> structure, defining an LSP. On return, this structure can contain handles to implicitly created NHLFE and NHLFE set entries.
<code>arg_pLspHandle</code>	Pointer to the handle identifying the created LSP.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. <code>IX_CC_ERROR_NULL</code>—operation failed due to an invalid pointer. <code>IX_CC_ERROR_DUPLICATE_ENTRY</code> – the specified FEC or label already exists. <code>IX_CC_ERROR_FULL</code> – no place in the ILM or NHLFE table.
--------------	---

29.4.2 `ix_cc_mpls_lsp_delete()`

This function deletes an LSP description from the CC_LSP table, together with its corresponding IP Routing table and ILM table entries. NHLFE and NHLFE set entries constituting the outgoing segment for a given LSP are not deleted; only their reference counts are decremented.

C Syntax

```
ix_error ix_cc_mpls_lsp_delete(
    ix_cc_mpls_lsp_handle arg_LspHandle,
    void                  *arg_pContext);
```

Input

<code>arg_pLspHandle</code>	Pointer to the handle identifying an LSP.
<code>arg_pContext</code>	Pointer to the MPLS core component context.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. <code>IX_CC_ERROR_NULL</code>—operation failed due to an invalid pointer. <code>IX_CC_ERROR_ENTRY_NOT_FOUND</code>—operation failed, LSP or NHLFE does not exist.
--------------	--

29.4.3 ix_cc_mpls_lsp_modify()

This function modifies the incoming to outgoing segment mapping for a given LSP. The new outgoing segment can be specified either as an NHLFE set handle, or as up to four NHLFE handles pointing to regular (non-set) NHLFE table entries.

C Syntax

```
ix_error ix_cc_mpls_lsp_modify(
    ix_cc_mpls_lsp_handle    arg_LspHandle,
    ix_uint8                 arg_nhlfeInfoType,
    ix_cc_mpls_nhlfe_handle  *arg_pNhlfeSetHandle,
    ix_uint16                arg_nhlfeHandles_num,
    ix_cc_mpls_nhlfe_handle  arg_nhlfeHandles[MPLS_MAX_NHLFE_IN_SET],
    ix_cc_mpls_lsp_param     arg_lspParams[MPLS_MAX_NHLFE_IN_SET],
    void                     *arg_pContext);
```

Input

<code>arg_pLspHandle</code>	Pointer to the handle identifying the modified LSP.
<code>arg_nhlfeInfoType</code>	Specifies whether the function should use a handle to a predefined NHLFE set in <code>arg_pNhlfeSetHandle</code> , or regular NHLFE handles in <code>arg_nhlfeHandles</code> .
<code>arg_nhlfeHandles_num</code>	Number of valid NHLFE handles in the <code>arg_nhlfeHandles</code> array.
<code>arg_nhlfeHandles</code>	Handles identifying NHLFEs to be associated with the LSP entry.
<code>arg_lspParams</code>	Array of LSP parameters for each element of the <code>arg_nhlfeHandles</code> array.
<code>arg_pContext</code>	Pointer to the MPLS core component context.

Input/Output

<code>arg_pNhlfeSetHandle</code>	On input, points to an NHLFE set handle, if <code>arg_nhlfeInfoType</code> equals to <code>NHLFE_SET_TYPE</code> . On output, points to an NHLFE set handle, if an NHLFE set was created during the LSP modification.
----------------------------------	--

Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • <code>IX_CC_ERROR_NULL</code>—operation failed due to an invalid pointer. • <code>IX_CC_ERROR_ENTRY_NOT_FOUND</code>—operation failed, LSP or NHLFE does not exist. • <code>IX_CC_MPLS_ERROR_ENTRY_NHLFE_SET</code> – one of the NHLFEs is an NHLFE set entry. • <code>IX_CC_MPLS_ERROR_ENTRY_USED</code> – an NHLFE is already used by a <code>FEC_TYPE</code> LSP.
--------------	---

29.4.4 ix_cc_mpls_lsp_query()

This function reads an LSP description.

C Syntax

```
ix_error ix_cc_mpls_lsp_query(
    ix_cc_mpls_lsp_handle arg_LspHandle,
    ix_cc_mpls_lsp        *arg_pLsp,
    void                  *arg_pContext);
```

Input

<code>arg_pLspHandle</code>	Pointer to the handle identifying the LSP.
<code>arg_pContext</code>	Pointer to the MPLS core component context.

Input/Output

<code>arg_pLsp</code>	Pointer to the buffer for storing the LSP information returned by the function.
-----------------------	---

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • <code>IX_CC_ERROR_NULL</code>—operation failed due to an invalid pointer. • <code>IX_CC_ERROR_ENTRY_NOT_FOUND</code>—operation failed, LSP or NHLFE does not exist.
--------------	--

29.4.5 ix_cc_mpls_lsp_stats_query()

This function reads LSP statistics.

C Syntax

```
ix_error ix_cc_mpls_lsp_stats_query(
    ix_cc_mpls_lsp_handle arg_LspHandle,
    ix_cc_mpls_lsp_stats *arg_pLspStatistics,
    void *arg_pContext);
```

Input

arg_pLspHandle Pointer to the handle identifying the LSP.

arg_pContext Pointer to the MPLS core component context.

Input/Output

arg_pLspStatistics Pointer to the buffer for storing the LSP statistics returned by the function.

Returns

Return Value Returns a valid `ix_error`.

- `IX_SUCCESS`—the operation succeeded.
- `IX_CC_ERROR_NULL`—operation failed due to an invalid pointer.
- `IX_CC_ERROR_ENTRY_NOT_FOUND`—operation failed, LSP or NHLFE does not exist.

29.4.6 ix_cc_mpls_lsp_purge()

This function deletes all LSP definitions, together with its corresponding IP Routing table and ILM entries.

C Syntax

```
ix_error ix_cc_mpls_lsp_purge(void *arg_pContext);
```

Input

arg_pContext Pointer to the MPLS core component context.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> .
	<ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • <code>IX_CC_ERROR_NULL</code>—operation failed due to an invalid pointer.

29.4.7 `ix_cc_mpls_nhlfe_create()`

This function creates an NHLFE and its corresponding CC_NHLFE table entries. Apart from forwarding data, each NHLFE has the following two attributes stored in its CC_NHLFE table entry:

- reference count, specifying the number of LSPs that use the NHLFE entry
- flag, denoting whether this entry is an element of an NHLFE set

On creation, the reference count field is set to 0. It is incremented by each LSP creation that specifies this NHLFE as its outgoing segment, and is decremented by such an LSP deletion.

An NHLFE entry can be used by none, one, or multiple LSPs.

An NHLFE entry can be an element of none or one NHLFE set.

C Syntax

```
ix_error ix_cc_mpls_nhlfe_create(
    ix_cc_mpls_nhlfe      arg_Nhlfe,
    ix_cc_mpls_nhlfe_handle *arg_pNhlfeHandle,
    void                  *arg_pContext);
```

Input

<code>arg_Nhlfe</code>	NHLFE data.
<code>arg_pContext</code>	Pointer to the MPLS core component context.

Input/Output

<code>arg_pNhlfeHandle</code>	Pointer to the handle identifying the created NHLFE.
-------------------------------	--

Returns

Return Value	Returns a valid <code>ix_error</code> .
	<ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • <code>IX_CC_ERROR_NULL</code>—operation failed due to an invalid pointer. • <code>IX_CC_ERROR_FULL</code>—operation failed, no place in the NHLFE table.

29.4.8 `ix_cc_mpls_nhlfe_delete()`

This function deletes an NHLFE and its corresponding CC_NHLFE table entries.

An NHLFE cannot be deleted as long as it is used by an LSP (that is, its reference count is greater than 0), or it is an element of an NHLFE set.

This function cannot be used for deleting NHLFEs containing set descriptions; for such entries use the `ix_cc_mpls_nhlfe_set_delete()` function.

C Syntax

```
ix_error ix_cc_mpls_nhlfe_delete(
    ix_cc_mpls_nhlfe_handle    arg_NhlfeHandle,
    ix_cc_mpls_context         *arg_pContext);
```

Input

<code>arg_NhlfeHandle</code>	Pointer to the handle identifying an NHLFE.
<code>arg_pContext</code>	Pointer to the MPLS core component context.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> .
	<ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • <code>IX_CC_ERROR_NULL</code>—operation failed due to an invalid pointer. • <code>IX_CC_ERROR_ENTRY_NOT_FOUND</code>—operation failed, LSP or NHLFE does not exist. • <code>IX_CC_MPLS_ERROR_ENTRY_NHLFE_SET</code>—operation failed, this is an NHLFE set entry. • <code>IX_CC_MPLS_ERROR_ENTRY_USED</code>—operation failed, entry referenced by an LSP.

29.4.9 `ix_cc_mpls_nhlfe_query()`

This function reads a regular (non-set) NHLFE entry.

This function cannot be used for reading NHLFEs containing set descriptions; for such entries use the `ix_cc_mpls_nhlfe_set_query()` function.

C Syntax

```
ix_error ix_cc_mpls_nhlfe_query(
    ix_cc_mpls_nhlfe_handle arg_NhlfeHandle,
    ix_cc_mpls_nhlfe        *arg_pNhlfe,
    ix_cc_mpls_context      *arg_pContext);
```

Input

<code>arg_pNhlfeHandle</code>	Pointer to the handle identifying an NHLFE.
<code>arg_pContext</code>	Pointer to the MPLS core component context.

Input/Output

<code>arg_pNhlfe</code>	Pointer to the buffer for storing the NHLFE information returned by the function.
-------------------------	---

Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. <code>IX_CC_ERROR_NULL</code>—operation failed due to an invalid pointer. <code>IX_CC_ERROR_ENTRY_NOT_FOUND</code>—operation failed, NHLFE does not exist. <code>IX_CC_MPLS_ERROR_ENTRY_NHLFE_SET</code>—operation failed, this is an NHLFE set entry.
--------------	--

29.4.10 `ix_cc_mpls_nhlfe_stats_query()`

This function reads NHLFE statistics. This function cannot be used for reading NHLFEs containing set descriptions; statistics are valid only for regular NHLFEs.

C Syntax

```
ix_error ix_cc_mpls_nhlfe_stats_query(
    ix_cc_mpls_nhlfe_handle arg_NhlfeHandle,
    ix_cc_mpls_nhlfe_stats *arg_pNhlfeStatistics,
    void                    *arg_pContext);
```

Input

`arg_pNhlfeHandle` Pointer to the handle identifying an NHLFE.

`arg_pContext` Pointer to the MPLS core component context.

Input/Output

`arg_NhlfeStatistics` Pointer to the buffer for storing the NHLFE statistics returned by the function.

Output/Returns

Return Value Returns a valid `ix_error`.

- `IX_SUCCESS`—the operation succeeded.
- `IX_CC_ERROR_NULL`—operation failed due to an invalid pointer.
- `IX_CC_ERROR_ENTRY_NOT_FOUND`—operation failed, NHLFE does not exist.
- `IX_CC_MPLS_ERROR_ENTRY_NHLFE_SET`—operation failed, this is an NHLFE set entry.

29.4.11 `ix_cc_mpls_nhlfe_purge()`

This function deletes all NHLFE table entries. If any of the entries has a non-zero reference count field, no entry is deleted. This function can be used for clearing the NHLFE table before any NHLFE is assigned to an LSP.

C Syntax

```
ix_error ix_cc_mpls_nhlfe_purge (
    ix_cc_mpls_context *arg_pContext);
```

Input

`arg_pContext` Pointer to the MPLS core component context.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> .
	<ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • <code>IX_CC_ERROR_NULL</code>—operation failed due to an invalid pointer. • <code>IX_CC_ERROR_ENTRY_USED</code>—operation failed, an NHLFE is referenced by an LSP.

29.4.12 `ix_cc_mpls_nhlfe_set_create()`

This function creates an NHLFE table entry containing an NHLFE set comprising up to four regular NHLFEs.

Note: The NHLFE handles specifying the set cannot point to other sets.

C Syntax

```
ix_error ix_cc_mpls_nhlfe_set_create(
    ix_uint16          arg_nhlfeSetSize,
    ix_cc_mpls_nhlfe_handle arg_nhlfeSetHandles[MPLS_MAX_NHLFE_IN_SET],
    ix_cc_mpls_nhlfe_handle *arg_pNhlfeHandle,
    void               *arg_pContext);
```

Input

<code>arg_nhlfeSetSize</code>	Number of elements in the NHLFE set.
<code>arg_nhlfeSetHandles</code>	Array of handles defining the new NHLFE set.
<code>arg_pContext</code>	Pointer to the MPLS core component context.

Input/Output

<code>arg_pNhlfeHandle</code>	Pointer to the handle identifying the created NHLFE set.
-------------------------------	--

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • <code>IX_CC_ERROR_NULL</code>—operation failed due to an invalid pointer. • <code>IX_CC_ERROR_RANGE</code>—operation failed, <code>arg_NhlfeSize</code> is outside the valid range of 1 to 4. • <code>IX_CC_ERROR_FULL</code>—operation failed, NHLFE table is full. • <code>IX_CC_ERROR_ENTRY_NOT_FOUND</code>—operation failed, LSP or NHLFE does not exist. • <code>IX_CC_MPLS_ERROR_ENTRY_NHLFE_SET</code>—operation failed, this is an NHLFE set entry.
--------------	--

29.4.13 `ix_cc_mpls_nhlfe_set_delete()`

This function deletes an NHLFE entry containing an NHLFE set. It does not delete the regular NHLFE table entries comprising the set; only the flag denoting their set membership is cleared. An NHLFE set entry cannot be deleted as long as it is used by any LSP.

This function cannot be used for deleting a regular NHLFE; for such an entry use the `ix_cc_mpls_nhlfe_delete()` function.

C Syntax

```
ix_error ix_cc_mpls_nhlfe_set_delete(
    ix_cc_mpls_nhlfe_handle    arg_NhlfeSetHandle,
    ix_cc_mpls_context         *arg_pContext);
```

Input

<code>arg_NhlfeSetHandle</code>	Handle identifying an NHLFE set entry.
<code>arg_pContext</code>	Pointer to the MPLS core component context.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • <code>IX_CC_ERROR_NULL</code>—operation failed due to an invalid pointer. • <code>IX_CC_ERROR_ENTRY_NOT_FOUND</code>—operation failed, NHLFE does not exist. • <code>IX_CC_ERROR_ENTRY_NHLFE</code>—operation failed, NHLFE is not a set entry. • <code>IX_CC_MPLS_ERROR_ENTRY_USED</code>—operation failed, an NHLFE is referenced by an LSP.
--------------	---

29.4.14 ix_cc_mpls_nhlfe_set_modify()

This function modifies the NHLFE table entry containing an NHLFE set. The NHLFE handles comprising the old set definition are replaced by the handles specified by the `arg_nhlfeSetSize` and `arg_nhlfeSetHandles[]` function arguments. The new set handles must point to regular (non-set) NHLFEs. The modified NHLFE set entry and its element cannot be used by any LSP.

C Syntax

```
ix_error ix_cc_mpls_nhlfe_set_modify(
    ix_cc_mpls_nhlfe_handle arg_NhlfeHandle,
    ix_uint16               arg_nhlfeSetSize,
    ix_cc_mpls_nhlfe_handle arg_nhlfeSetHandles[MPLS_MAX_NHLFE_IN_SET],
    void                   *arg_pContext);
```

Input

<code>arg_NhlfeHandle</code>	Handle identifying an NHLFE entry containing an NHLFE set.
<code>arg_nhlfeSetSize</code>	Number of elements in the new NHLFE set.
<code>arg_nhlfeSetHandles</code>	Array of handles defining the new NHLFEs for the set.
<code>arg_pContext</code>	Pointer to the MPLS core component context.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> .
	<ul style="list-style-type: none"> <code>IX_SUCCESS</code>—the operation succeeded. <code>IX_CC_ERROR_NULL</code>—operation failed due to an invalid pointer. <code>IX_CC_ERROR_ENTRY_NOT_FOUND</code>—operation failed, one of the set NHLFEs does not exist. <code>IX_CC_MPLS_ERROR_ENTRY_NHLFE_SET</code>—operation failed, one of the set NHLFEs is an NHLFE set entry. <code>IX_CC_MPLS_ERROR_ENTRY_USED</code>—operation failed, an NHLFE is referenced by an LSP.

29.4.15 ix_cc_mpls_nhlfe_set_query()

This function reads an NHLFE entry containing an NHLFE set. It returns only the NHLFE handles of the set elements. The definition of each set element must be read separately by calling the `ix_cc_mpls_nhlfe_query()` function for each returned handle.

C Syntax

```
ix_error ix_cc_mpls_nhlfe_set_query(
    ix_cc_mpls_nhlfe_handle arg_NhlfeSetHandle,
    ix_uint16               *arg_nhlfeSetSize,
    ix_cc_mpls_nhlfe_handle arg_nhlfeSetHandles[MPLS_MAX_NHLFE_IN_SET],
```

```
void                                *arg_pContext);
```

Input

<code>arg_pNhlfeSetHandle</code>	Handle identifying an NHLFE entry containing an NHLFE set.
<code>arg_pContext</code>	Pointer to the MPLS core component context.

Input/Output

<code>arg_nhlfeSetSize</code>	Pointer to the buffer storing the number of the NHLFE set handles returned by the function.
<code>arg_nhlfeSetHandles</code>	Pointer to the buffer for storing the NHLFE handles identifying the set elements returned by the function.

Output/Returns

Return Value	Returns a valid <code>ix_error</code> . <ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • <code>IX_CC_ERROR_NULL</code>—operation failed due to an invalid pointer. • <code>IX_CC_ERROR_ENTRY_NOT_FOUND</code>—operation failed, NHLFE does not exist. • <code>IX_CC_ERROR_ENTRY_NHLFE</code>—operation failed, NHLFE is not a set entry.
--------------	---

29.4.16 `ix_cc_mpls_param_query()`

This function returns MPLS parameters defined in the `ix_cc_mpls_params` structure.

C Syntax

```
ix_error ix_cc_mpls_param_query (
    ix_cc_mpls_params  *arg_pMplsParameters,
    void               *arg_pContext);
```

Input

<code>arg_pContext</code>	Pointer to the MPLS core component context.
---------------------------	---

Input/Output

<code>arg_pMplsParameters</code>	Pointer to the buffer storing the MPLS parameters returned by the function. See ix_cc_mpls_params for details.
----------------------------------	--

Output/Returns

Return Value	Returns a valid <code>ix_error</code> .
	<ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • <code>IX_CC_ERROR_NULL</code>—operation failed due to an invalid pointer.

29.4.17 `ix_cc_mpls_port_stats_query()`

This function returns MPLS statistics for a given input port.

C Syntax

```
ix_error ix_cc_mpls_port_stats_query (
    ix_uint16          arg_PortIndex,
    ix_cc_mpls_port_stats *arg_pMplsPortStatistics,
    ix_cc_mpls_context *arg_pContext);
```

Input

<code>arg_PortIndex</code>	Local port index.
<code>arg_pContext</code>	Pointer to the MPLS core component context.

Input/Output

<code>arg_pMplsPortStatistics</code>	Pointer to the buffer storing the MPLS port statistics returned by the function. See ix_cc_mpls_port_stats for details.
--------------------------------------	---

Output/Returns

Return Value	Returns a valid <code>ix_error</code> .
	<ul style="list-style-type: none"> • <code>IX_SUCCESS</code>—the operation succeeded. • <code>IX_CC_ERROR_NULL</code>—operation failed due to an invalid pointer.

A

AAL	ATM Adaptation Layer - The ATM standards layer that allows multiple applications to have data converted to and from an ATM cell. A protocol used that translates higher layer services into the size and format of an ATM cell.
AAL5	ATM Adaptation Layer 5 - AAL functionality to support variable bit rate, delay-tolerant connection-oriented data traffic.
ACE	Acronym for <i>Active Computing Element</i> .
active computing element	(ACE) A logical entity that represents a specific packet-processing activity in the IXA SDK 2.0. IXP1200 applications use ACEs to process packets. The ACE Programming Framework in the IXA SDK 2.0 is now replaced by microblocks and core components in the IXA SDK 3.0
API	Acronym for <i>application programming interface</i> .
application programming interface	(API) A set of routines, classes, methods, structures, and/or functions used to write applications.
ATM	Asynchronous Transfer Mode - A transfer mode in which information is organized into cells. It is asynchronous in the sense that the recurrence of cells containing information from an individual user is not necessarily periodic.

B

big endian	A compiler term specifying that, for multibyte values, the most significant byte is first. See also <i>little endian</i> , <i>network byte order</i> .
byte order	The way a system stores numeric data, with the most or least significant byte first. Most significant byte first, or <i>big endian</i> byte order, is also known as <i>network byte order</i> . See also <i>endianness</i> .

C

CBR	Constant Bit Rate - an ATM service class.
CBS	Committed Burst Size - an IP QoS traffic contract parameter/metric.
CIR	Committed Information Rate - an IP QoS traffic contract parameter/metric.
CLP	Cell Loss Priority - an ATM QoS traffic contract parameter/metric.

content addressable memory	(CAM) This is a hardware feature where a content match is performed to get an index to associated information.
context pipeline	A software pipeline where in different functions are performed on different microengines as time progresses and the packet context is passed between the functions or microengines. Each microengine constitutes a context pipe-stage and cascading two or more context pipe-stages constitutes a context pipeline. The context pipeline get it's name from the fact that it is the context that moves through the pipeline.
control plane	The abstraction for a functional area of an application that controls and configures the data plane and handles exception packets, distinguished from the <i>data processing plane</i> . Control plane activities are typically performed by <i>code modules</i> within the <i>IXA application</i> . Compare <i>management plane</i> , whose activities are usually outside the IXA application, in a <i>host</i> application.
core component	A packet-processing entity that configures its microblock, initializes and maintains common data structures that may be updated by other applications, and provides exception as well as control message handlers to process packets/messages sent by the microblock.
core component infrastructure	The core component infrastructure includes a number of APIs to support the creation and setup of core components.
CRC	Cyclic Redundancy Check - A mathematically computed numerical value transmitted with packet data to ensure the integrity of packet data transmitted between endpoints.
critical section	A critical section is section of code in which only one microengine thread has exclusive modification privileges for a global resource (such as a memory location) at any one time. The IXP2400 uses inter-thread signaling to implement critical sections across microEngines.
CSR	Acronym for <i>control status register</i> .

D

Decap	Decapsulation - Removing one or more protocol headers from a packet.
DiffServ	Differentiated Service. A means of classifying IP packets into “classes”, based on the DiffServ codepoint (DSCP) in the packet's IP header.
Dispatch Loop	A Dispatch Loop combines microblocks running on a microengine thread and implements the data flow between them.
DRR	Deficit Round Robin. A QoS queue-scheduling algorithm.
DSCP	DiffServ Code Point. A 6-bit field in the IPv4 header.

E

EE	Acronym for execution engine.
Encap	Encapsulation - Adding one ore more protocol headers to a packet.

endian, endianness	A compiler term for the <i>byte order</i> of multibyte values. See <i>big endian</i> and <i>little endian</i> .
Ethernet	A local area network (LAN) technology designed for interconnecting networking nodes over a shared medium, as specified in standard IEEE 802.3. Also typically used to refer to the Layer 2 networking protocol as specified in standard IEEE 802.2.

F

Fast Path	The data path where in the packet is completely processed on the MEv2 microengines without any intervention from the XScale Core.
Folding	A software technique used by threads running on the same ME, to optimize read/modify/writes in a critical section. The technique uses the CAM and strict thread ordering enforced via inter-thread signaling to fold the read/modify/write into a single read, multiple modifies and one or more writes depending on the cache eviction policy.
Functional Pipeline	A software pipeline where in the context remains with an microengine while different functions are performed on the packet as time progresses. The microengine execution time is divided into “n” pipe-stages and each pipe-stage performs a different functions. The Functional pipeline get it's name from the fact that it is the function that moves through the pipeline.

G

GFR	Guaranteed Frame Rate - an ATM service class.
-----	---

H

Head of Line Blocking	A term used to describe a situation where the transmit operation on a group of ports is blocked by a single port at the head of the transmit queue. This scenario typically occurs when the port at the head of the transmit queue is blocked because of flow control issues and the remaining ports on the queue have data pending but need to wait for this port to finish its transmit operation.
HEC	Header Error Check - An 8-bit field within an ATM header that is generated by a sender, and checked by a receiver, to determine the validity of an ATM header.

I

Intel® Internet Exchange Architecture	(IXA) A new approach to designing networking and telecommunications equipment based on reprogrammable silicon and open interfaces. Manufacturers of networking and communications equipment can use components from the IX-based product portfolio for designing new, more intelligent network systems.
---------------------------------------	---

intrinsic	A C function-like interface that implements a chip-specific hardware feature, not otherwise supported by the C language. Direct use of intrinsics results in non-portable code.
IP	An acronym for <i>internet protocol</i> , a standard network protocol. See also <i>TCP/IP</i> .
IPv4	Internet Protocol Version 4.
IPv6	Internet Protocol Version 6.
IXP	Acronym for <i>Intel</i> ® <i>Internet Exchange Processor</i> , and a current instance of this processor.
IXP2400, IXP 2800	Internet eXchange network processors. The IXP2400 has 8 microengines targeted at OC-48 POS line rates and the IXP2800 has 16 microengine targeted at OC-192 POS line rates.

J

K

L

L2	Layer 2.
L3	Layer 3.
LLCSNAP	Logical Link Control/Sub Network Access Protocol - Data link layer packet encapsulation headers that identify a protocol, as well as client and control information. Refer to IEEE standards 802.3 with 802.2.
LPM	Longest Prefix Match - algorithm IP routers apply to an IP packet destination address to determine the packet's egress port, and hence forward the packet out the egress port.
little endian	A compiler term specifying that, for multibyte values, the least significant byte is first. See also <i>big endian</i> .
longword	A 32-bit word; 4 bytes long.

M

MAC	Medium Access Control - A protocol layer responsible for providing access to a shared communications medium. Also Medium Access Controller - The device used to interface with the physical layer medium.
ME	An acronym for <i>microengine</i> .
MEv2	A microengine specific to the IXP2xxx network processor family.
Microblock	A discrete unit of IXP2xxx code written in microcode or MicroC that is written to the guidelines specified in the IXA Software Framework. Microblocks conform to one of three different types—source, transform or sink. Typically, a microblock has an XScale component that is used to configure and manage the microblock.

Microblock Group	One or more microblocks that have been combined into a thread executable on a microengine. Typically all threads on the microengine will execute the same microblock group, but it is not required. Furthermore, a typical use instantiates the same microblock group on several microengines.
Microengine	One of many (8 for IXP2400, 16 for IXP2800) programmable, specialized processors.
Mixed Pipeline	A software pipeline where some microengines run a single function (context pipe-stage) and others run multiple functions (functional pipeline)
MPKT	M Packet - An IXP2xxx media bus interface data transfer unit that can be configured to be 64, 128, or 256 bytes in length.
MEv2	Microengine version 2, which are the microengines used for the IXP2400 and IXP2800 network processors.
microcode	Hardware-specific machine code. A <i>code module</i> written in microcode can run only on the processor it is written for.
mutual exclusion, mutex	Mutual exclusion is used to guard the critical sections accessed by threads.

N

nrtVBR	Non-Real Time Variable Bit Rate - an ATM service class.
network byte order	The system of storing numeric data with the most significant byte first. See also <i>big endian</i> , <i>endianness</i> .
network services application	General descriptive term for the kind of application you build with the <i>IXA SDK</i> .

O

OAM	Operations Administration and Maintenance - A group of network management functions that provide network fault indication, performance information, and data and diagnosis functions within an ATM network. Also the type of ATM cell payload used to carry such information.
OC-12, OC-48c	Optical Carrier (SONET) - Level (e.g. Level = 3, 12, 48, 192). Often used to specify data rates; the base level rate is 51.84 Mbps (OC-1); each level thereafter operates at a multiple of the base level rate (thus, OC-3 runs at 155.52 Mbps, OC-12 runs at 622.08 Mbps, etc).
OS	Acronym for <i>operating system</i> .
OSSL	Acronym for <i>operating system services library</i> .
Operating System Services Library	(OSSL) An OS abstraction API used within the <i>IXA SDK</i> to achieve portability.

optimized data plane
libraries

A library of low-level macros and functions for microEngine program development. The purpose of this library is to provide a layer of portability, so programmers can write code that will run on IXP1200, IXP2400, IXP2800 and future IXP chips.

P

payload

The part of a packet that carries data, as opposed to those parts that carry information about the packet.

Q

quadword

A 64-bit word; 8 bytes long.

QoS

Acronym for *quality of service*.

quality of service

A networking term that specifies a guaranteed throughput level. (*QoS*)

R

Resource Manager

A programming interface between Intel XScale® core applications and the microcode running on the microengines for the IXP2400 and IXP2800 network processors.

RR

Round Robin. A scheduling algorithm in which entities/queues are services/scheduled in a consistent serial manner.

rtVBR

Real Time Variable Bit Rate - an ATM service class.

Rx

Receive.

S

SAR

Segmentation And Reassembly - The process of transforming frames-to-cells and cells-to-frames.

SDE

Acronym for *software development environment*.

SDK

Acronym for *software development kit*.

semaphore

Semaphores are the primary means for providing thread synchronization.

sink microblock

A function or macro that disposes of a packet, that is, either enqueues it within the IXP or sends it to an external interface.

slow path

The execution path of the packets that require exceptional handling. This may be error packets or packets that need to be handled differently than the normal case. In this case, it will take longer to process because they will be handled by a general-purpose processor (Intel XScale® core in our case). See also *Fast Path*.

software pipeline

The MEv2 employs a software pipeline model in the fast path processing of packets. There are three different types of pipelines—*Context Pipeline*, *Functional Pipeline*, and *Mixed Pipeline*.

source microblock	A function or macro that obtains a packet, that is, either dequeues it within the IXP or gets it from an external interface.
SP	Acronym for <i>scheduling policy</i> .
stdmac	Acronym for <i>standard macros</i> . Assembly macros that are microengine-specific, for instruction simplification.

T

TCP	An acronym for <i>transmission control protocol</i> , a standard network protocol in which transmission status can be confirmed. Establishes a point-to-point connection, in contrast to <i>UDP</i> which is connectionless. See also <i>TCP/IP</i> .
TCP/IP	A standard network protocol, using <i>TCP</i> over <i>IP</i> . See <i>TCP</i> and <i>IP</i> .
TM4.1	Traffic Management version 4.1 - An ATM specification for managing/controlling traffic congestion within an ATM network by the actions of buffering, adjusting transmission rates, and policing VCs.
TOS	Type of Service. Refers to an 8-bit field in the IPv4 header.
Tx	Transmit.
thread	A thread is an independent task, which can be processed in parallel with other tasks.
transform microblock	A function or macro that parses, analyzes, classifies, or modifies a packet.

U

UBR	Unspecified Bit Rate - an ATM service class.
UPC	Usage Parameter Control - VC traffic contract characteristics, that permit ATM network nodes to monitor, control, and police the traffic within the ATM network.

V

VBR	Variable Bit Rate - an ATM service class.
VC	Virtual Connection or Virtual Channel - A communications channel between ATM systems nodes that provides for the sequential transport of ATM cells.
VCi	Virtual Connection Identifier - A 16-bit numerical tag within an ATM cell header that identifies a virtual channel over which the cell is to travel.
VPI	Virtual Path Identifier - An 8-bit numerical tag within an ATM cell header that indicates the virtual path over which the cell should be routed.
VPN	Virtual Private Network.

VPORT Virtual Port - A field accompanying a MPKT that identifies the port (and possibly line card) to/from which the MPKT payload is sent/received.

W

WAN Wide Area Network - A network that spans a large geographical area relative to a LAN (Local Area Network). A WAN typically experiences greater traffic delays (due to distance between nodes and greater network congestion) and packet loss (due to switches dropping packets).

WRR Acronym for *weighted round robin*.

X

XScale core The ARM architecture core processor in the IXP2400 and IXP2800 network processors.

Y

Z

1	Introduction	27
1.1	About this Document	27
1.2	Audience	27
1.3	Other Sources of Information	28
2	Core Components Overview	29
2.1	Overview	29
2.1.1	Functional and Data Flow	31
2.1.2	Functional APIs Design Concept	32
2.2	APIs for Dynamic Property Updates	33
2.2.1	Dynamic Properties and Clients	33
2.2.2	Property Updates API and Data Structures	33
2.2.2.1	Properties Data Structure	33
2.2.3	Property ID	38
2.2.4	Generic Prototype of Property API	40
2.2.4.1	<code>ix_cc_<name of the cc>_set_property</code>	40
2.2.4.2	Current Behavior of Property API	40
2.3	Handler Registration	41
2.3.1	Advantages	41
2.3.2	Core Component Handler Registration	41
2.3.3	Handler Configuration	42
2.3.4	API	42
2.3.4.1	<code>ix_cc_add_packet_handler_list()</code>	42
2.3.4.2	<code>ix_cci_cc_add_message_handler_list()</code>	44
2.3.5	Support for the IXA Portability Framework and Core Components Infrastructure	45
2.3.5.1	Usage of <code>init()</code> and <code>fini()</code> Function	46
2.3.6	OS Independence of Core Components	46
2.4	High-Level Overview of the Core Components	48
2.4.1	System Application	48
2.4.2	POS RX	48
2.4.3	CSIX RX	48
2.4.4	Ethernet RX	49
2.4.5	CSIX TX	49
2.4.6	ATM/POS TX	49
2.4.7	Ethernet TX	50
2.4.7.1	Ethernet ARP Module	50
2.4.8	Queue Manager (QM)	50
2.4.9	Queue Manager for DiffServ	51
2.4.10	Scheduler	51
2.4.11	Scheduler for DiffServ	52
2.4.12	IPv4 Forwarder	52
2.4.13	IPv6 Forwarder	52
2.4.14	IPv6 to IPv4 Tunneling	53
2.4.15	Six-Tuple Classifier	53
2.4.16	Three Color Meter	54
2.4.17	Weighted Random Early Detection (WRED)	54
2.4.18	DSCP Classifier	54
2.4.19	Route Table Manager	54
2.4.19.1	Next Hop Database	55
2.4.19.2	TCAM	55

2.4.19.3	Software LPM	55
2.4.20	Route Table Manager for IPv6.....	56
2.4.21	L2 Table Manager.....	56
2.4.22	Message Helper and Support Library	56
2.4.23	Stack Driver	57
2.4.24	SoftSAR Core Components.....	57
2.4.24.1	SAR Core Components	57
2.4.24.2	ATM RX Core Components	57
2.4.24.3	ATM TX Core Components.....	58
2.4.24.4	TM4.1 Core Components.....	58
2.4.25	MPLS Forwarder Core Component	58
3	System Application	61
3.1	Start and Shut Down the System Application.....	61
3.1.1	ix_sa_create()	62
3.1.2	_ix_sa_entry()	62
3.1.3	ix_sa_shutdown()	63
3.2	Loading Microcode and Starting Microengines.....	63
3.2.1	ix_sa_start_microengines()	64
3.3	User Initialization and Shutdown Hooks	64
3.3.1	ix_sa_init_hook_first()	65
3.3.2	ix_sa_init_hook_pre_ee()	65
3.3.3	ix_sa_init_hook_ee()	66
3.3.4	ix_sa_init_hook_pre_me()	66
3.3.5	ix_sa_init_hook_last()	67
3.3.6	ix_sa_shutdown_hook_first()	67
3.3.7	ix_sa_shutdown_hook_post_me()	68
3.3.8	ix_sa_shutdown_hook_post_ee()	68
3.3.9	ix_sa_shutdown_hook_ee()	69
3.3.10	ix_sa_shutdown_hook_last()	69
Receive		
4	POS RX API.....	73
4.1	Core Component Infrastructure API	73
4.1.1	ix_cc_pos_rx_init()	73
4.1.2	ix_cc_pos_rx_fini()	75
4.1.3	ix_cc_pos_rx_msg_handler()	76
4.1.4	ix_cc_pos_rx_pkt_handler()	77
4.2	Messaging API.....	78
4.2.1	ix_cc_pos_rx_async_get_statistics_info()	78
4.2.1.1	ix_s_cc_pos_rx_statistics_info_context	79
4.2.1.2	ix_cc_pos_rx_cb_get_statistics_info	80
4.2.2	ix_cc_pos_rx_async_get_interface_state()	80
4.2.2.1	ix_cc_pos_rx_cb_get_interface_state	81
4.2.2.2	ix_s_cc_pos_rx_if_state_context	81
4.2.2.3	ix_cc_pos_rx_if_state	82
4.3	Library API.....	82
4.3.1	ix_cc_pos_rx_get_statistics_info()	82
4.3.1.1	ix_e_cc_pos_rx_statistics_info	83
4.3.1.2	ix_s_cc_statistics_info_data	84

4.3.2	<code>ix_cc_pos_rx_get_interface_state()</code>	84
5	CSIX RX	85
5.4	Core Component Infrastructure API	85
5.4.1	<code>ix_cc_csix_rx_init()</code>	85
5.4.2	<code>ix_cc_csix_rx_fini()</code>	86
5.4.3	<code>ix_cc_csix_rx_msg_handler()</code>	87
5.5	Messaging API	88
5.5.1	<code>ix_cc_csix_rx_async_get_statistics_info()</code>	88
5.5.1.1	<code>ix_s_cc_csix_rx_statistics_info_context</code>	89
5.5.1.2	<code>ix_cc_csix_rx_cb_get_statistics_info</code>	90
5.6	Library API	90
5.6.1	<code>ix_cc_csix_rx_get_statistics_info()</code>	90
5.6.1.1	<code>ix_e_cc_csix_rx_statistics_info</code> Enumeration	91
5.6.1.2	<code>ix_cc_statistics_info_data</code>	92
6	Ethernet RX	93
6.1	Core Component Infrastructure API	93
6.1.1	<code>ix_cc_eth_rx_init()</code>	94
6.1.2	<code>ix_cc_eth_rx_fini()</code>	95
6.1.3	<code>ix_cc_eth_rx_msg_handler()</code>	95
6.1.4	<code>ix_cc_eth_rx_high_priority_pkt_handler()</code>	96
6.1.5	<code>ix_cc_eth_rx_low_priority_pkt_handler()</code>	97
6.2	Messaging API	98
6.2.1	<code>ix_cc_eth_rx_async_get_statistics_info()</code>	98
6.2.1.1	<code>ix_s_cc_eth_rx_statistics_info_context</code>	99
6.2.1.2	<code>ix_cc_eth_rx_cb_get_statistics_info</code>	100
6.2.2	<code>ix_cc_eth_rx_async_get_interface_state()</code>	101
6.2.2.1	<code>ix_s_cc_eth_rx_if_state_context</code>	101
6.2.2.2	<code>ix_cc_eth_rx_cb_get_interface_state</code>	103
6.2.3	<code>ix_cc_eth_rx_async_add_mac_addr()</code>	103
6.2.3.1	<code>ix_cc_eth_rx_cb_mac_addr_op</code>	104
6.2.4	<code>ix_cc_eth_rx_async_delete_mac_addr()</code>	105
6.2.4.1	<code>ix_cc_eth_rx_cb_mac_addr_op</code>	105
6.2.5	<code>ix_cc_eth_rx_async_lookup_port()</code>	106
6.2.5.1	<code>ix_cc_eth_rx_cb_lookup_port</code>	107
6.3	Library API	107
6.3.1	<code>ix_cc_eth_rx_get_interface_state()</code>	107
6.3.1.1	<code>enum ix_e_cc_eth_rx_if_state</code> Enumeration	108
6.3.2	<code>ix_cc_eth_rx_set_property()</code>	109
6.3.3	<code>ix_cc_eth_rx_add_mac_addr()</code>	109
6.3.4	<code>ix_cc_eth_rx_del_mac_addr()</code>	110
6.3.5	<code>ix_cc_eth_rx_lookup_port()</code>	111
Transmit		
7	CSIX TX	115
7.1	Core Component Infrastructure API	115
7.1.1	<code>ix_cc_csix_tx_init()</code>	115
7.1.2	<code>ix_cc_csix_tx_fini()</code>	116
7.1.3	<code>ix_cc_csix_tx_msg_handler()</code>	117

7.2	Messaging API.....	118
7.2.1	ix_cc_csix_tx_async_get_statistics_info()	118
7.2.1.1	ix_s_cc_csix_tx_statistics_info_context	119
7.2.1.2	ix_e_cc_csix_tx_statistics_info Enumeration.....	119
7.2.1.3	ix_cc_csix_tx_cb_get_statistics_info	120
7.3	Library API.....	120
7.3.1	ix_cc_csix_tx_get_statistics_info()	120
7.3.1.1	ix_s_cc_statistics_info_data	121
8	ATM/POS TX.....	123
8.1	Core Component Infrastructure API	123
8.1.1	ix_cc_atm_pos_tx_init()	123
8.1.2	ix_cc_atm_pos_tx_fini()	124
8.1.3	ix_cc_atm_pos_tx_msg_handler()	125
8.1.4	ix_cc_atm_pos_tx_property_msg_handler()	126
8.2	Messaging API.....	126
8.2.1	ix_cc_atm_pos_tx_async_get_statistics_info()	127
8.2.1.1	ix_s_cc_atm_pos_tx_statistics_info_context	127
8.2.1.2	ix_e_cc_atm_pos_tx_statistics_info Enumeration	128
8.2.1.3	ix_cc_atm_pos_tx_cb_get_statistics_info	129
8.2.2	ix_cc_atm_pos_tx_async_get_interface_state()	130
8.2.2.1	ix_s_cc_atm_pos_tx_if_state_context	130
8.2.2.2	ix_cc_atm_pos_tx_cb_get_interface_state Callback.....	132
8.2.2.3	ix_e_cc_atm_pos_tx_if_state	132
8.3	Library API.....	132
8.3.1	ix_cc_atm_pos_tx_get_statistics_info()	133
8.3.1.1	ix_s_cc_atm_pos_tx_statistics_info_data	134
8.3.2	ix_cc_atm_pos_tx_get_interface_state()	134
8.3.3	ix_cc_atm_pos_tx_set_property()	135
9	Ethernet TX	137
9.1	Data Structures.....	137
9.1.1	ix_cc_eth_tx_next_hop_info	138
9.1.2	ix_ether_addr	138
9.1.3	ix_cc_eth_tx_if_state	139
9.2	Core Component Infrastructure API	139
9.2.1	ix_cc_eth_tx_init()	139
9.2.2	ix_cc_eth_tx_fini()	140
9.2.3	ix_cc_eth_tx_msg_handler()	141
9.2.4	ix_cc_eth_tx_property_msg_handler()	142
9.2.5	ix_cc_eth_tx_pkt_handler()	143
9.3	Messaging API.....	144
9.3.1	ix_cc_eth_tx_async_get_statistics_info()	145
9.3.1.1	ix_cc_eth_tx_statistics_info_context	145
9.3.1.2	ix_cc_eth_tx_statistics_info	146
9.3.1.3	ix_cc_eth_tx_cb_get_statistics_info()	148
9.3.2	ix_cc_eth_tx_async_get_interface_state()	149
9.3.2.1	ix_cc_eth_tx_if_state_context	150
9.3.2.2	ix_cc_eth_tx_cb_get_interface_state()	150
9.3.3	ix_cc_eth_tx_async_create_arp_entry()	151
9.3.3.1	ix_cc_eth_tx_cb_create_arp_entry()	152

9.3.4	<code>ix_cc_eth_tx_async_add_arp_entry()</code>	152
9.3.4.1	<code>ix_cc_eth_tx_cb_add_arp_entry()</code>	153
9.3.5	<code>ix_cc_eth_tx_async_del_arp_entry()</code>	154
9.3.5.1	<code>ix_cc_eth_tx_cb_del_arp_entry()</code>	154
9.3.6	<code>ix_cc_eth_tx_async_purge_arp_cache()</code>	155
9.3.7	<code>ix_cc_eth_tx_async_dump_arp_cache()</code>	155
9.4	Library API	156
9.4.1	<code>ix_cc_eth_tx_get_statistics_info()</code>	156
9.4.1.1	<code>ix_s_cc_statistics_info_data</code>	157
9.4.2	<code>ix_cc_eth_tx_get_interface_state()</code>	158
9.4.3	<code>ix_cc_eth_tx_create_arp_entry()</code>	158
9.4.4	<code>ix_cc_eth_tx_add_arp_entry()</code>	159
9.4.5	<code>ix_cc_eth_tx_del_arp_entry()</code>	160
9.4.6	<code>ix_cc_eth_tx_purge_arp_cache()</code>	160
9.4.7	<code>ix_cc_eth_tx_dump_arp_cache()</code>	161
9.4.8	<code>ix_cc_eth_tx_set_property()</code>	161
10	Ethernet ARP Module	163
10.1	Error Types	164
10.2	Library API	165
10.2.1	<code>ix_cc_arp_init()</code>	165
10.2.2	<code>ix_cc_arp_fini()</code>	166
10.2.3	<code>ix_cc_arp_create_entry()</code>	166
10.2.4	<code>ix_cc_arp_add_entry()</code>	167
10.2.5	<code>ix_cc_arp_update_entry()</code>	168
10.2.6	<code>ix_cc_arp_del_entry()</code>	169
10.2.7	<code>ix_cc_arp_purge()</code>	169
10.2.8	<code>ix_cc_arp_dump()</code>	170
10.2.9	<code>ix_cc_arp_resolve_l2_addr()</code>	170
10.2.10	<code>ix_cc_arp_process_arp_pkts()</code>	171
10.2.11	<code>ix_cc_arp_create_gratuitous_arp()</code>	173
Queue Manager		
11	Queue Manager	177
11.1	Core Component Infrastructure API	177
11.1.1	<code>ix_cc_qm_init()</code>	177
11.1.2	<code>ix_cc_qm_fini()</code>	178
11.1.3	<code>ix_cc_qm_pkt_handler()</code>	179
11.1.4	Sending Packets	179
11.1.5	<code>ix_cc_qm_msg_handler()</code>	180
11.2	Messaging API	181
11.2.1	<code>ix_cc_qm_async_get_packet_count()</code>	181
11.2.1.1	<code>ix_cc_qm_cb_pkt_count</code>	181
11.3	Library API	182
11.3.1	<code>ix_cc_qm_get_packet_count()</code>	182
12	Egress Queue Manager (DiffServ)	183
Scheduler		

13	Scheduler	187
13.1	Core Component Infrastructure API	188
13.1.1	ix_cc_scheduler_init()	188
13.1.2	ix_cc_scheduler_fini()	189
14	Egress Scheduler (DiffServ)	191
Forwarder		
15	IPv4 Forwarder	195
15.1	Data Structures, Types and Macros	195
15.1.1	IX_CC_RTMV4_DUMP_ROUTE_SIZE	196
15.1.2	IX_CC_RTMV4_DUMP_NEXT_HOP_SIZE	196
15.1.3	ix_cc_rtmv4_nhid	196
15.1.4	Reserved Next Hop Ids	196
15.1.4.1	IX_CC_RTMV4_NHID_NO_ROUTE	196
15.1.5	ix_cc_rtmv4_next_hop_info	196
15.1.6	ix_cc_ipv4_dump_data	196
15.1.7	ix_cc_ipv4_stats_data	197
15.2	Core Component Infrastructure API	198
15.2.1	ix_cc_ipv4_init()	198
15.2.2	ix_cc_ipv4_fini()	200
15.2.3	ix_cc_ipv4_msg_handler()	201
15.2.4	ix_cc_ipv4_microblock_high_priority_pkt_handler()	202
15.2.5	ix_cc_ipv4_microblock_low_priority_pkt_handler()	203
15.2.6	ix_cc_ipv4_stackdrv_pkt_handler()	204
15.2.7	ix_cc_ipv4_common_pkt_handler()	204
15.3	Message Helper API	205
15.3.1	ix_cc_ipv4_async_add_route()	206
15.3.1.1	ix_cc_ipv4_cb_route_op	207
15.3.2	ix_cc_ipv4_async_delete_route()	207
15.3.2.1	ix_cc_ipv4_cb_route_op	208
15.3.3	ix_cc_ipv4_async_update_route()	209
15.3.3.1	ix_cc_ipv4_cb_route_op	210
15.3.4	ix_cc_ipv4_async_lookup_route()	210
15.3.4.1	ix_cc_ipv4_cb_lookup_route	211
15.3.5	ix_cc_ipv4_async_purge_routes()	211
15.3.6	ix_cc_ipv4_async_dump_routes()	212
15.3.6.1	ix_cc_ipv4_cb_dump_data	212
15.3.7	ix_cc_ipv4_async_add_next_hop()	213
15.3.7.1	ix_cc_ipv4_cb_route_op	213
15.3.8	ix_cc_ipv4_async_delete_next_hop()	214
15.3.8.1	ix_cc_ipv4_cb_route_op	215
15.3.9	ix_cc_ipv4_async_update_next_hop()	215
15.3.9.1	ix_cc_ipv4_cb_route_op	216
15.3.10	ix_cc_ipv4_async_get_next_hop()	216
15.3.10.1	ix_cc_ipv4_cb_get_next_hop	217
15.3.11	ix_cc_ipv4_async_dump_next_hops()	217
15.3.11.1	ix_cc_ipv4_cb_dump_data	218
15.3.12	ix_cc_ipv4_async_purge_rtm()	219

15.3.13	<code>ix_cc_ipv4_async_set_mtu()</code>	219
15.3.14	<code>ix_cc_ipv4_cb_route_op</code>	220
15.3.15	<code>ix_cc_ipv4_async_set_flags()</code>	220
15.3.15.1	<code>ix_cc_ipv4_cb_route_op</code>	221
15.3.16	<code>ix_cc_ipv4_async_get_sleep_time()</code>	222
15.3.16.1	<code>ix_cc_ipv4_cb_get_sleep_time</code>	222
15.3.17	<code>ix_cc_ipv4_async_set_sleep_time()</code>	223
15.3.18	<code>ix_cc_ipv4_async_get_queue_depth()</code>	223
15.3.18.1	<code>x_cc_ipv4_cb_get_queue_depth</code>	224
15.3.19	<code>ix_cc_ipv4_async_get_packets_to_drain()</code>	224
15.3.19.1	<code>ix_cc_ipv4_cb_get_packets_to_drain</code>	225
15.3.20	<code>ix_cc_ipv4_async_set_packets_to_drain()</code>	225
15.3.21	<code>ix_cc_ipv4_async_get_statistics()</code>	226
15.3.21.1	<code>ix_cc_ipv4_cb_get_statistics</code>	227
15.4	Library API	227
15.4.1	<code>ix_cc_ipv4_add_route()</code>	229
15.4.2	<code>ix_cc_ipv4_delete_route()</code>	230
15.4.3	<code>ix_cc_ipv4_update_route()</code>	230
15.4.4	<code>ix_cc_ipv4_lookup_route()</code>	231
15.4.5	<code>ix_cc_ipv4_purge_routes()</code>	232
15.4.6	<code>ix_cc_ipv4_dump_routes()</code>	232
15.4.7	<code>ix_cc_ipv4_add_next_hop()</code>	233
15.4.8	<code>ix_cc_ipv4_delete_next_hop()</code>	234
15.4.9	<code>ix_cc_ipv4_update_next_hop()</code>	235
15.4.10	<code>ix_cc_ipv4_get_next_hop()</code>	235
15.4.11	<code>ix_cc_ipv4_dump_next_hops()</code>	236
15.4.12	<code>ix_cc_ipv4_purge_rtm()</code>	238
15.4.13	<code>ix_cc_ipv4_set_mtu()</code>	238
15.4.13.1	<code>ix_cc_ipv4_set_flags()</code>	239
15.4.14	<code>ix_cc_ipv4_get_rtm_handle()</code>	240
15.4.15	<code>ix_cc_ipv4_get_sleep_time()</code>	240
15.4.16	<code>ix_cc_ipv4_set_sleep_time()</code>	241
15.4.17	<code>ix_cc_ipv4_get_queue_depth()</code>	241
15.4.17.1	<code>ix_cc_ipv4_get_packets_to_drain()</code>	242
15.4.18	<code>ix_cc_ipv4_set_packets_to_drain()</code>	242
15.4.19	<code>ix_cc_ipv4_get_statistics()</code>	243
15.4.20	<code>ix_cc_ipv4_set_property()</code>	244
16	IPv6 Forwarder	245
16.1	Data Structures, Types and Macros	245
16.1.1	<code>IX_CC_RTMV6_DUMP_ROUTE_SIZE</code>	245
16.1.2	<code>ix_cc_rtmv6_nhid</code>	246
16.1.3	Reserved Next Hop Ids	246
16.1.4	<code>ix_cc_rtmv6_next_hop_info</code>	246
16.1.5	<code>ix_cc_ipv6_dump_data</code>	246
16.1.6	<code>ix_cc_in_ipv6_stats_data</code>	247
16.1.7	<code>ix_cc_out_ipv6_stats_data</code>	247
16.1.8	<code>ix_cc_ipv6_stats_data</code>	248
16.1.9	<code>ix_cc_ipv6_icmp_err_type</code>	248
16.1.10	<code>ix_cc_ipv6_icmp_err_code</code>	248

16.2	Core Component Infrastructure API	249
16.2.1	ix_cc_ipv6_init()	249
16.2.2	ix_cc_ipv6_fini()	251
16.2.3	ix_cc_ipv6_msg_handler()	252
16.2.4	ix_cc_ipv6_microblock_high_priority_pkt_handler()	254
16.2.5	ix_cc_ipv6_microblock_low_priority_pkt_handler()	254
16.2.6	ix_cc_ipv6_stackdrv_pkt_handler()	255
16.2.7	ix_cc_ipv6_common_pkt_handler()	256
16.3	Message Helper API.....	257
16.3.1	ix_cc_ipv6_async_add_prefix()	258
16.3.1.1	ix_cc_ipv6_cb_route_op	258
16.3.2	ix_cc_ipv6_async_delete_prefix()	259
16.3.3	ix_cc_ipv6_async_update_prefix()	260
16.3.4	ix_cc_ipv6_async_lookup_prefix()	260
16.3.4.1	ix_cc_ipv6_cb_lookup_route	261
16.3.5	ix_cc_ipv6_async_purge_prefixes()	262
16.3.6	ix_cc_ipv6_async_dump_prefixes()	262
16.3.6.1	ix_cc_ipv6_cb_dump_data	263
16.3.7	ix_cc_ipv6_async_add_next_hop()	263
16.3.8	ix_cc_ipv6_async_delete_next_hop()	264
16.3.9	ix_cc_ipv6_async_update_next_hop()	265
16.3.10	ix_cc_ipv6_async_get_next_hop()	266
16.3.10.1	ix_cc_ipv6_cb_get_next_hop	266
16.3.11	ix_cc_ipv6_async_dump_next_hops()	267
16.3.12	ix_cc_ipv6_async_purge_rtm()	267
16.3.13	ix_cc_ipv6_async_set_mtu()	268
16.3.14	ix_cc_ipv6_async_set_flags()	268
16.3.15	ix_cc_ipv6_async_get_rate_limit_time()	269
16.3.15.1	ix_cc_ipv6_cb_get_rate_limit_time	270
16.3.16	ix_cc_ipv6_async_set_rate_limit_time()	270
16.3.17	ix_cc_ipv6_async_get_queue_depth()	271
16.3.17.1	ix_cc_ipv6_cb_get_queue_depth	271
16.3.18	ix_cc_ipv6_async_get_statistics()	272
16.3.18.1	ix_cc_ipv6_cb_get_statistics	272
16.3.19	ix_cc_ipv6_async_perform_addr_resolution()	273
16.3.19.1	ix_cc_ipv6_cb_route_op	273
16.3.20	ix_cc_ipv6_async_add_neighbor()	274
16.3.21	ix_cc_ipv6_async_del_neighbor()	275
16.3.22	ix_cc_ipv6_sync_add_neighbor()	276
16.3.23	ix_cc_ipv6_sync_del_neighbor()	277
16.3.24	ix_cc_ipv6_async_send_icmp_error()	277
16.3.25	ix_cc_ipv6_async_send_icmp_info_message()	279
16.4	Library API.....	280
16.4.1	ix_cc_ipv6_add_prefix()	281
16.4.2	ix_cc_ipv6_delete_prefix()	282
16.4.3	ix_cc_ipv6_update_prefix()	282
16.4.4	ix_cc_ipv6_lookup_prefix()	283
16.4.5	ix_cc_ipv6_purge_prefixes()	284
16.4.6	ix_cc_ipv6_dump_prefixes()	284
16.4.7	ix_cc_ipv6_add_next_hop()	285

16.4.8	<code>ix_cc_ipv6_delete_next_hop()</code>	285
16.4.9	<code>ix_cc_ipv6_update_next_hop()</code>	286
16.4.10	<code>ix_cc_ipv6_get_next_hop()</code>	287
16.4.11	<code>ix_cc_ipv6_dump_next_hops()</code>	288
16.4.12	<code>ix_cc_ipv6_purge_rtm()</code>	288
16.4.13	<code>ix_cc_ipv6_set_mtu()</code>	289
16.4.14	<code>ix_cc_ipv6_set_flags()</code>	289
16.4.15	<code>ix_cc_ipv6_get_rate_limit_time()</code>	290
16.4.16	<code>ix_cc_ipv6_set_rate_limit_time()</code>	291
16.4.17	<code>ix_cc_ipv6_get_queue_depth()</code>	291
16.4.18	<code>ix_cc_ipv6_get_statistics()</code>	292
16.4.19	<code>ix_cc_ipv6_set_property()</code>	292
16.4.20	<code>ix_cc_ipv6_perform_addr_resolution()</code>	293
16.4.21	<code>ix_cc_ipv6_add_neighbor()</code>	295
16.4.22	<code>ix_cc_ipv6_del_neighbor()</code>	296
16.4.23	<code>ix_cc_ipv6_send_icmp_error()</code>	296
16.4.24	<code>ix_cc_ipv6_send_icmp_info_message()</code>	297
17	IPv6 to IPv4 Tunneling	299
17.1	Data Structures, Types and Macros	299
17.1.1	<code>ix_cc_v6v4_tunnel_handle</code>	300
17.1.2	<code>ix_cc_v6v4_end_tunnel_config</code>	300
17.1.3	<code>ix_cc_v6v4_end_tunnel_info</code>	300
17.1.4	End Tunnel Flags	301
17.1.5	<code>ix_cc_v6v4_ingress_source_entry</code>	301
17.1.6	<code>ix_cc_v6v4_interface_id</code>	301
17.1.7	<code>ix_cc_v6v4_start_tunnel_config</code>	301
17.1.7.1	<code>IX_CC_V6V4_PATH_MTU_INTERFACE</code>	302
17.1.7.2	<code>IX_CC_V6V4_BROADCAST_INTERFACE</code>	302
17.1.8	<code>ix_cc_v6v4_start_tunnel_info</code>	303
17.1.9	Start Tunnel Flags	303
17.1.10	<code>ix_cc_v6v4_statistics</code>	303
17.1.11	Option Values	304
17.2	Core Component Infrastructure API	304
17.2.1	<code>ix_cc_v6v4_init()</code>	304
17.2.2	<code>ix_cc_v6v4_fini()</code>	305
17.2.3	<code>ix_cc_v6v4_msg_handler()</code>	306
17.2.4	<code>ix_cc_v6v4_microblock_pkt_handler()</code>	308
17.2.5	<code>ix_cc_v6v4_ipv6_pkt_handler()</code>	308
17.2.6	<code>ix_cc_v6v4_ipv4_pkt_handler()</code>	309
17.3	Message Helper API	310
17.3.1	<code>ix_cc_v6v4_async_add_end_tunnel()</code>	311
17.3.1.1	<code>ix_cc_v6v4_cb_add_tunnel</code>	312
17.3.2	<code>ix_cc_v6v4_async_delete_end_tunnel()</code>	312
17.3.2.1	<code>ix_cc_v6v4_cb</code>	313
17.3.3	<code>ix_cc_v6v4_async_set_decap_tos_option()</code>	314
17.3.4	<code>ix_cc_v6v4_async_set_src_validation()</code>	315
17.3.5	<code>ix_cc_v6v4_async_get_end_tunnel()</code>	315
17.3.5.1	<code>ix_cc_v6v4_cb_end_tunnel</code>	316
17.3.6	<code>ix_cc_v6v4_async_add_allowed_source()</code>	317

17.3.7	<code>ix_cc_v6v4_async_delete_allowed_source()</code>	317
17.3.8	<code>ix_cc_v6v4_async_get_allowed_sources()</code>	318
17.3.8.1	<code>ix_cc_v6v4_cb_get_sources</code>	319
17.3.9	<code>ix_cc_v6v4_async_dump_end_tunnels()</code>	319
17.3.9.1	<code>ix_cc_v6v4_cb_dump_end_tunnels</code>	320
17.3.10	<code>ix_cc_v6v4_async_clear_allowed_sources()</code>	320
17.3.11	<code>ix_cc_v6v4_async_add_start_tunnel()</code>	321
17.3.12	<code>ix_cc_v6v4_async_delete_start_tunnel()</code>	323
17.3.13	<code>ix_cc_v6v4_async_set_ttl()</code>	323
17.3.14	<code>ix_cc_v6v4_async_set_encap_tos_option()</code>	324
17.3.15	<code>ix_cc_v6v4_async_set_tos()</code>	325
17.3.16	<code>ix_cc_v6v4_async_set_mtu()</code>	327
17.3.17	<code>ix_cc_v6v4_async_set_subnet_broadcast()</code>	327
17.3.18	<code>ix_cc_v6v4_async_get_start_tunnel()</code>	329
17.3.18.1	<code>ix_cc_v6v4_cb_start_tunnel</code>	329
17.3.19	<code>ix_cc_v6v4_async_dump_start_tunnels()</code>	330
17.3.19.1	<code>ix_cc_v6v4_cb_dump_start_tunnels</code>	331
17.3.20	<code>ix_cc_v6v4_async_get_statistics()</code>	331
17.3.20.1	<code>ix_cc_v6v4_cb_statistics</code>	332
17.4	Library API	332
17.4.1	<code>ix_cc_v6v4_add_end_tunnel()</code>	333
17.4.2	<code>ix_cc_v6v4_delete_end_tunnel()</code>	334
17.4.3	<code>ix_cc_v6v4_set_decap_tos_option()</code>	334
17.4.4	<code>ix_cc_v6v4_set_src_validation()</code>	335
17.4.5	<code>ix_cc_v6v4_get_end_tunnel()</code>	337
17.4.6	<code>ix_cc_v6v4_add_allowed_source()</code>	338
17.4.7	<code>ix_cc_v6v4_delete_allowed_source()</code>	338
17.4.8	<code>ix_cc_v6v4_clear_allowed_sources()</code>	339
17.4.9	<code>ix_cc_v6v4_get_allowed_sources()</code>	340
17.4.10	<code>ix_cc_v6v4_dump_end_tunnels()</code>	341
17.4.11	<code>ix_cc_v6v4_add_start_tunnel()</code>	342
17.4.12	<code>ix_cc_v6v4_delete_start_tunnel()</code>	342
17.4.13	<code>ix_cc_v6v4_set_ttl()</code>	343
17.4.14	<code>ix_cc_v6v4_set_encap_tos_option()</code>	343
17.4.15	<code>ix_cc_v6v4_set_tos()</code>	344
17.4.16	<code>ix_cc_v6v4_set_mtu()</code>	346
17.4.17	<code>ix_cc_v6v4_set_subnet_broadcast()</code>	347
17.4.18	<code>ix_cc_v6v4_get_start_tunnel()</code>	348
17.4.19	<code>ix_cc_v6v4_dump_start_tunnels()</code>	348
17.4.20	<code>ix_cc_v6v4_get_statistics()</code>	349
17.4.21	<code>ix_cc_v6v4_set_property()</code>	350
18	NAT-PT Translation	351
18.1	Data Structures, Types and Macros	351
18.1.1	<code>ix_cc_natpt_config_params</code>	352
18.1.2	<code>ix_cc_natpt_v6addr</code>	352
18.1.3	<code>ix_cc_natpt_v4addr</code>	352
18.1.4	<code>ix_cc_natpt_static_v6v4map</code>	352
18.1.5	<code>ix_cc_natpt_v4v6portmap</code>	353
18.1.6	<code>ix_cc_natpt_naptmode</code>	353

18.1.7	<code>ix_cc_natpt_session</code>	353
18.1.8	Translation CC Specific Error Codes	354
18.2	Core Component Infrastructure API	355
18.2.1	<code>ix_cc_natpt_init()</code>	355
18.2.2	<code>ix_cc_natpt_fini()</code>	356
18.2.3	<code>ix_cc_natpt_msg_handler()</code>	357
18.2.4	<code>ix_cc_natpt_microblock_pkt_handler()</code>	358
18.3	Message Helper API.....	359
18.3.1	<code>ix_cc_natpt_async_set_config_parameters()</code>	360
18.3.1.1	<code>ix_cc_natpt_cb()</code>	361
18.3.2	<code>ix_cc_natpt_async_get_config_parameters()</code>	361
18.3.2.1	<code>ix_cc_natpt_cb_get_config_parameters</code>	362
18.3.3	<code>ix_cc_natpt_async_add_static_mapping()</code>	362
18.3.4	<code>ix_cc_natpt_async_remove_static_mapping()</code>	363
18.3.5	<code>ix_cc_natpt_async_get_static_mapping()</code>	364
18.3.5.1	<code>ix_cc_natpt_cb_get_static_mapping</code>	365
18.3.6	<code>ix_cc_natpt_async_add_v4addr_pool()</code>	365
18.3.7	<code>ix_cc_natpt_async_remove_v4addr_pool()</code>	366
18.3.8	<code>ix_cc_natpt_async_get_v4addr_pool()</code>	367
18.3.8.1	<code>ix_cc_natpt_cb_get_v4addr_pool</code>	367
18.3.9	<code>ix_cc_natpt_async_set_natpt_mode()</code>	368
18.3.10	<code>ix_cc_natpt_async_add_v4v6port_mapping()</code>	369
18.3.11	<code>ix_cc_natpt_async_remove_v4v6port_mapping()</code>	369
18.3.12	<code>ix_cc_natpt_async_get_v4v6port_mapping()</code>	370
18.3.12.1	<code>ix_cc_natpt_cb_get_v4v6port_mapping</code>	371
18.3.13	<code>ix_cc_natpt_async_get_active_sessions()</code>	371
18.3.13.1	<code>ix_cc_natpt_cb_get_active_sessions</code>	372
18.4	Library API.....	373
18.4.1	<code>ix_cc_natpt_set_config_parameters()</code>	373
18.4.2	<code>ix_cc_natpt_get_config_parameters()</code>	375
18.4.3	<code>ix_cc_natpt_add_static_mapping()</code>	375
18.4.4	<code>ix_cc_natpt_remove_static_mapping()</code>	376
18.4.5	<code>ix_cc_natpt_get_static_mapping()</code>	377
18.4.6	<code>ix_cc_natpt_add_v4addr_pool()</code>	378
18.4.7	<code>ix_cc_natpt_remove_v4addr_pool()</code>	379
18.4.8	<code>ix_cc_natpt_get_v4addr_pool()</code>	380
18.4.9	<code>ix_cc_natpt_set_natpt_mode()</code>	380
18.4.10	<code>ix_cc_natpt_add_v4v6port_mapping()</code>	381
18.4.11	<code>ix_cc_natpt_remove_v4v6port_mapping()</code>	382
18.4.12	<code>ix_cc_natpt_get_v4v6port_mapping()</code>	383
18.4.13	<code>ix_cc_natpt_get_active_sessions()</code>	383

DiffServ Components

19	Six-Tuple Exact Match Classifier	387
19.1	Data Structures.....	387
19.1.1	Classification Pattern Data Type	387
19.1.1.1	<code>ix_s_cc_classifier_6t_pattern</code>	388
19.1.2	Classification Result Data Type.....	388
19.1.2.1	<code>ix_cc_classifier_6t_table_type</code> Enumeration.....	388

19.1.2.2	ix_cc_classifier_6t_qos_output Enumeration.....	388
19.1.2.3	ix_s_cc_classifier_6t_qos_result	389
19.1.2.4	ix_s_cc_classifier_6t_fwd_result	389
19.1.2.5	ix_cc_classifier_6t_result	389
19.1.3	Statistics Data Type.....	390
19.1.3.1	ix_cc_classifier_6t_statistics	390
19.2	Core Component Infrastructure API	390
19.2.1	ix_cc_classifier_6t_init()	391
19.2.2	ix_cc_classifier_6t_fini()	392
19.2.3	ix_cc_classifier_6t_pkt_handler()	393
19.2.4	ix_cc_classifier_6t_msg_handler()	394
19.3	Message Helper API.....	396
19.3.1	ix_cc_classifier_6t_async_add_entry()	396
19.3.1.1	ix_cc_classifier_6t_cb_add_entry	397
19.3.2	ix_cc_classifier_6t_async_remove_entry()	398
19.3.2.1	ix_cc_classifier_6t_cb_remove_entry	399
19.3.3	ix_cc_classifier_6t_async_update_entry()	399
19.3.3.1	ix_cc_classifier_6t_cb_update_entry	400
19.3.4	ix_cc_classifier_6t_async_search_entry()	401
19.3.4.1	ix_cc_classifier_6t_cb_search_entry	402
19.3.5	ix_cc_classifier_6t_async_get_statistics()	403
19.3.5.1	ix_cc_classifier_6t_cb_get_statistics	404
19.3.6	ix_cc_classifier_6t_async_get_statistics_def()	405
19.3.6.1	ix_cc_classifier_6t_cb_get_statistics_def	405
19.4	Library API.....	406
19.4.1	ix_cc_classifier_6t_add_entry()	407
19.4.2	ix_cc_classifier_6t_remove_entry()	408
19.4.3	ix_cc_classifier_6t_update_entry()	409
19.4.4	ix_cc_classifier_6t_search_entry()	410
19.4.5	ix_cc_classifier_6t_get_statistics()	411
19.4.6	ix_cc_classifier_6t_get_statistics_def()	412
20	Three Color Meter.....	413
20.1	Data Structures.....	413
20.1.1	TCM Parameters Data Type.....	413
20.1.1.1	ix_cc_tc_meter_output Enumerration.....	413
20.1.1.2	ix_cc_tc_meter_algo_type Enumeration.....	414
20.1.1.3	ix_cc_tc_meter_parameters	414
20.1.2	TCM Statistics Data Type.....	415
20.1.2.1	ix_cc_tc_meter_statistics	415
20.2	Core Component Infrastructure API	415
20.2.1	ix_cc_tc_meter_init()	415
20.2.2	ix_cc_tc_meter_fini()	416
20.2.3	ix_cc_tc_meter_pkt_handler()	417
20.2.4	ix_cc_tc_meter_msg_handler()	418
20.3	Message Helper API.....	419
20.3.1	ix_cc_tc_meter_async_add_entry()	419
20.3.1.1	ix_cc_tc_meter_cb_add_entry	420
20.3.2	ix_cc_tc_meter_async_remove_entry()	421
20.3.2.1	ix_cc_tc_meter_cb_remove_entry	422
20.3.3	ix_cc_tc_meter_async_update_entry()	422

20.3.3.1	ix_cc_tc_meter_cb_update_entry	423
20.3.4	ix_cc_tc_meter_async_get_statistics()	424
20.3.4.1	ix_cc_tc_meter_cb_get_statistics	425
20.4	Library API	426
20.4.1	ix_cc_tc_meter_add_entry()	426
20.4.2	ix_cc_tc_meter_remove_entry()	429
20.4.3	ix_cc_tc_meter_update_entry()	430
20.4.4	ix_cc_tc_meter_get_statistics()	432
21	Weighted Random Early Detection	435
21.1	Data Structures	435
21.1.1	WRED Parameters Data Types	435
21.1.1.1	ix_s_cc_red_instance	435
21.1.1.2	ix_s_cc_wred_parameters	436
21.1.2	WRED Statistics Data Type	436
21.1.2.1	ix_cc_wred_statistics	436
21.2	Core Component Infrastructure API	437
21.2.1	ix_cc_wred_init()	437
21.2.2	ix_cc_wred_fini()	439
21.2.3	ix_cc_wred_pkt_handler()	440
21.2.4	ix_cc_wred_msg_handler()	440
21.3	Message Helper API	442
21.3.1	ix_cc_wred_async_add_entry()	442
21.3.1.1	ix_cc_wred_cb_add_entry	443
21.3.2	ix_cc_wred_async_remove_entry()	444
21.3.2.1	ix_cc_wred_cb_remove_entry	445
21.3.3	ix_cc_wred_async_update_entry()	445
21.3.3.1	ix_cc_wred_cb_update_entry	446
21.3.4	ix_cc_wred_async_get_statistics()	447
21.3.4.1	ix_cc_wred_cb_get_statistics	448
21.4	Library API	449
21.4.1	ix_cc_wred_add_entry()	449
21.4.2	ix_cc_wred_remove_entry()	452
21.4.3	ix_cc_wred_update_entry()	452
21.4.4	ix_cc_wred_get_statistics()	455
22	DSCP Classifier	457
22.1	Data Structures in Functional APIs	457
22.1.1	Classification Result Data Type	458
22.1.1.1	ix_s_cc_classifier_dscp_result	458
22.1.2	Statistics Data Type	458
22.1.2.1	ix_s_cc_classifier_dscp_statistics	458
22.2	Core Component Infrastructure API	458
22.2.1	ix_cc_classifier_dscp_init()	459
22.2.2	ix_cc_classifier_dscp_fini()	460
22.2.3	ix_cc_classifier_dscp_pkt_handler()	461
22.2.4	ix_cc_classifier_dscp_msg_handler()	462
22.3	Message Helper API	464
22.3.1	ix_cc_classifier_dscp_async_add_if_config()	464
22.3.1.1	ix_cc_classifier_dscp_cb_add_if_config	465

22.3.2	<code>ix_cc_classifier_dscp_async_remove_if_config()</code>	466
22.3.2.1	<code>ix_cc_classifier_7t_cb_remove_if_config</code>	466
22.3.3	<code>ix_cc_classifier_dscp_async_update_if_config()</code>	467
22.3.3.1	<code>ix_cc_classifier_dscp_cb_update_if_config</code>	468
22.3.4	<code>ix_cc_classifier_dscp_async_get_if_config()</code>	469
22.3.4.1	<code>ix_cc_classifier_dscp_cb_get_if_config</code>	470
22.3.5	<code>ix_cc_classifier_dscp_async_set_def_rules()</code>	470
22.3.5.1	<code>ix_cc_classifier_dscp_cb_set_def_rules</code>	471
22.3.6	<code>ix_cc_classifier_dscp_async_get_def_rules()</code>	472
22.3.6.1	<code>ix_cc_classifier_dscp_cb_get_def_rules</code>	472
22.3.7	<code>ix_cc_classifier_dscp_async_get_statistics()</code>	473
22.3.7.1	<code>ix_cc_classifier_dscp_cb_get_statistics</code>	474
22.4	Library API	475
22.4.1	<code>ix_cc_classifier_dscp_add_if_config()</code>	475
22.4.2	<code>ix_cc_classifier_dscp_remove_if_config()</code>	477
22.4.3	<code>ix_cc_classifier_dscp_update_if_config()</code>	477
22.4.4	<code>ix_cc_classifier_dscp_set_def_rules()</code>	478
22.4.5	<code>ix_cc_classifier_dscp_get_def_rules()</code>	479
22.4.6	<code>ix_cc_classifier_dscp_get_if_config()</code>	480
22.4.7	<code>ix_cc_classifier_dscp_get_statistics()</code>	481

Support Libraries

23	Route Table Manager	485
23.1	Data Structures, Types, and Macros	485
23.1.1	Data Structures, Types	485
23.1.1.1	<code>ix_cc_rtmv4</code>	486
23.1.1.2	<code>ix_cc_rtmv4_nhid</code>	486
23.1.1.3	<code>ix_cc_rtmv4_next_hop_info</code>	486
23.1.1.4	<code>ix_cc_rtmv4_symbols</code>	487
23.1.1.5	<code>ix_cc_rtmv4_statistics</code>	488
23.1.1.6	<code>ix_cc_rtmv4_lkup_type</code>	488
23.1.1.7	<code>ix_cc_rtmv4_mem_type</code>	488
23.1.1.8	<code>ix_cc_rtmv4_config</code>	488
23.1.2	Macros	489
23.1.2.1	<code>IX_CC_RTMV4_DUMP_ROUTE_SIZE()</code>	489
23.1.2.2	<code>IX_CC_RTMV4_DUMP_NEXT_HOP_SIZE</code>	490
23.1.2.3	<code>IX_CC_RTMV4_NHID_NO_ROUTE</code>	490
23.1.2.4	<code>IX_CC_RTMV4_L2INDEX_NO_ROUTE</code>	490
23.2	Core Component Infrastructure API	492
23.2.1	<code>ix_cc_rtmv4_init()</code>	493
23.2.2	<code>ix_cc_rtmv4_fini()</code>	494
23.2.3	<code>ix_cc_rtmv4_add_next_hop()</code>	495
23.2.4	<code>ix_cc_rtmv4_delete_next_hop()</code>	496
23.2.5	<code>ix_cc_rtmv4_update_next_hop()</code>	497
23.2.6	<code>ix_cc_rtmv4_get_next_hop()</code>	498
23.2.7	<code>ix_cc_rtmv4_set_mtu()</code>	499
23.2.8	<code>ix_cc_rtmv4_set_flags()</code>	500
23.2.9	<code>ix_cc_rtmv4_add_route()</code>	501
23.2.10	<code>ix_cc_rtmv4_update_route()</code>	502
23.2.11	<code>ix_cc_rtmv4_delete_route()</code>	504

23.2.12	<code>ix_cc_rtmv4_get_route</code>	505
23.2.13	<code>ix_cc_rtmv4_lookup()</code>	506
23.2.14	<code>ix_cc_rtmv4_dump_next_hops()</code>	507
23.2.15	<code>ix_cc_rtmv4_dump_routes()</code>	508
23.2.16	<code>ix_cc_rtmv4_purge()</code>	509
23.2.17	<code>ix_cc_rtmv4_purge_routes()</code>	510
23.2.18	<code>ix_cc_rtmv4_get_symbols()</code>	511
23.2.19	<code>ix_cc_rtmv4_get_statistics()</code>	512
24	Route Table Manager for IPV6	513
24.1	Data Structures, Types, and Macros	513
24.1.1	<code>ix_cc_rtmv6</code>	514
24.1.2	<code>ix_cc_rtmv6_nhidx</code>	514
24.1.3	<code>ix_cc_rtmv6_next_hop_info</code>	514
24.1.4	<code>ix_cc_rtmv6_symbols</code>	515
24.1.5	<code>ix_cc_rtmv6_statistics</code>	515
24.1.6	<code>ix_cc_rtmv6_lkup_type</code>	515
24.1.7	<code>ix_cc_rtmv6_mem_type</code>	516
24.1.8	<code>IX_CC_RTMV6_DUMP_ROUTE_SIZE</code>	516
24.1.9	<code>IX_CC_RTMV6_DUMP_NEXT_HOP_SIZE</code>	516
24.1.10	<code>IX_CC_RTMV6_NHID_NO_ROUTE</code>	517
24.2	Core Component Infrastructure API	517
24.2.1	<code>ix_cc_rtmv6_init()</code>	518
24.2.2	<code>ix_cc_rtmv6_fini()</code>	519
24.2.3	<code>ix_cc_rtmv6_add_next_hop()</code>	519
24.2.4	<code>ix_cc_rtmv6_delete_next_hop()</code>	520
24.2.5	<code>ix_cc_rtmv6_update_next_hop()</code>	521
24.2.6	<code>ix_cc_rtmv6_get_next_hop()</code>	522
24.2.7	<code>ix_cc_rtmv6_set_mtu()</code>	522
24.2.8	<code>ix_cc_rtmv6_set_flags()</code>	523
24.2.9	<code>ix_cc_rtmv6_add_route()</code>	524
24.2.10	<code>ix_cc_rtmv6_delete_route()</code>	525
24.2.11	<code>ix_cc_rtmv6_update_route()</code>	526
24.2.12	<code>ix_cc_rtmv6_lookup()</code>	527
24.2.13	<code>ix_cc_rtmv6_dump_next_hops()</code>	528
24.2.14	<code>ix_cc_rtmv6_dump_routes()</code>	529
24.2.15	<code>ix_cc_rtmv6_purge()</code>	529
24.2.16	<code>ix_cc_rtmv6_purge_routes()</code>	530
24.2.17	<code>ix_cc_rtmv6_get_symbols()</code>	531
24.2.18	<code>ix_cc_rtmv6_get_statistics()</code>	532
25	L2 Table Manager	533
25.1	Data Structures, Types and Definitions	533
25.1.1	<code>ix_s_cc_l2tm_atm_header</code>	534
25.1.2	<code>ix_s_cc_l2tm_config</code>	534
25.1.3	<code>ix_s_cc_l2tm_entry</code>	535
25.1.4	<code>ix_s_cc_l2tm_ether_header</code>	536
25.1.5	<code>ix_s_cc_l2tm_stats</code>	536
25.1.6	<code>ix_s_cc_l2tm_symbols</code>	537
25.1.7	<code>ix_u_cc_l2tm_ipaddr</code>	537

25.1.8	<code>ix_cc_l2tm</code>	538
25.1.9	<code>ix_cc_l2tm_ipaddr_type</code> Enumeration.....	538
25.1.10	<code>ix_cc_l2tm_l2addr_type</code> Enumeration.....	538
25.1.11	<code>ix_e_cc_l2tm_error</code> Enumeration.....	539
25.1.12	<code>ix_cc_l2tm_memory_type</code> Enumeration.....	539
25.1.13	<code>ix_cc_l2tm_state</code> Enumeration.....	539
25.2	Library API	540
25.2.1	<code>ix_cc_l2tm_create()</code>	540
25.2.2	<code>ix_cc_l2tm_destroy()</code>	541
25.2.3	<code>ix_cc_l2tm_init()</code>	542
25.2.4	<code>ix_cc_l2tm_fini()</code>	543
25.2.5	<code>ix_cc_l2tm_add_entry()</code>	544
25.2.6	<code>ix_cc_l2tm_update_entry()</code>	545
25.2.7	<code>ix_cc_l2tm_delete_entry()</code>	545
25.2.8	<code>ix_cc_l2tm_get_entry()</code>	546
25.2.9	<code>ix_cc_l2tm_add_l3_info()</code>	547
25.2.10	<code>ix_cc_l2tm_update_l3_info()</code>	548
25.2.11	<code>ix_cc_l2tm_assign_l2_info()</code>	549
25.2.12	<code>ix_cc_l2tm_clear_l2_info()</code>	550
25.2.13	<code>ix_cc_l2tm_get_l2_index()</code>	550
25.2.14	<code>ix_cc_l2tm_get_symbols()</code>	551
25.2.15	<code>ix_cc_l2tm_get_statistics()</code>	552
25.2.16	<code>ix_cc_l2tm_purge()</code>	552
26	Message Helper and Support Library.....	553
26.1	Message Support Library API.....	553
26.1.1	<code>ix_cc_msup_init()</code>	553
26.1.2	<code>ix_cc_msup_fini()</code>	554
26.1.3	<code>ix_cc_msup_send_msg()</code>	555
26.1.4	<code>ix_cc_msup_send_async_msg()</code>	555
26.1.5	<code>ix_cc_msup_send_bcast_msg()</code>	556
26.1.6	<code>ix_cc_msup_send_sync_msg()</code>	557
26.1.7	<code>IX_MSUP_EXTRACT_MSG()</code>	558
26.1.8	<code>ix_cc_msup_send_reply_msg()</code>	558
Stack Driver		
27	Stack Driver.....	561
27.1	Core Component Module Data Structures and Types.....	561
27.1.1	<code>ix_cc_stkdrv_packet_type</code>	561
27.1.2	<code>ix_cc_stkdrv_handler_id</code>	562
27.1.3	<code>ix_cc_stkdrv_handler_func</code>	563
27.1.4	<code>ix_cc_stkdrv_virtual_if</code>	564
27.1.5	<code>ix_cc_stkdrv_physical_if_info</code>	565
27.1.6	<code>ix_cc_stkdrv_physical_if_node</code>	566
27.1.7	<code>ix_cc_stkdrv_handler_module</code>	566
27.1.8	<code>ix_cc_stkdrv_fp_node</code>	567
27.1.9	<code>ix_cc_stkdrv_ctrl</code>	568
27.2	Stack Driver Callback Prototypes	569
27.2.1	Communication Handler Packet Processing	569

27.2.1.1	<code>ix_cc_stkdrv_packet_cb()</code>	569
27.2.2	Communication Handler Message Processing	570
27.2.2.1	<code>ix_cc_stkdrv_msg_str_cb()</code>	570
27.2.2.2	<code>ix_cc_stkdrv_msg_int_cb()</code>	571
27.2.3	Shutdown	571
27.2.3.1	<code>ix_cc_stkdrv_handler_module_fini_cb()</code>	571
27.3	Core Component API	572
27.3.1	Core Component Infrastructure API	572
27.3.1.1	<code>ix_cc_stkdrv_init()</code>	573
27.3.1.2	<code>ix_cc_stkdrv_fini()</code>	574
27.3.1.3	<code>ix_cc_stkdrv_high_priority_pkt_handler()</code>	574
27.3.1.4	<code>ix_cc_stkdrv_low_priority_pkt_handler()</code>	575
27.3.1.5	<code>ix_cc_stkdrv_pkt_to_remote_handler()</code>	576
27.3.1.6	<code>ix_cc_stkdrv_msg_handler()</code>	577
27.3.2	Core Component Infrastructure Separation	578
27.3.2.1	<code>_ix_cc_stkdrv_process_pkt()</code>	578
27.3.2.2	<code>_ix_cc_stkdrv_process_pkt_to_remote()</code>	579
27.3.3	Packet and Message Processing API	579
27.3.3.1	<code>ix_cc_stkdrv_send_packet()</code>	580
27.3.3.2	<code>ix_cc_stkdrv_send_msg_str()</code>	580
27.3.3.3	<code>ix_cc_stkdrv_send_msg_int()</code>	581
27.3.4	Properties API	582
27.3.4.1	<code>ix_cc_stkdrv_async_get_property()</code>	582
27.3.4.2	<code>ix_cc_stkdrv_async_get_num_ports()</code>	583
27.4	Packet Classifier	586
27.4.1	Packet Classifier Data Structures	586
27.4.1.1	<code>ix_cc_stkdrv_filter_type</code>	586
27.4.1.2	<code>ix_cc_stkdrv_filter_priority</code>	587
27.4.1.3	<code>ix_cc_stkdrv_ipv4_address_range</code>	587
27.4.1.4	<code>ix_cc_stkdrv_ipv6_address_range</code>	587
27.4.1.5	<code>ix_cc_stkdrv_port_range</code>	588
27.4.1.6	<code>ix_cc_stkdrv_ipv4_range_filter</code>	588
27.4.1.7	<code>ix_cc_stkdrv_ipv6_range_filter</code>	589
27.4.1.8	<code>ix_cc_stkdrv_filter</code>	589
27.4.1.9	<code>ix_cc_stkdrv_filter_index_node</code>	590
27.4.1.10	<code>ix_cc_stkdrv_filter_handle</code>	590
27.4.1.11	<code>ix_cc_stkdrv_filter_ctrl</code>	590
27.4.2	Core Component Infrastructure API	591
27.4.2.1	<code>ix_cc_stkdrv_cb_filter_ops()</code>	591
27.4.2.2	<code>ix_cc_stkdrv_init_filters()</code>	592
27.4.2.3	<code>ix_cc_stkdrv_fini_filters()</code>	592
27.4.2.4	<code>ix_cc_stkdrv_classify_pkt()</code>	594
27.4.3	Message Helper API	594
27.4.3.1	<code>ix_cc_stkdrv_async_add_filter()</code>	595
27.4.3.2	<code>ix_cc_stkdrv_async_remove_filter()</code>	595
27.4.3.3	<code>ix_cc_stkdrv_async_remove_all_filters()</code>	596
27.4.3.4	<code>ix_cc_stkdrv_async_modify_filter()</code>	596
27.4.4	Library API	597
27.4.4.1	<code>ix_cc_stkdrv_add_filter()</code>	598
27.4.4.2	<code>ix_cc_stkdrv_remove_filter()</code>	598
27.4.4.3	<code>ix_cc_stkdrv_remove_all_filters()</code>	599
27.4.4.4	<code>ix_cc_stkdrv_async_modify_filter()</code>	599

27.5	Outgoing Packet Classifier Design	600
27.5.1	Outgoing Packet Classifier Data Structures	600
27.5.1.1	ix_cc_stkdrv_og_pkt_type	601
27.5.1.2	ix_cc_stkdrv_og_filter_type	601
27.5.1.3	ix_cc_stkdrv_og_cc_type	601
27.5.1.4	ix_cc_stkdrv_port_range	602
27.5.1.5	ix_cc_stkdrv_cb_og_chk_filter	602
27.5.1.6	ix_cc_stkdrv_og_filter	602
27.5.1.7	ix_cc_stkdrv_og_comm_cc_map	603
27.5.1.8	ix_cc_stkdrv_og_filter_ctrl	603
27.5.2	Outgoing Packet Classifier Internal API Functions	604
27.5.2.1	ix_cc_stkdrv_init_og_filters()	604
27.5.2.2	ix_cc_stkdrv_fini_og_filters()	604
27.5.2.3	ix_cc_stkdrv_classify_output_id()	605
27.6	VIDD for VxWorks*	606
27.6.1	VIDD System Data Structures for VxWorks.....	606
27.6.1.1	DEV_OBJ	606
27.6.1.2	END_ERR	607
27.6.1.3	END_OBJ	608
27.6.1.4	M2_INTERFACETBL	610
27.6.2	VIDD Local Data Structures for VxWorks	611
27.6.2.1	ix_cc_stkdrv_vidd_physical_if_node	612
27.6.2.2	ix_cc_stkdrv_vidd_fp_node	612
27.6.3	ix_cc_stkdrv_vidd_ctrl	613
27.7	MUX Interface API.....	614
27.7.1	NET_FUNCS	615
27.7.2	VIDD System Function Calls	616
27.7.2.1	ix_cc_stkdrv_vidd_npt_load()	617
27.7.2.2	ix_cc_stkdrv_vidd_npt_unload()	618
27.7.2.3	ix_cc_stkdrv_vidd_npt_start()	618
27.7.2.4	ix_cc_stkdrv_vidd_npt_stop()	619
27.7.2.5	ix_cc_stkdrv_vidd_npt_ioctl()	619
27.7.2.6	ix_cc_stkdrv_vidd_npt_send()	620
27.7.2.7	ix_cc_stkdrv_vidd_npt_mCastAddrAdd()	621
27.7.2.8	ix_cc_stkdrv_vidd_npt_mCastAddrDel()	622
27.7.2.9	ix_cc_stkdrv_vidd_npt_mCastAddrGet()	622
27.7.2.10	ix_cc_stkdrv_vidd_npt_pollSend()	623
27.7.2.11	ix_cc_stkdrv_vidd_npt_pollRcv()	624
27.7.3	MUX API used by the VIDD.....	625
27.7.3.1	muxTkReceive()	625
27.7.3.2	muxError()	626
27.7.3.3	muxTxRestart()	626
27.8	VIDD for Linux*	626
27.8.1	VIDD System Data Structures for Linux	627
27.8.1.1	sk_buff	627
27.8.1.2	net_device	628
27.8.1.3	ifreq	628
27.8.1.4	ix_cc_stkdrv_vidd_physical_if_node	629
27.8.2	VIDD System API for Linux.....	630
27.8.2.1	ix_cc_stkdrv_vidd_ifd_open	630
27.8.2.2	ix_cc_stkdrv_vidd_ifd_stop	630
27.8.2.3	ix_cc_stkdrv_vidd_ifd_tx	631
27.8.2.4	ix_cc_stkdrv_vidd_ifd_set_config	631

27.8.2.5	ix_cc_stkdrv_vidd_ifd_do_ioctl	632
27.8.2.6	ix_cc_stkdrv_vidd_ifd_get_stats	632
27.8.2.7	ix_cc_stkdrv_vidd_ifd_set_multicast_list	633
27.8.2.8	ix_cc_stkdrv_vidd_ifd_init	633
27.8.3	VIDD Linux Driver Support API.....	634
27.8.3.1	ix_cc_stkdrv_vidd_init	634
27.8.3.2	ix_cc_stkdrv_vidd_fini	635
27.8.3.3	ix_cc_stkdrv_vidd_if_devinet_ioctl	635
27.8.3.4	ix_cc_stkdrv_vidd_if_dev_ioctl	636
27.8.3.5	ix_cc_stkdrv_vidd_if_up	636
27.8.3.6	ix_cc_stkdrv_vidd_if_down	637
27.8.3.7	ix_cc_stkdrv_vidd_receive_pkt	637
27.8.3.8	ix_cc_stkdrv_set_ip_mask	638
27.9	Transport Module.....	639
27.9.1	Transport Data Structures	639
27.9.1.1	ix_cc_stkdrv_tm_ctrl	639
27.9.2	Transport API.....	639
27.9.2.1	ix_cc_stkdrv_tm_receive_pkt()	639
27.9.2.2	ix_cc_stkdrv_tm_pkt_handler()	641

SoftSAR

28	SoftSAR	645
28.1	SAR Core Components	645
28.1.1	Data Structures.....	645
28.1.1.1	VC-related Data Structures.....	646
28.1.1.2	Port-Related Data Structures.....	648
28.1.2	Core Component Infrastructure API	649
28.1.2.1	ix_cc_atmsar_init()	649
28.1.2.2	ix_cc_atmsar_fini()	650
28.1.2.3	ix_cc_atmsar_msg_handler()	651
28.1.2.4	ix_cc_atmsar_pkt_handler()	652
28.1.3	Messaging API.....	653
28.1.3.1	ix_cc_atmsar_async_vc_create()	653
28.1.3.2	ix_cc_atmsar_async_vc_update()	655
28.1.3.3	ix_cc_atmsar_async_vc_remove()	656
28.1.3.4	ix_cc_atmsar_async_port_create()	658
28.1.3.5	ix_cc_atmsar_async_port_remove()	659
28.1.3.6	ix_cc_atmsar_async_get_vc_stats()	661
28.1.3.7	ix_cc_atmsar_async_get_port_stats()	662
28.1.4	Library API	664
28.1.4.1	ix_cc_atmsar_vc_create()	664
28.1.4.2	ix_cc_atmsar_vc_update()	665
28.1.4.3	ix_cc_atmsar_vc_remove()	665
28.1.4.4	ix_cc_atmsar_port_create()	666
28.1.4.5	ix_cc_atmsar_port_remove()	667
28.1.4.6	ix_cc_atmsar_get_vc_stats()	667
28.1.4.7	ix_cc_atmsar_get_port_stats()	668
28.1.5	Plug-in API.....	669
28.1.5.1	ix_cc_atmsar_plugin_get_cfg_params()	669
28.1.5.2	ix_cc_atmsar_plugin_reg_vc_service()	670
28.1.5.3	ix_cc_atmsar_plugin_reg_port_service()	671
28.1.5.4	ix_cc_atmsar_plugin_reg_vc_handle_service()	672

28.1.5.5	ix_cc_atmsar_plugin_reg_port_handle_service()	672
28.1.5.6	ix_cc_atmsar_plugin_reg_done()	673
28.2	ATM RX Core Components	674
28.2.1	Core Component Infrastructure API	674
28.2.1.1	ix_cc_atmr_x_init()	675
28.2.1.2	ix_cc_atmr_x_fini()	675
28.2.1.3	ix_cc_atmr_x_msg_handler()	676
28.2.1.4	ix_cc_atmr_x_pkt_handler()	677
28.3	ATM TX Core Components	678
28.3.1	Core Component Infrastructure API	678
28.3.1.1	ix_cc_atmt_x_init()	678
28.3.1.2	ix_cc_atmt_x_fini()	679
28.4	TM4.1 Core Components	680
28.4.1	Core Component Infrastructure API	680
28.4.1.1	ix_cc_atmtm41_init()	681
28.4.1.2	ix_cc_atmtm41_fini()	683
28.4.1.3	ix_cc_atmtm41_msg_handler()	683
28.4.1.4	ix_cc_atmtm41_pkt_handler()	684

MPLS Core Component

29	MPLS Forwarder	687
29.1	Data Structures and Types	687
29.1.1	ix_cc_mpls_fec	689
29.1.2	ix_cc_mpls_label	690
29.1.3	ix_cc_mpls_ilm	690
29.1.4	ix_cc_mpls_params	691
29.1.5	ix_cc_mpls_nhlfe	692
29.1.6	ix_cc_mpls_nhlfe_handle	693
29.1.7	ix_cc_mpls_cc_nhlfe	693
29.1.8	ix_cc_mpls_nhlfe_stats	693
29.1.9	ix_cc_mpls_cc_lsp	694
29.1.10	ix_cc_mpls_lsp_param	694
29.1.11	ix_cc_mpls_lsp	695
29.1.12	ix_cc_mpls_lsp_stats	695
29.1.13	ix_cc_mpls_port_stats	696
29.2	Core Component Infrastructure API	696
29.2.1	ix_cc_mpls_init()	696
29.2.2	ix_cc_mpls_fini()	697
29.2.3	ix_cc_mpls_msg_handler()	698
29.2.4	ix_cc_mpls_microblock_pkt_handler()	699
29.3	Message Helper API	701
29.3.1	ix_cc_mpls_cb_general()	702
29.3.2	ix_cc_mpls_async_lsp_create()	702
29.3.2.1	ix_cc_mpls_cb_lsp_create()	703
29.3.3	ix_cc_mpls_async_lsp_delete()	704
29.3.4	ix_cc_mpls_async_lsp_modify()	704
29.3.4.1	ix_cc_mpls_cb_lsp_modify()	705
29.3.5	ix_cc_mpls_async_lsp_query()	706
29.3.5.1	ix_cc_mpls_cb_lsp_query()	706
29.3.6	ix_cc_mpls_async_lsp_stats_query()	707

29.3.6.1	ix_cc_mpls_cb_lsp_stats_query()	707
29.3.7	ix_cc_mpls_async_lsp_purge()	708
29.3.8	ix_cc_mpls_async_nhlfe_create()	709
29.3.8.1	ix_cc_mpls_cb_nhlfe_create()	709
29.3.9	ix_cc_mpls_async_nhlfe_delete()	710
29.3.10	ix_cc_mpls_async_nhlfe_query()	710
29.3.10.1	ix_cc_mpls_cb_nhlfe_query()	711
29.3.11	ix_cc_mpls_async_nhlfe_stats_query()	711
29.3.11.1	ix_cc_mpls_cb_nhlfe_stats_query()	712
29.3.12	ix_cc_mpls_async_nhlfe_purge()	712
29.3.13	ix_cc_mpls_async_nhlfe_set_create()	713
29.3.13.1	ix_cc_mpls_cb_nhlfe_set_create()	714
29.3.14	ix_cc_mpls_async_nhlfe_set_delete()	714
29.3.15	ix_cc_mpls_async_nhlfe_set_modify()	715
29.3.16	ix_cc_mpls_async_nhlfe_set_query()	716
29.3.17	ix_cc_mpls_async_param_query()	716
29.3.17.1	ix_cc_mpls_cb_param_query()	717
29.3.18	ix_cc_mpls_async_port_stats_query()	717
29.3.18.1	ix_cc_mpls_cb_port_stats_query()	718
29.4	Library API	719
29.4.1	ix_cc_mpls_lsp_create()	719
29.4.2	ix_cc_mpls_lsp_delete()	721
29.4.3	ix_cc_mpls_lsp_modify()	722
29.4.4	ix_cc_mpls_lsp_query()	723
29.4.5	ix_cc_mpls_lsp_stats_query()	724
29.4.6	ix_cc_mpls_lsp_purge()	724
29.4.7	ix_cc_mpls_nhlfe_create()	725
29.4.8	ix_cc_mpls_nhlfe_delete()	726
29.4.9	ix_cc_mpls_nhlfe_query()	726
29.4.10	ix_cc_mpls_nhlfe_stats_query()	727
29.4.11	ix_cc_mpls_nhlfe_purge()	728
29.4.12	ix_cc_mpls_nhlfe_set_create()	729
29.4.13	ix_cc_mpls_nhlfe_set_delete()	730
29.4.14	ix_cc_mpls_nhlfe_set_modify()	731
29.4.15	ix_cc_mpls_nhlfe_set_query()	731
29.4.16	ix_cc_mpls_param_query()	732
29.4.17	ix_cc_mpls_port_stats_query()	733
A	Glossary	735



2-1 Software Core Components	30
29-1 MPLS Core Component Data Structures.....	688



2-1	Dynamic Properties and Clients	33
2-2	Properties Data Structure	33
2-3	Property API	40
2-4	Core Component Handler Registration API	42
2-5	Core Component Functionality Mapped to Different Framework Layers	47
3-1	Starting and Shutting Down System Application	61
3-2	Loading Microcode and Starting Engines	63
3-3	User Initialization and Shutdown Hooks	64
4-1	POS RX Core Component Infrastructure API	73
4-2	POS RX Messaging API	78
4-3	POS RX Library API	82
5-1	Core Components of the CSIX API	85
5-2	Messaging API Function Calls of the CSIX RX Core Component	88
5-3	Library API Function Calls of the CSIX RX Core Component	90
6-1	Ethernet RX Core Component Functions	93
6-2	Ethernet Interface RX Messaging Functions	98
6-3	Ethernet Interface RX Library Functions	107
7-1	CSIX TX Core Component Infrastructure API	115
7-2	Messaging API for the CSIX TX Core Component	118
7-3	CSIX Library API	120
8-1	ATM/POS Interface TX Core Component Infrastructure Functions	123
8-2	ATM/POS Interface TX Messaging Functions	126
8-3	ATM/POS Interface Tx Library Functions	132
9-1	Ethernet TX Core Component Data Structures	137
9-2	Ethernet TX Core Component Infrastructure API	139
9-3	Ethernet TX Messaging Functions	144
9-4	Ethernet TX Library API	156
10-1	Ethernet ARP Library API	165
11-1	Queue Manager Core Component Infrastructure API	177
11-2	Queue Manager Core Component Messaging API	181
11-3	Queue Manager Core Component Library API	182
13-1	Scheduler Core Component Infrastructure API	188
14-1	New Patching Symbols in Scheduler (DiffServ) Core Component	191
15-1	IPv4 Forwarder Data Structures, Types, and Macros	195
15-2	IPv4 Forwarder Core Component Infrastructure API	198
15-3	IPv4 Forwarder Core Component Messages	201
15-4	IPv4 Forwarder Message Helper API	205
15-5	IPv4 Forwarder Library API	227
16-1	IPv6 Forwarder Data Structures, Types and Macros	245
16-2	IPv6 Forwarder Core Component Infrastructure API	249
16-3	IPv6 Forwarder Messages	252
16-4	IPv6 Forwarder Message Helper API	257
16-5	IPv6 Forwarder Library API	280
17-1	IPv6-IPv4 Tunneling Data Structures, Types and Macros	299
17-2	IPv6 to IPv4 Tunnelling Core Component Infrastructure AP	304
17-3	Message Types for the Tunneling Core Component	307
17-4	IPv6 to IPv4 Tunnelling Core Component Message Helper API	310
17-5	IPv6 to IPv4 Tunnelling Library API	332
18-1	Data Structures, Types and Macros in Translation Core Components	351
18-2	Translation-specific Error Codes	354

18-3	Translation Core Component Infrastructure API.....	355
18-4	Message Types for the Translation Core Component.....	357
18-5	Message Helper API in the Translation Core Component.....	359
18-6	Library API in the Translation Core Component.....	373
19-1	Data Structures for Configuring Exact-match Rules.....	387
19-2	6-tuple Classifier Core Components Infrastructure API.....	390
19-3	Microblock and Core Component Mapping Rules.....	393
19-4	6-tuple Core Component Supported Message Types.....	395
19-5	6-tuple Core Component Message Helper API.....	396
19-6	6-tuple Core Component Library API.....	406
20-1	SRTCM Core Component Data Structures.....	413
20-2	SRTCM Core Component Infrastructure APIs.....	415
20-3	SRTCM Message Types supported.....	418
20-4	SRTCM Message Helper APIs.....	419
20-5	SRTCM Library API.....	426
20-6	SRAM Entry Field Values Set by the ix_cc_tc_meter_add API.....	427
20-7	SRAM Entry Field Values Set by the ix_cc_tc_meter_update API.....	431
21-1	WRED Core Component Data Structures for Message Helper and Library APIs.....	435
21-2	WRED Core Component Infrastructure API.....	437
21-3	WRED Empty Entry Pattern.....	438
21-4	Supported Messages (ix_cc_wred_msg_handler).....	441
21-5	WRED Message Helper API.....	442
21-6	WRED Library API.....	449
21-7	WRED Table Entry Fields Set by ix_cc_wred_add_entry.....	450
21-8	WRED Table Entry Fields Set by ix_cc_wred_update_entry.....	453
22-1	Data Structures Defined by the DSCP Classifier Core Component.....	457
22-2	DSCP Classifier Core Component Infrastructure API.....	458
22-3	Messages Supported by DSCP Classifier.....	463
22-4	DSCP Classifier Message Helper API.....	464
22-5	DSCP Classifier Library API.....	475
23-1	Route Table Manager Data Structures and Types.....	485
23-2	Route Table Manager Macros.....	489
23-3	Route Table Manager Core Component Infrastructure API.....	492
24-1	RTMv6 Data Structures, Types and Macros.....	513
24-2	RTMv6 Core Component Infrastructure API.....	517
25-1	Data Structures, Types, Definitions and Enumerations in L2TM.....	533
25-2	L2 Table Manager Library API.....	540
26-1	Message Support Library API.....	553
27-1	Stack Driver Core Component Data Structures and Types.....	561
27-2	Stack Driver Callback Prototypes.....	569
27-3	Stack Driver Core Component Infrastructure API.....	572
27-4	Stack Driver Core Component Infrastructure Separation API.....	578
27-5	Stack Driver Packet and Message Processing API.....	579
27-6	Stack Driver Properties API.....	582
27-7	Stack Driver Packet Classifier Data Structures.....	586
27-8	Stack Driver Core Component Infrastructure API.....	591
27-9	Stack Driver Message Helper API.....	594
27-10	Stack Driver Library API.....	597
27-11	Stack Driver Outgoing Packet Classifier Data Structures.....	600
27-12	Stack Driver Outgoing Packet Classifier Internal API.....	604



27-13 Stack Driver VID D System Data Structures	606
27-14 <code>END_ERR</code> Error Codes	607
27-15 Stack Driver VID D Local Data Structures	611
27-16 Required Functions for the Stack Driver MUX Interface	614
27-17 VID D System API	616
27-18 IOCTL Commands	619
27-19 VID D MUX API	625
27-20 VID D System Data Structures for Linux	627
27-21 VID D System API for Linux	630
27-22 VID D Linux Driver Support API	634
27-23 Transport Module Data Structures	639
27-24 Transport Module External API	639
28-1 SAR Data Structures and Data Type Definitions	645
28-2 SAR Core Component Infrastructure API	649
28-3 SAR Messaging API	653
28-4 SAR Library API	664
28-5 SAR Control Plug-in API	669
28-6 Core Component Infrastructure API	674
28-7 Message Type defined in ATM RX Core Components	676
28-8 ATM TX Core Component Infrastructure API	678
28-9 TM4.1 Core Component supported Core Component Infrastructure API	680
29-1 MPLS Forwarder Core Component Data Types and Structures	689
29-2 MPLS Core Component Infrastructure API	696
29-3 Messages Supported by MPLS Forwarder Core Components	699
29-4 MPLS Forwarder Core Component Message Helper API	701
29-5 MPLS Forwarder Core Component Library API	719

