



Intel® Internet Exchange Architecture Portability Framework

Reference Manual

November 2003



INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL® PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT. Intel products are not intended for use in medical, life saving, life sustaining applications. Intel may make changes to specifications and product descriptions at any time, without notice.

Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them.

The Intel® Internet Exchange Architecture Software Development Kit (Intel® IXA SDK) may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

This document and the software described in it are furnished under license and may only be used or copied in accordance with the terms of the license. The information in this document is furnished for informational use only, is subject to change without notice, and should not be construed as a commitment by Intel Corporation. Intel Corporation assumes no responsibility or liability for any errors or inaccuracies that may appear in this document or any software that may be provided in association with this document. Except as permitted by such license, no part of this document may be reproduced, stored in a retrieval system, or transmitted in any form or by any means without the express written consent of Intel Corporation.

This document contains information on products in the design phase of development. The information here is subject to change without notice. Do not finalize a design with this information.

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 order 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 web site at <http://www.intel.com>.

Copyright © Intel Corporation, 2003.

Intel is a registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

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, Screamline, 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 brands and names may be claimed as the property of others.

Contents

1	Introduction	15
1.1	About this Document	15
1.2	Audience	15
1.3	IXA Portability Framework Overview	15
1.4	In This Manual	16
1.5	Other Sources of Information	16
2	Dispatch Loop	19
2.1	Dispatch Loop Variables	19
2.1.1	Microengine Assembler Dispatch Variables	19
2.1.2	Microengine C Loop Data Structure	21
2.2	Dispatch Loop Interface	23
2.2.1	Dispatch Loop API Functions for Meta Data	26
2.2.1.1	dl_buf_init[]	26
2.2.1.2	dl_buf_alloc[]	26
2.2.1.3	dl_buf_free[]	27
2.2.1.4	dl_buf_get_desc[]	27
2.2.1.5	dl_buf_get_data[]	28
2.2.1.6	dl_buf_get_data_from_meta[]	28
2.2.1.7	dl_meta_init_cache[]	29
2.2.1.8	dl_meta_flush_cache[]	30
2.2.1.9	dl_meta_get_hw_next[]	31
2.2.1.10	dl_meta_set_hw_next[]	31
2.2.1.11	dl_meta_load_cache[]	31
2.2.1.12	dl_meta_get_buffer_next[]	32
2.2.1.13	dl_meta_set_buffer_next[]	33
2.2.1.14	dl_meta_get_offset[]	33
2.2.1.15	dl_meta_set_offset[]	34
2.2.1.16	dl_meta_get_free_list[]	34
2.2.1.17	dl_meta_set_free_list[]	35
2.2.1.18	dl_meta_get_rx_stat[]	35
2.2.1.19	dl_meta_set_rx_stat[]	36
2.2.1.20	dl_meta_get_buffer_size[]	36
2.2.1.21	dl_meta_set_buffer_size[]	37
2.2.1.22	dl_meta_get_input_port[]	37
2.2.1.23	dl_meta_set_input_port[]	38
2.2.1.24	dl_meta_get_packet_size[]	38
2.2.1.25	dl_meta_set_packet_size[]	39
2.2.1.26	dl_meta_get_nexthop_id[]	39
2.2.1.27	dl_meta_set_nexthop_id[]	40
2.2.1.28	dl_meta_get_output_port[]	40
2.2.1.29	dl_meta_set_output_port[]	41
2.2.1.30	dl_meta_get_fabric_port[]	41
2.2.1.31	dl_meta_set_fabric_port[]	42
2.2.1.32	dl_meta_get_flow_id[]	42
2.2.1.33	dl_meta_set_flow_id[]	43
2.2.1.34	dl_meta_get_class_id[]	43
2.2.1.35	dl_meta_set_class_id[]	44
2.2.1.36	dl_buf_set_SOP[]	44

2.2.1.37	dl_buf_set_EOP[]	45
2.2.1.38	dl_buf_get_cell_count[]	45
2.2.1.39	dl_buf_set_cell_count[]	45
2.2.1.40	dl_set_exception[]	46
2.2.1.41	dl_meta_get_nexthop_id_type[]	46
2.2.1.42	dl_meta_set_nexthop_id_type[]	47
2.2.2	Dispatch Loop API Functions for Extended Meta Data	47
2.2.2.1	dl_meta_parent_get_ref_cnt[]	48
2.2.2.2	dl_meta_child_get_child_offset[]	48
2.2.2.3	dl_meta_child_set_child_offset[]	49
2.2.2.4	dl_meta_child_get_child_buffer_size[]	49
2.2.2.5	dl_meta_child_set_child_buffer_size[]	49
2.2.2.6	dl_meta_child_get_child_freelist_id[]	50
2.2.2.7	dl_meta_child_set_child_freelist_id[]	50
2.2.2.8	dl_meta_child_get_parent_offset[]	51
2.2.2.9	dl_meta_child_set_parent_offset[]	51
2.2.2.10	dl_meta_child_get_parent_buffer_size[]	52
2.2.2.11	dl_meta_child_set_parent_buffer_size[]	52
2.2.2.12	dl_meta_child_get_header_type[]	53
2.2.2.13	dl_meta_child_set_header_type[]	53
2.2.2.14	dl_meta_child_get_parent_free_list[]	54
2.2.2.15	dl_meta_child_set_parent_free_list[]	54
2.2.2.16	dl_meta_child_get_rx_stat[]	55
2.2.2.17	dl_meta_child_set_rx_stat[]	55
2.2.2.18	dl_meta_child_get_packet_size[]	56
2.2.2.19	dl_meta_child_set_packet_size[]	56
2.2.2.20	dl_meta_child_get_output_port[]	57
2.2.2.21	dl_meta_child_set_output_port[]	57
2.2.2.22	dl_meta_child_get_input_port[]	57
2.2.2.23	dl_meta_child_set_input_port[]	58
2.2.2.24	dl_meta_child_get_nexthop_id[]	58
2.2.2.25	dl_meta_child_set_nexthop_id[]	59
2.2.2.26	dl_meta_child_get_fabric_port[]	59
2.2.2.27	dl_meta_child_set_fabric_port[]	60
2.2.2.28	dl_meta_child_get_nexthop_id_type[]	60
2.2.2.29	dl_meta_child_set_nexthop_id_type[]	61
2.2.2.30	dl_meta_child_get_flow_id[]	61
2.2.2.31	dl_meta_child_set_flow_id[]	61
2.2.2.32	dl_meta_child_get_color[]	62
2.2.2.33	dl_meta_child_set_color[]	62
2.2.2.34	dl_meta_child_get_class_id[]	63
2.2.2.35	dl_meta_child_set_class_id[]	63
2.2.2.36	dl_meta_child_get_parent_buffer_id[]	64
2.2.2.37	dl_meta_child_set_parent_buffer_id[]	64
2.2.2.38	dl_meta_child_get_buffer_next[]	65
2.2.2.39	dl_meta_child_set_buffer_next[]	65
2.2.2.40	dl_meta_child_get_packet_next[]	65
2.2.2.41	dl_meta_child_set_packet_next[]	66
3	Resource Manager	67
3.1	Defined Types, Enumerations, and Data Structures	69
3.2	System API	70
3.2.1	Defined Types, Enumerations, and Data Structures	71
3.2.1.1	ix_rm_error_code	71

3.2.1.2	<code>ix_phy_type</code>	76
3.2.1.3	<code>ix_port_type</code>	78
3.2.1.4	<code>ix_port</code>	79
3.2.1.5	<code>ix_subsystem_type</code>	79
3.2.1.6	<code>ix_sys_config</code>	80
3.2.1.7	<code>ix_memory_reserved_area</code>	82
3.2.2	API Functions	82
3.2.2.1	<code>ix_rm_init()</code>	82
3.2.2.2	<code>ix_rm_term()</code>	83
3.2.2.3	<code>ix_rm_error_get_string()</code>	84
3.2.2.4	<code>ix_rm_sys_config_get()</code>	84
3.2.2.5	<code>ix_rm_version_get_string()</code>	84
3.2.2.6	<code>ix_rm_sys_config_set()</code>	85
3.3	Microengine API	86
3.3.1	Defined Types, Enumerations, and Data Structures	86
3.3.1.1	<code>ix_imported_symbol</code>	86
3.3.2	API Functions	87
3.3.2.1	<code>ix_rm_ueng_set_ucode()</code>	87
3.3.2.2	<code>ix_rm_ueng_map_ucode()</code>	88
3.3.2.3	<code>ix_rm_ueng_reset_all()</code>	88
3.3.2.4	<code>ix_rm_ueng_patch_symbols()</code>	89
3.3.2.5	<code>ix_rm_ueng_load()</code>	90
3.3.2.6	<code>ix_rm_ueng_start()</code>	90
3.3.2.7	<code>ix_rm_ueng_stop()</code>	91
3.3.2.8	<code>ix_rm_ueng_reset()</code>	92
3.3.2.9	<code>ix_rm_ueng_enable()</code>	92
3.3.2.10	<code>ix_rm_ueng_disable()</code>	93
3.4	Hardware Resource Management API	94
3.4.1	SRAM Queues	94
3.4.1.1	Defined Types, Enumerations, and Data Structures	96
3.4.1.1.1	<code>ix_hw_queue_handle</code>	96
3.4.1.1.2	<code>ix_hw_ring_handle</code>	97
3.4.1.2	API Functions	98
3.4.1.2.1	<code>ix_rm_hw_queue_create()</code>	98
3.4.1.2.2	<code>ix_rm_hw_queue_delete()</code>	100
3.4.1.2.3	<code>ix_rm_hw_queue_array_get_base_address()</code>	100
3.4.1.2.4	<code>ix_rm_hw_enqueue()</code>	101
3.4.1.2.5	<code>ix_rm_hw_dequeue()</code>	102
3.4.2	SRAM and Scratch Rings	103
3.4.2.1	Defined Types, Enumerations, and Data Structures	104
3.4.2.1.1	Handles	104
3.4.2.1.2	<code>ix_sram_ring_size</code>	104
3.4.2.1.3	<code>ix_scratch_ring_size</code>	105
3.4.2.2	API Functions	106
3.4.2.2.1	Bit-Field Macros	106
3.4.2.2.2	Memory Type Macros	106
3.4.2.2.3	Ring Index Macros	106
3.4.2.2.4	<code>ix_rm_hw_sram_ring_create()</code>	107
3.4.2.2.5	<code>ix_rm_hw_scratch_ring_create()</code>	108
3.4.2.2.6	<code>ix_rm_hw_ring_delete()</code>	109
3.4.2.2.7	<code>ix_rm_hw_ring_put()</code>	110
3.4.2.2.8	<code>ix_rm_hw_ring_get()</code>	111
3.5	Buffer Management API	112

3.5.1	Generic Buffers.....	112
3.5.1.1	Defined Types, Enumerations, and Data Structures	114
3.5.1.1.1	ix_buffer_handle	114
3.5.1.1.2	ix_buffer_free_list_handle	115
3.5.1.1.3	ix_buffer_free_list_info	116
3.5.1.1.4	ix_buffer_type	117
3.5.1.2	API Functions	118
3.5.1.2.1	ix_rm_hw_buffer_free_list_create()	118
3.5.1.2.2	ix_rm_sw_buffer_free_list_create()	119
3.5.1.2.3	ix_rm_buffer_free_list_delete()	121
3.5.1.2.4	ix_rm_buffer_free_list_get_info()	121
3.5.1.2.5	ix_rm_buffer_alloc()	122
3.5.1.2.6	ix_rm_buffer_free()	122
3.5.1.2.7	ix_rm_buffer_free_chain()	123
3.5.1.2.8	ix_rm_buffer_get_meta()	123
3.5.1.2.9	ix_rm_buffer_get_data()	124
3.5.1.2.10	ix_rm_buffer_is_eop()	124
3.5.1.2.11	ix_rm_buffer_is_sop()	125
3.5.1.2.12	ix_rm_buffer_get_type()	126
3.5.1.2.13	ix_rm_buffer_get_next()	126
3.5.1.2.14	ix_rm_buffer_link()	127
3.5.1.2.15	ix_rm_buffer_unlink()	128
3.5.2	Framework Buffer Structure	128
3.5.2.1	Packet Metadata Description.....	129
3.5.2.1.1	ix_hw_buffer_meta	129
3.5.2.1.2	Extending Packet Metadata	129
3.5.2.1.3	IX_DECLARE_HW_BUFFER_META_DATA for Common Meta Data....	130
3.5.2.2	Split Meta Data Configuration Details.....	131
3.5.2.2.1	ix_hw_internal_buffer_meta	131
3.5.2.2.2	IX_DECLARE_HW_BUFFER_META_DATA for Split Meta Data	131
3.5.2.3	Packed Field Macros	133
3.6	Communication API.....	134
3.6.1	Defined Types, Enumerations, and Data Structures	138
3.6.1.1	ix_comm_data_handler	138
3.6.1.2	ix_communication_id	138
3.6.1.2.1	ix_comm_select_action_set	139
3.6.1.2.2	ix_comm_id_mode	140
3.6.2	API Functions	140
3.6.2.1	Helper Macros	140
3.6.2.1.1	IX_RM_COMM_ID_GET_LOCAL_ID()	140
3.6.2.1.2	IX_RM_COMM_ID_GET_SYSTEM_TYPE()	141
3.6.2.1.3	IX_RM_COMM_ID_GET_BLADE_ID()	141
3.6.2.1.4	IX_RM_COMM_MAKE_ID()	141
3.6.2.1.5	IX_RM_COMM_MAKE_LOCAL_ID()	141
3.6.2.2	ix_rm_packet_set_receive_mode()	141
3.6.2.3	ix_rm_message_set_receive_mode()	143
3.6.2.4	ix_rm_packet_set_consumer_mode()	143
3.6.2.5	ix_rm_message_set_consumer_mode()	144
3.6.2.6	ix_rm_packet_set_producer_mode()	145
3.6.2.7	ix_rm_message_set_producer_mode()	145
3.6.2.8	ix_rm_packet_handler_register()	146
3.6.2.9	ix_rm_packet_handler_unregister()	147
3.6.2.10	ix_rm_message_handler_register()	147

3.6.2.11	<code>ix_rm_message_handler_unregister()</code>	148
3.6.2.12	<code>ix_rm_packet_send()</code>	149
3.6.2.13	<code>ix_rm_packet_send_wait()</code>	150
3.6.2.14	<code>ix_rm_message_send()</code>	151
3.6.2.15	<code>ix_rm_message_send_wait()</code>	151
3.6.2.16	<code>ix_rm_packet_peek()</code>	152
3.6.2.17	<code>ix_rm_packet_get()</code>	153
3.6.2.18	<code>ix_rm_packet_get_wait()</code>	153
3.6.2.19	<code>ix_rm_message_peek()</code>	154
3.6.2.20	<code>ix_rm_message_get()</code>	155
3.6.2.21	<code>ix_rm_message_get_wait()</code>	155
3.6.2.22	<code>ix_rm_comm_select()</code>	157
3.6.2.23	<code>ix_rm_ublock_packet_comm_init()</code>	158
3.6.2.24	<code>ix_rm_ublock_message_comm_init()</code>	159
3.7	Remote Communication Extension API	160
3.7.1	Defined Types, Enumerations, and Data Structures	161
3.7.1.1	<code>ix_remote_comm_service</code>	161
3.7.2	Callback Function Prototypes	162
3.7.2.1	<code>ix_remote_comm_data_handler</code>	162
3.7.2.2	<code>ix_remote_comm_service_initializer</code>	163
3.7.2.3	<code>ix_remote_comm_service_finalizer</code>	163
3.7.3	API Functions	164
3.7.3.1	<code>ix_rm_remote_comm_service_register()</code>	164
3.7.3.2	<code>ix_rm_remote_comm_service_unregister()</code>	164
3.7.3.3	<code>ix_rm_init_pci_remote_communication()</code>	165
3.7.3.4	<code>ix_rm_register_pci_communication_hw_free_list()</code>	165
3.7.3.5	<code>ix_rm_unregister_pci_communication_hw_free_list()</code>	166
3.8	Memory Management API	167
3.8.1	Defined Types, Enumerations, and Data Structures	168
3.8.1.1	<code>ix_memory_type</code>	168
3.8.1.2	<code>ix_memory_info</code>	169
3.8.1.3	<code>ix_memory_alignment_type</code>	170
3.8.2	API Functions	171
3.8.2.1	<code>ix_rm_mem_alloc()</code>	171
3.8.2.2	<code>ix_rm_mem_alloc_aligned()</code>	172
3.8.2.3	<code>ix_rm_mem_reserve()</code>	173
3.8.2.4	<code>ix_rm_mem_reserve_aligned()</code>	174
3.8.2.5	<code>ix_rm_mem_free()</code>	175
3.8.2.6	<code>ix_rm_mem_info()</code>	176
3.8.2.7	<code>ix_rm_mem_local_alloc()</code>	177
3.8.2.8	<code>ix_rm_mem_local_reserve()</code>	178
3.8.2.9	<code>ix_rm_mem_local_free()</code>	179
3.8.2.10	<code>ix_rm_mem_local_info()</code>	180
3.8.2.11	<code>ix_rm_get_phys_offset()</code>	180
3.8.2.12	<code>ix_rm_get_virtual_address()</code>	181
3.8.2.13	Read/Write Macros	182
3.8.2.13.1	<code>IX_RM_MEM_UINT8_READ</code>	182
3.8.2.13.2	<code>IX_RM_MEM_UINT16_READ</code>	182
3.8.2.13.3	<code>IX_RM_MEM_UINT32_READ</code>	184
3.8.2.13.4	<code>IX_RM_MEM_UINT64_READ</code>	184
3.8.2.13.5	<code>IX_RM_MEM_UINT8_WRITE</code>	185
3.8.2.13.6	<code>IX_RM_MEM_UINT16_WRITE</code>	185
3.8.2.13.7	<code>IX_RM_MEM_UINT32_WRITE</code>	186

	3.8.2.13.8 IX_RM_MEM_UINT64_WRITE	186
3.9	System Repository API.....	187
3.9.1	Defined Types, Enumerations, and Data Structures	188
	3.9.1.1 ix_configuration_property_handle	188
	3.9.1.2 ix_cp_property_info	188
3.9.2	API Functions	189
	3.9.2.1 ix_rm_cp_property_create()	189
	3.9.2.2 ix_rm_cp_property_delete()	190
	3.9.2.3 ix_rm_cp_property_open()	191
	3.9.2.4 ix_rm_cp_property_close()	192
	3.9.2.5 ix_rm_cp_property_attach()	193
	3.9.2.6 ix_rm_cp_property_detach()	194
	3.9.2.7 ix_rm_cp_property_set_value()	195
	3.9.2.8 ix_rm_cp_property_get_value()	196
	3.9.2.9 ix_rm_cp_property_set_value_uint32()	197
	3.9.2.10 ix_rm_cp_property_get_value_uint32()	198
	3.9.2.11 ix_rm_cp_property_delete_value()	199
	3.9.2.12 ix_rm_cp_property_get_info()	199
	3.9.2.13 ix_rm_cp_property_get_subproperty()	200
3.10	64-Bit Counters API.....	200
3.10.1	Defined Types, Enumerations, and Data Structures	201
	3.10.1.1 ix_counter_64bit_handle()	201
3.10.2	API Functions	201
	3.10.2.1 ix_rm_counter_64bit_new()	201
	3.10.2.2 ix_rm_counter_64bit_delete()	203
	3.10.2.3 ix_rm_counter_64bit_get_internal_overflow_time()	203
	3.10.2.4 ix_rm_counter_64bit_set_internal_overflow_time()	204
	3.10.2.5 ix_rm_counter_64bit_get_value()	204
	3.10.2.6 ix_rm_counter_64bit_set_value()	205
3.11	Services API.....	206
3.11.1	API Functions	207
	3.11.1.1 ix_rm_atomic_sram_swap()	207
	3.11.1.2 ix_rm_atomic_sram_add()	207
	3.11.1.3 ix_rm_atomic_sram_test_and_add()	208
	3.11.1.4 ix_rm_atomic_sram_subtract()	209
	3.11.1.5 ix_rm_atomic_sram_test_and_subtract()	209
	3.11.1.6 ix_rm_atomic_sram_bit_set()	210
	3.11.1.7 ix_rm_atomic_sram_bit_test_and_set()	210
	3.11.1.8 ix_rm_atomic_sram_bit_clear()	211
	3.11.1.9 ix_rm_atomic_sram_bit_test_and_clear()	212
	3.11.1.10 ix_rm_atomic_scratch_swap().....	212
	3.11.1.11 ix_rm_atomic_scratch_add().....	213
	3.11.1.12 ix_rm_atomic_scratch_test_and_add().....	213
	3.11.1.13 ix_rm_atomic_scratch_subtract().....	214
	3.11.1.14 ix_rm_atomic_scratch_test_and_subtract().....	215
	3.11.1.15 ix_rm_atomic_scratch_bit_set().....	215
	3.11.1.16 ix_rm_atomic_scratch_bit_test_and_set().....	216
	3.11.1.17 ix_rm_atomic_scratch_bit_clear().....	217
	3.11.1.18 ix_rm_atomic_scratch_bit_test_and_clear().....	217
	3.11.1.19 ix_rm_managed_to_os_memory_copy().....	218
	3.11.1.20 ix_rm_os_to_managed_memory_copy().....	219
	3.11.1.21 ix_rm_managed_to_managed_memory_copy().....	219
3.12	Hash API	220

3.12.1	Defined Types, Enumerations, and Data Structures	221
3.12.1.1	ix_hash_48	221
3.12.1.2	ix_hash_64	221
3.12.1.3	ix_hash_128	222
3.12.1.4	ix_hash_multiplier_48	222
3.12.1.5	ix_hash_multiplier_64	223
3.12.1.6	ix_hash_multiplier_128	223
3.12.2	API Functions	223
3.12.2.1	ix_rm_hash_48_hash()	223
3.12.2.2	ix_rm_hash_48_multiplier_set()	224
3.12.2.3	ix_rm_hash_48_multiplier_get()	224
3.12.2.4	ix_rm_hash_64_hash()	224
3.12.2.5	ix_rm_hash_64_multiplier_set()	225
3.12.2.6	ix_rm_hash_64_multiplier_get()	225
3.12.2.7	ix_rm_hash_128_hash()	226
3.12.2.8	ix_rm_hash_128_multiplier_set()	226
3.12.2.9	ix_rm_hash_128_multiplier_get()	227
3.13	Microengine Services API.....	227
3.13.1	Defined Types, Enumerations, and Data Structures	228
3.13.1.1	ix_me_xscale_lock_handle	228
3.13.1.2	ix_me_xscale_lock_status	229
3.13.1.3	ix_me_xscale_lock_owner	229
3.13.1.4	ix_me_xscale_lock_info	229
3.13.1.5	ix_me_transfer_register_type	230
3.13.2	API Functions	230
3.13.2.1	ix_rm_me_xscale_lock_new()	230
3.13.2.2	ix_rm_me_xscale_lock_delete()	231
3.13.2.3	ix_rm_me_xscale_lock_acquire()	231
3.13.2.4	ix_rm_me_xscale_lock_release()	232
3.13.2.5	ix_rm_me_xscale_lock_get_info()	232
3.13.2.6	ix_rm_me_transfer_register_read()	233
3.13.2.7	ix_rm_me_transfer_register_write()	234
3.13.2.8	ix_rm_me_signal()	234
3.14	Debug Support API.....	235
3.14.1	API Functions	236
3.14.1.1	ix_rm_mem_status_print()	236
3.14.1.2	ix_rm_scratch_ring_print_info()	237
3.14.1.3	ix_rm_scratch_ring_print_data()	237
3.14.1.4	ix_rm_sram_ring_print_info()	237
3.14.1.5	ix_rm_sram_ring_print_data()	238
3.14.1.6	ix_rm_free_list_print_available_buffers()	238
3.14.1.7	ix_rm_free_list_print_buffers_info()	239
3.14.1.8	ix_rm_free_list_print_info()	239
3.14.1.9	ix_rm_buffer_print_meta()	240
3.14.1.10	ix_rm_buffer_print_data()	240
3.14.1.11	ix_rm_buffer_print_debug_info()	240
4	Core Component Infrastructure	243
4.1	API Functions	244
4.1.1	ix_cci_cc_add_event_handler()	244
4.1.2	ix_event_func()	245
4.1.3	ix_cci_cc_add_event_handler_ex()	246
4.1.4	ix_cci_change_event()	247

4.1.5	<code>ix_cci_cc_add_message_handler()</code>	248
4.1.5.1	<code>ix_msg_handler()</code>	250
4.1.6	<code>ix_cci_cc_add_packet_handler()</code>	251
4.1.6.1	<code>ix_pkt_handler()</code>	252
4.1.7	<code>ix_cci_cc_create()</code>	253
4.1.7.1	<code>ix_cc_init()</code>	254
4.1.7.2	<code>ix_cc_fini()</code>	254
4.1.8	<code>ix_cci_cc_destroy()</code>	255
4.1.9	<code>ix_cci_cc_remove_event_handler()</code>	256
4.1.10	<code>ix_cci_cc_remove_message_handler()</code>	257
4.1.11	<code>ix_cci_cc_remove_packet_handler()</code>	258
4.1.12	<code>ix_cci_exe_add_policy()</code>	258
4.1.13	<code>ix_cci_exe_get_info()</code>	259
4.1.14	<code>ix_cci_exe_run()</code>	260
4.1.14.1	<code>ix_exe_init()</code>	262
4.1.14.2	<code>ix_exe_fini()</code>	263
4.1.15	<code>ix_cci_exe_set_default()</code>	264
4.1.16	<code>ix_cci_exe_shutdown()</code>	265
4.1.17	<code>ix_cci_init()</code>	266
4.1.18	<code>ix_cci_fini()</code>	266
4.1.19	<code>ix_cci_policy_add_branch()</code>	267
4.1.20	<code>ix_cci_policy_add_leaf()</code>	268
4.1.21	<code>ix_cci_policy_create()</code>	269
4.1.22	<code>ix_cci_policy_destroy()</code>	270
4.1.23	<code>ix_cci_register_fatal_error_handler()</code>	271
4.1.23.1	<code>ix_ferror_func()</code>	272
4.1.24	<code>ix_cci_send_message()</code>	273
4.1.25	<code>ix_cci_send_packet()</code>	274
4.2	Symbolic Constants—Tuning Behavior and Memory Footprint.....	275
5	TCAM Lookup Libraries.....	279
5.1	Defined Types, Enumerations, and Data Structures	280
5.1.1	Constants	280
5.1.2	<code>ix_lkup</code>	280
5.1.3	<code>ix_lkup_table</code>	281
5.1.4	<code>ix_lkup_table_type</code>	281
5.1.5	<code>ix_lkup_tcam_params</code>	282
5.1.6	<code>ix_lkup_table_conf</code>	282
5.1.7	<code>ix_lkup_cookie</code>	283
5.2	Lookup Management Library.....	283
5.2.1	Initialization APIs	283
5.2.1.1	<code>ix_lkup_sw_init()</code>	283
5.2.1.2	<code>ix_lkup_tcam_init()</code>	284
5.2.2	Table Macros.....	285
5.2.2.1	<code>IX_LKUP_CREATE_TABLE()</code>	285
5.2.2.2	<code>IX_LKUP_DESTROY_TABLE()</code>	286
5.2.2.3	<code>IX_LKUP_FINI()</code>	287
5.2.2.4	<code>IX_LKUP_ADD_ENTRY()</code>	288
5.2.2.5	<code>IX_LKUP_REMOVE_ENTRY()</code>	289
5.2.2.6	<code>IX_LKUP_UPDATE_ENTRY()</code>	290
5.2.2.7	<code>IX_LKUP_SEARCH_TABLE()</code>	291

5.2.2.8	IX_LKUP_FIND_ENTRY()	292
5.2.2.9	IX_LKUP_READ_FIRST_ENTRY()	293
5.2.2.10	IX_LKUP_READ_NEXT_ENTRY()	294
5.2.2.11	IX_LKUP_RESET_TABLE()	295
5.2.2.12	IX_LKUP_SET_PROPERTY()	296
5.2.2.13	IX_LKUP_GET_PROPERTY()	297
5.2.2.14	IX_LKUP_GET_TABLE_INFO()	298
5.3	Microengine Hardware Lookup	299
5.3.1	TCAM Lookup APIs	299
5.3.1.1	ix_tcam_lkup_build_handle()	299
5.3.1.2	ix_tcam_lkup_start()	300
5.3.1.3	ix_tcam_lkup_complete()	301
5.3.1.4	ix_tcam_lkup_get_data()	302
5.4	Microengine Software Lookup	303
5.4.1	Longest Prefix Match APIs	303
5.4.1.1	ix_sw_lkup_lpm_build_handle()	304
5.4.1.2	ix_sw_lkup_lpm_search()	304
5.4.2	Exact Match APIs	305
5.4.2.1	ix_sw_lkup_exact_build_handle()	305
5.4.2.2	ix_sw_lkup_exact_search()	306
5.4.3	Range Match APIs	307
5.4.3.1	ix_sw_lkup_range_build_handle()	307
5.4.3.2	ix_sw_lkup_range_search()	308
5.5	Implementation Considerations	309
5.5.1	ix_s_lkup	309
5.5.2	ix_s_lkup_table	310
6	Operating System Services Layer (OSSL) Support	311
7	Intel XScale® Core Support	313
8	Optimized Data Plane Libraries Support	315
9	Metadata Configuration Tool	317
9.1	Introduction	317
9.2	Metadata Configuration Tool Process	317
9.2.1	Creating an Application Configuration File	317
9.2.1.1	Naming Conventions	317
9.2.1.2	Application Configuration File Contents	318
9.2.2	Creating a Dispatch Loop Configuration File	318
9.2.2.1	Naming Conventions	318
9.2.2.2	Dispatch Loop Configuration File Contents	318
9.2.3	Creating a Microblock Configuration File	319
9.2.3.1	Naming Conventions	319
9.2.3.2	Microblock Configuration File Contents	319
9.3	List File Overview	321
9.4	dl_meta.uc and buffer.h File Overview	321
9.4.1	dl_meta.uc File Description	321
9.4.1.1	Accessor Macros	321
9.4.1.2	Block Macros	322
9.4.1.3	Dispatch Loop Constants	322
9.4.1.4	dl_meta.uc Example	322
9.4.2	buffer.h File Description	324

9.5	Using the Metadata Configuration Tool	325
9.5.1	System and Software Requirements	325
9.5.2	File Locations	325
9.5.3	Invoking the Tool	326
A	Glossary	327

Figures

3-1	Hardware Queue Handle Encoding	96
3-2	Ring Handle Encoding	97
3-3	Hardware Buffer Bit-Field Mapping	114
3-4	Software Buffer Bit-Field Mapping	115

Tables

2-1	Microengine Assembler Dispatch Loop Variables	19
2-2	Dispatch Loop API Functions for Meta Data	23
2-3	Dispatch Loop API Functions for Extended Meta Data	24
3-1	Resource Manager API Functional Groups	68
3-2	Basic Types Supported by the Resource Manager	69
3-3	Resource Manager System API	70
3-4	Error Codes for the Resource Manager	71
3-5	Resource Manager Microengine API	86
3-6	Resource Manager Hardware API	94
3-7	Resource Manager Buffer Management API	113
3-8	Resource Manager Packet Metadata Definitions	130
3-9	Resource Manager Packet Metadata Definitions for Split Meta Data	132
3-10	Second 32-bit Word For Resource Manager Communication Signaling to the Core	135
3-11	Resource Manager Communication API	136
3-12	Bit Field Mapping for ix_communication_id	139
3-13	Resource Manager Remote Communication Extension API	160
3-14	Resource Manager Memory Management API	168
3-15	Resource Manager Memory Management Macros	182
3-16	Resource Manager System Repository API	187
3-17	Resource Manager 64-Bit Counter API	201
3-18	Resource Manager Services API	206
3-19	Resource Manager Hash API	220
3-20	Resource Manager Microengine Services API	228
3-21	Resource Manager Debug Support API	236
4-1	cci API	243
5-1	TCAM Lookup Library	279
9-1	Field Name Values	320



Revision History

Date	Revision	Description
May 2002	001	SDK 3.0 Pre-Release 3
August 2002	002	SDK 3.0 Release 4
October 2002	003	SDK 3.0 Release 5
February 2003	004	SDK 3.0 Release 6 Field Trial
April 2003	005	SDK 3.0 Release 6 FCS Removed chapters describing the Optimized Data Plane Libraries (HAL for microengines, Utilities, and Protocols) and added a pointer to the Optimized Data Plane Libraries Reference Manual (part of the IXA SDK Tools Release). Removed chapter describing the OSSL Libraries and added a pointer to the Software Reference Manual (part of the IXA SDK Tools Release).
July 2003	006	SDK 3.1 Release Added new chapter for TCAM Lookup Libraries.
November 2003	007	SDK 3.5 Release In Dispatch Loop chapter: <ul style="list-style-type: none">Added new APIs for handling extended meta data. In Resource Manager chapter: <ul style="list-style-type: none">Added new APIs for Microengine Services and Debug Support.Significantly modified Services API to include 12 new APIs.Added support for hardware free list queue on all channels.Added description of split meta data configuration.Added Resource Manager Error Codes table. Added new chapter for Metadata Configuration Tool. Throughout manual: Changed multiple mentions of "Intel XScale™ core" to "Intel XScale® core" due to trademark registration.

1.1 About this Document

This Reference Manual introduces you to the Intel Exchange Architecture (IXA) Portability Framework, which is a part of the IXA Software Development Kit (SDK). This software enables you to develop and deliver network applications that utilize the Intel® IXP2400 and IXP2800 Network Processors.

This manual provides details on functions, data structures, and parameters contained in the Portability Framework libraries. This manual is a companion guide to the *Intel® Internet Exchange Architecture Portability Framework Developer's Manual* which provides guidelines for using the Portability Framework to develop applications.

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 or Assembler Programming
- Realtime network applications

1.3 IXA Portability Framework Overview

The IXA Portability Framework consists of microengine microblocks, flow-of-control among microblocks through dispatch loops, the coordination of microengine and Intel XScale® core resources through the Resource Manager, as well as Intel XScale® core common components.

Network processing in Intel IXA is essentially a series of tasks that are applied to a constant stream of packet or cell data. With the multi-processor/multi-threaded architecture of the IXP 2400 and IXP 2800 network processors, these tasks are distributed over several microengines, each of which is programmed to perform specific tasks. When a microengine completes its tasks, it passes the context to the next microengine so that it can continue processing the data.

The IXA Portability Framework is implemented using a layered architecture on both the MEv2 microengines and the Intel XScale[®] core. When developing an IXA Application using this layered architecture, application code can use the entire IXA Portability Framework or can use only part of the framework up to a specific level. For example, you can choose to write microblocks on the microengines but use only the Resource Manager API on the Intel XScale[®] core.

For a complete introduction to the IXA Portability Framework, see *Intel[®] Internet Exchange Architecture Portability Framework Developer's Manual*.

1.4 In This Manual

This manual includes the following chapters:

- [Chapter 1, “Introduction,”](#) (this chapter) presents the organization of this manual and provides an overview of the IXA Portability Framework.
- [Chapter 2, “Dispatch Loop,”](#) describes support for application-specific microengine flow of control as well as various utilities.
- [Chapter 3, “Resource Manager,”](#) describes the programming interface between Intel XScale[®] core applications and the microcode running on the microengines for the Intel[®] IXP2400 and IXP2800 Network Processors.
- [Chapter 4, “Core Component Infrastructure,”](#) describes the core component infrastructure interface which provides framework support for core components running in their own execution engines and for prioritizing message and packet data paths.
- [Chapter 5, “TCAM Lookup Libraries,”](#) describes a common API used for managing and searching tables on the Intel XScale[®] core and on the microengines for Intel[®] IXP2400 and IXP2800 Network Processors.
- [Chapter 6, “Operating System Services Layer \(OSSL\) Support,”](#) briefly describes APIs that provide portability across the operating systems and provides a cross-reference to further documentation.
- [Chapter 7, “Intel XScale[®] Core Support,”](#) briefly describes APIs that provide additional support for Intel XScale[®] core applications and provides a cross-reference to further documentation.
- [Chapter 8, “Optimized Data Plane Libraries Support,”](#) briefly describes the optimized data plane libraries and provides a cross-reference to further documentation.
- [Chapter 9, “Metadata Configuration Tool,”](#) contains full details on a utility that takes metadata configuration files as input and creates an output file containing metadata fields, block macros, and accessor macros for get and set operations.

1.5 Other Sources of Information

This manual is part of the Intel[®] Internet Exchange Architecture Software Development Kit (Intel[®] IXA SDK) documentation set. The following documents are located on the IXA SDK Software Framework CD:

- *Intel[®] Internet Exchange Architecture Software Development Kit Software Framework Getting Started Guide*
- *Intel[®] Internet Exchange Architecture Portability Framework Developer's Manual*

- *Intel® Internet Exchange Architecture Software Building Blocks Developer's Manual*
- *Intel® Internet Exchange Architecture Software Building Blocks Reference Manual*
- *Intel® Internet Exchange Architecture Software Building Blocks Applications Design Guide*

The following documents are also relevant when using the IXA SDK. They are located on the IXA SDK Tools CD:

- *Intel® Internet Exchange Architecture (IXA) Software Reference Manual*
- *Intel® Internet Exchange Architecture Optimized Data Plane Libraries Reference Manual*
- *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 Hardware Reference Manual*
- *Intel® IXP2400 Network Processor Hardware Reference Manual*

The following documents are available from the Internet.

- *Intel® IXP2400 Network Processor Datasheet*
- *Intel® IXP2800 Network Processor Datasheet*

They can be retrieved at: <http://fdbi.intel.com>.

The dispatch loop combines microblocks on a microengine and implements the data flow between them. The dispatch loop also caches commonly used variables in registers or local memory. These variables can be accessed by microblocks using a set of helper macros or Microengine C functions. The dispatch loop also provides source and sink blocks to send and receive packets to the Intel XScale® core and to send packets to a different microblock group. Each of these is described in this chapter.

Note: A dispatch loop is application-specific. Microblocks should be as reusable as possible. But the dispatch loop and the internal implementation of its associated helper macros or functions may, and often must, be optimized for a specific application.

For a more detailed description of Dispatch Loop concepts, see *Intel® Internet Exchange Architecture Portability Framework Developer's Manual*.

This chapter describes dispatch loop related programming interfaces.

2.1 Dispatch Loop Variables

The dispatch loop maintains some global state cached in registers or local memory. For Microengine C, this state is maintained in a global C structure. The compiler decides whether this structure can be cached in registers or if some data structure elements are cached in local memory.

2.1.1 Microengine Assembler Dispatch Variables

Table 2-1 lists variables that may be cached by a dispatch loop—the actual variables cached depend on the nature of the application and can be customized for a specific application.

Table 2-1. Microengine Assembler Dispatch Loop Variables¹

Field Name	Size	Use
exception_id	8 bits	This is used by microblocks when sending packets to the Intel XScale® core. The microblock must set the exception_id to the microblock ID when indicating an exception.
exception_code	8 bits	The microblock sets an 8-bit exception code when a buffer is sent to the Intel XScale® core component. This exception code is treated as opaque data by the Dispatch Loop and Resource Manager.
dl_next_block	8 bits	Identifies the next logical block to process after the current block. The current block sets this value after it is done processing.
dl_buf_handle	32 bits	The buffer handle containing the start of the packet.

Table 2-1. Microengine Assembler Dispatch Loop Variables¹ (Continued)

Field Name	Size	Use
dl_eop_buf_handle	32 bits	The buffer handle containing the end of the packet.
buffer_size	16 bits	This is the length of the buffer containing the start of the current packet. This is the total length of the buffer including all of the headers. If the buffer is not complete, this is the amount of data currently in the buffer.
packet_size	16 bits	This is the total length of the packet across multiple buffers.
buffer_offset	16 bits	This is the offset from the start of the buffer to the buffer data. Transform blocks that pack or unpack the buffer must change this offset.
input_port	16 bits	This contains the logical port number on which the packet was received. Port numbers should be in the range of zero to 255.
rx_stat	4 bits	These are receive status flags for a buffer. Supported flags are IX_RXSTAT_UCAST, IX_RXSTAT_BCAST, IX_RXSTAT_MCAST, and IX_RXSTAT_PROMISC.
output_port_egress	24 bits	This is the number of the port interface on which the packet is to be transmitted in a given blade.
output_port_fabric	8 bits	When multiple blades are connected to the fabric this is the blade ID.
output_port_type	4 bits	The type of interface on which the packet is to be transmitted—for example, POS, ATM, Ethernet, and so on.
cache_flags	4 bits	This is used for caching packet headers. Each bit represents 32 bytes of the packet header—the cache line. 2 bits are used to detect if a cache line is in local memory. 2 bits are used to check if a cache line is dirty and needs to be written out.
next_hop_id	32 bits	The next hop IP ID.
flow_id (QoS only)	32 bits	The flow identifier that is used for metering and other QoS functions.
queue_id (QoS only)	16 bits	The output queue identifier. This is set when classifying packets for quality of service processing.

1. These variables may be cached.

Apart from the above, other variables specific to POS, Ethernet, or ATM may be cached in an extension to this structure.

2.1.2 Microengine C Loop Data Structure

For microblocks written in Microengine C, the variables described in [Section 2.1.1](#) and the packet headers are stored in global data structures. This section describes the data structure used to store this data. Microblocks written in Microengine C access data in this structure by directly referencing its member fields. Consequently there are no get or set functions for this data structure.

Microengine C Syntax

```
typedef __declspec(packed) union {
    struct {
        dl_buf_handle_t buffer_next;
        uint16_t buffer_size;
        uint16_t offset;
        uint32_t packet_size      : 16;
        uint32_t free_list_id     : 4;
        uint32_t rx_stat          : 4;
        uint32_t header_type      : 8;
        uint16_t input_port;
        uint16_t output_port;
        uint32_t next_hop_id      : 16;
        uint32_t fabric_port      : 8;
        uint32_t reserved         : 4;
        uint32_t nhid_type        : 4;
        uint32_t flow_id;
        uint32_t class_id;
        uint32_t reserved_2;
        uint32_t packet_next;
    }; // end of struct
    //
    uint32_t value[8]; /* aggregate for the above fields */
    //
} dl_meta_t;
```

Data Members

buffer_next	The next buffer in the chain.
buffer_size	The amount of data currently in the buffer.
offset	The offset in DRAM where the data begins.
packet_size	The amount of data in the buffer chain.
free_list_id	The free list to which this buffer belongs.
rx_stat	The receive status.
header_type	The header type—IPv4, IPv6, and so on.
input_port	The input port on which this packet was received.
output_port	The output port on which this packet is to be transmitted.

Data Members (Continued)

<code>next_hop_id</code>	The next hop ID.
<code>fabric_port</code>	The blade port.
<code>reserved</code>	Reserved for future use.
<code>nhid_type</code>	The next hop ID type.
<code>flow_id</code>	The flow ID.
<code>class_id</code>	The class ID.
<code>reserved_2</code>	Reserved for future use.
<code>packet_next</code>	The next packet in the chain—used only in <i>Hierarchical Queuing</i> .

2.2 Dispatch Loop Interface

Of the variables described in [Section 2.1, “Dispatch Loop Variables,”](#) the buffer handles and the next block are stored in global variables (`dl_buf_handle`, `dl_eop_buf_handle`, and `dl_next_block`). The remaining variables are packed into registers.

The IXA SDK provides macros for buffer allocation, buffer freeing, the return or modification of various IP header fields, and so on. There are two categories of helper macros that support Dispatch Loops. The following sections provide full details on these macros:

- [Section 2.2.1, “Dispatch Loop API Functions for Meta Data”](#) on page 26
- [Section 2.2.2, “Dispatch Loop API Functions for Extended Meta Data”](#) on page 47

[Table 2-2](#) summarizes the Dispatch Loop macros supporting meta data and [Table 2-3](#) summarizes the macros that support extended meta data.

Table 2-2. Dispatch Loop API Functions for Meta Data

Name	Description
<code>dl_buf_init[]</code>	Initialize the Buffer API.
<code>dl_buf_alloc[]</code>	Allocates a buffer.
<code>dl_buf_free[]</code>	Frees a buffer.
<code>dl_buf_get_desc[]</code>	Returns the SRAM pointer to the meta data given a buffer handle.
<code>dl_buf_get_data[]</code>	Returns the DRAM pointer to the buffer data given a buffer handle.
<code>dl_buf_get_data_from_meta[]</code>	Returns the DRAM pointer using the SRAM base as input.
<code>dl_meta_init_cache[]</code>	Populates a meta data cache.
<code>dl_meta_flush_cache[]</code>	Flushes meta data to SRAM.
<code>dl_meta_load_cache[]</code>	Loads meta data from SRAM into registers.
<code>dl_meta_get_buffer_next[]</code>	Returns the handle of next buffer in the buffer chain—for large packets.
<code>dl_meta_set_buffer_next[]</code>	Sets the handle of next buffer in the buffer chain—for large packets.
<code>dl_meta_get_hw_next[]</code>	Gets the hardware next field in the handle
<code>dl_meta_set_hw_next[]</code>	Sets the hardware next field in the handle
<code>dl_meta_get_offset[]</code>	Returns the offset at which data begins within a buffer.
<code>dl_meta_set_offset[]</code>	Sets the offset at which data begins within a buffer.
<code>dl_meta_get_free_list[]</code>	Returns the free list from which the current buffer—that is, the buffer pointed to by <code>dl_buf_handle</code> —was allocated. There may be multiple free lists—that is, buffer pools—but only one is in use at any point in time.
<code>dl_meta_set_free_list[]</code>	Sets the free list to which the current buffer belongs. There may be multiple free lists—that is, buffer pools—but only one is currently used.
<code>dl_meta_get_rx_stat[]</code>	Returns the receive status.
<code>dl_meta_set_rx_stat[]</code>	Sets the receive status.
<code>dl_meta_get_buffer_size[]</code>	Returns the buffer size of the current buffer in the packet.

Table 2-2. Dispatch Loop API Functions for Meta Data (Continued)

Name	Description
<code>dl_meta_set_buffer_size[]</code>	Sets the buffer size of the current buffer in the packet.
<code>dl_meta_get_input_port[]</code>	Returns the input port over which the packet came in.
<code>dl_meta_set_input_port[]</code>	Sets the input port.
<code>dl_meta_get_packet_size[]</code>	Returns the total packet size.
<code>dl_meta_set_packet_size[]</code>	Sets the total packet size.
<code>dl_meta_get_nexthop_id[]</code>	Returns the ID for the next hop.
<code>dl_meta_set_nexthop_id[]</code>	Sets the ID for the next hop.
<code>dl_meta_get_output_port[]</code>	Returns the output port. This is the port on the egress IXP2400 out of which the packet is transmitted.
<code>dl_meta_set_output_port[]</code>	Sets the output port. This is the port on the egress IXP2400 out of which the packet is transmitted.
<code>dl_meta_get_fabric_port[]</code>	Returns the output blade (when multiple blades are connected to the fabric) from which the packet is transmitted.
<code>dl_meta_set_fabric_port[]</code>	Sets the output blade (when multiple blades are connected to the fabric) from which the packet is transmitted out.
<code>dl_meta_get_flow_id[]</code>	Returns the flow ID.
<code>dl_meta_set_flow_id[]</code>	Sets the flow ID.
<code>dl_meta_get_class_id[]</code>	Returns the class ID.
<code>dl_meta_set_class_id[]</code>	Sets the class ID.
<code>dl_buf_set_SOP[]</code>	Sets the SOP bit in the buffer handle. This indicates that the buffer contains the start-of-packet.
<code>dl_buf_set_EOP[]</code>	Sets the EOP bit in the buffer handle. This indicates that the buffer contains the end-of-packet.
<code>dl_buf_get_cell_count[]</code>	Gets cell count from the buffer handle.
<code>dl_buf_set_cell_count[]</code>	Sets the cell count in the buffer handle.
<code>dl_set_exception[]</code>	Sets the exception code.
<code>dl_meta_get_nexthop_id_type[]</code>	Returns the next hop ID type—IPv4, IPv6, and so on.
<code>dl_meta_set_nexthop_id_type[]</code>	Sets the next hop ID type—IPv4, IPv6, and so on.

Table 2-3. Dispatch Loop API Functions for Extended Meta Data

Name	Description
<code>dl_meta_parent_get_ref_cnt[]</code>	Obtains the reference count value.
<code>dl_meta_child_get_child_offset[]</code>	Obtains the child buffer data offset in bytes.
<code>dl_meta_child_set_child_offset[]</code>	Sets the child buffer data offset in bytes.
<code>dl_meta_child_get_child_buffer_size[]</code>	Obtains the child buffer data size in bytes.
<code>dl_meta_child_set_child_buffer_size[]</code>	Sets the child buffer data size in bytes.
<code>dl_meta_child_get_child_freelist_id[]</code>	Obtains the freelist ID of the child buffer.
<code>dl_meta_child_set_child_freelist_id[]</code>	Sets the freelist ID of the child buffer.
<code>dl_meta_child_get_parent_offset[]</code>	Obtains the data offset of the parent buffer.

Table 2-3. Dispatch Loop API Functions for Extended Meta Data (Continued)

Name	Description
dl_meta_child_set_parent_offset[]	Sets the data offset of the parent buffer.
dl_meta_child_get_parent_buffer_size[]	Obtains the data size of the parent buffer.
dl_meta_child_set_parent_buffer_size[]	Sets the data size of the parent buffer.
dl_meta_child_get_header_type[]	Obtains the header type of the packet.
dl_meta_child_set_header_type[]	Sets the header type of the packet.
dl_meta_child_get_parent_free_list[]	Obtains the free list ID of the parent buffer.
dl_meta_child_set_parent_free_list[]	Sets the free list ID of the parent buffer.
dl_meta_child_get_rx_stat[]	Obtains the receive status of the packet.
dl_meta_child_set_rx_stat[]	Sets the receive status of the packet.
dl_meta_child_get_packet_size[]	Obtains the size of the packet across all buffers.
dl_meta_child_set_packet_size[]	Sets the size of the packet across all buffers.
dl_meta_child_get_output_port[]	Obtains the output port number for this packet.
dl_meta_child_set_output_port[]	Sets the output port number for this packet.
dl_meta_child_get_input_port[]	Obtains the input port number for this packet.
dl_meta_child_set_input_port[]	Sets the input port number for this packet.
dl_meta_child_get_nexthop_id[]	Obtains the next hop ID for this packet.
dl_meta_child_set_nexthop_id[]	Sets the next hop ID for this packet.
dl_meta_child_get_fabric_port[]	Obtains the fabric port number for this packet.
dl_meta_child_set_fabric_port[]	Sets the fabric port number for this packet.
dl_meta_child_get_nexthop_id_type[]	Obtains the nexthop ID type for this packet.
dl_meta_child_set_nexthop_id_type[]	Sets the nexthop ID type for this packet.
dl_meta_child_get_flow_id[]	Obtains the flow ID for this packet.
dl_meta_child_set_flow_id[]	Sets the flow ID for this packet.
dl_meta_child_get_color[]	Obtains the color of this packet.
dl_meta_child_set_color[]	Sets the color of this packet.
dl_meta_child_get_class_id[]	Obtains the class ID of this packet.
dl_meta_child_set_class_id[]	Sets the class ID of this packet.
dl_meta_child_get_parent_buffer_id[]	Obtains the parent buffer ID to which this child buffer is linked.
dl_meta_child_set_parent_buffer_id[]	Sets the parent buffer ID to which this child buffer is linked.
dl_meta_child_get_buffer_next[]	Obtains the next buffer handle for this child buffer.
dl_meta_child_set_buffer_next[]	Sets the next buffer handle for this child buffer.
dl_meta_child_get_packet_next[]	Obtains the next packet handle for this child buffer.
dl_meta_child_set_packet_next[]	Sets the next packet handle for this child buffer.

2.2.1 Dispatch Loop API Functions for Meta Data

This section describes the dispatch loop Microengine Assembler macros in detail.

2.2.1.1 `dl_buf_init[]`

Initializes the buffering mechanism. This macro should be called once before any of the other `dl_buffer` macros can be called. Typically this macro creates the freelist of buffers.

Microengine Assembler Syntax

```
#macro dl_buf_init[]
```

Estimated Size

N/A

2.2.1.2 `dl_buf_alloc[]`

Allocates a free packet buffer. If no buffer is available this function returns zero in `buffer_handle`. There may be multiple pools of buffers in which case the caller specifies the pool from which this buffer needs to be allocated.

Microengine Assembler Syntax

```
#macro dl_buf_alloc(buf_handle, free_list, req_sig, sig_action)
```

Input

<code>free_list</code>	Specifies the pool from which the buffer is to be allocated.
<code>req_sig</code>	Signal to use in the I/O operation.
<code>sig_action</code>	The flag that determines how this operation responds to the I/O operation. <ul style="list-style-type: none"> • If this argument contains a signal expression, wait for one or more specified signals • If this argument contains <code>SIG_NONE</code>, do not wait—simply return from this call For details, see the <i>Intel® Internet Exchange Architecture Optimized Data Plane Libraries Reference Manual</i> located on the IXA SDK Tools CD.

Output

<code>buf_handle</code>	The handle of the newly allocated buffer. The <code>SOP</code> and <code>count</code> fields are set to zero. The <code>EOP</code> Field is set to one. <code>buf_handle</code> <i>must</i> be an SRAM transfer register.
-------------------------	---

Estimated Size

N/A

2.2.1.3 `dl_buf_free[]`

Frees the buffer that was previously allocated using `dl_buf_alloc[]`. Only one buffer can be freed at a time by this macro. Chained buffers—used to represent large packets—are not supported by this macro. EOP, SOP and cell count in the buffer handle are all reset to zero by this macro—the application need not set them to zero before calling this function.

Microengine Assembler Syntax

```
#macro dl_buf_free[buf_handle, free_list]
```

Input

<code>buf_handle</code>	The buffer handle.
<code>free_list</code>	Specifies the pool to which this buffer is to be released.

Estimated Size

N/A

2.2.1.4 `dl_buf_get_desc[]`

Given an opaque buffer handle, this macro returns the SRAM address where the meta data—called the buffer descriptor—specified by `buf_handle` is present. The lower 24 bits of the handle correspond to the SRAM address. No additional arithmetic is required. The address returned is an absolute address and not an offset from a base address.

Microengine Assembler Syntax

```
#macro dl_buf_get_desc[sram_offset, buf_handle]
```

Input

<code>buf_handle</code>	The buffer handle.
-------------------------	--------------------

Output

<code>sram_offset</code>	The SRAM byte address where the meta data for this buffer is present.
--------------------------	---

Estimated Size

Two or three instructions.

2.2.1.5 `dl_buf_get_data[]`

Given an opaque buffer handle pointing to a buffer, this macro returns the SDRAM address where packet data for that buffer is present. The lower 24 bits of the buffer handle correspond to the SRAM offset for meta data. From this we calculate the DRAM offset as follows:

```
meta=buf_handle & 0xffffffff;
index=(meta - sram_base) / sizeof (meta_data)
data =sram_base + index * size_of_packet_buffer_in_dram
```

However the above calculation is optimized as follows (grouping all constants together):

```
DL_DS_RATIO = Sizeof (Packet buffer) / Sizeof (Meta data)
DL_REL_BASE = BUF_SDRAM_BASE-(BUF_SRAM_BASE*DL_DS_RATIO)
Data = DL_REL_BASE + (HANDLE & 0xFFFFFFF) * DL_DS_RATIO
```

DL_DS_RATIO and DL_REL_BASE are constants which can be defined at compile or load time so that the assignment of Data can be coded in two instructions.

Microengine Assembler Syntax

```
#macro dl_buf_get_data[sdram_offset, buf_handle]
```

Input

`buf_handle` The buffer handle.

Output

`sdram_offset` The SDRAM address offset from `buf_handle` to the packet buffer.

Estimated Size

Two to four instructions.

2.2.1.6 `dl_buf_get_data_from_meta[]`

This macro returns the SDRAM offset for a given SRAM offset.

This macro can be used in place of `dl_buf_get_data[]` in repetitive calls where both SRAM offset and SDRAM offset are used. Instead of taking a buffer handle as an input parameter, it takes the SRAM offset where the buffer meta data is present. Calling `dl_buf_get_desc[]` with `buf_handle` as an argument returns the SRAM offset and saves two instructions.

See `dl_buf_get_data[]` for further details.

Microengine Assembler Syntax

```
#macro dl_buf_get_data_from_meta[sdram_offset, sram_offset]
```


Input

`sram_offset` The SRAM offset where the meta data for this buffer is present.

Output

`sram_offset` The SDRAM address where the packet buffer for this buffer is present.

Estimated Size

One to two instructions.

2.2.1.7 `dl_meta_init_cache[]`

Initializes the meta data cache with the given values.

Microengine Assembler Syntax

```
#macro dl_meta_init_cache[d0, d1, d2, d3, d4, d5, d6, d7]
```

Input

`d0` The values in registers for `LW0` to `LW7` of the meta data. When a given field
or is to be ignored, pass a constant value of zero for the argument
`d7` corresponding to that field.

Estimated Size

One to seven instructions.

2.2.1.8 `dl_meta_flush_cache[]`

Flushes the cache of meta data in a general-purpose register to SRAM.

Microengine Assembler Syntax

```
#macro dl_meta_flush_cache[wxfex_prefix, buf_handle, req_sig, \
    sig_action, START_LW, NUM_LW]
```

Input

<code>wxfex_prefix</code>	The write transfer register prefix for use by the XBUF macros. (For details, see the <i>Intel® Internet Exchange Architecture Optimized Data Plane Libraries Reference Manual</i> located on the IXA SDK Tools CD.) At least <code>NUM_LW</code> write transfer registers must be allocated prior to invoking this macro.
<code>buf_handle</code>	The buffer handle.
<code>req_sig</code>	The signal to be used in the I/O operation.
<code>sig_action</code>	The flag that determines how this operation responds to the I/O operation. <ul style="list-style-type: none"> • If this argument contains a signal expression, wait for the specified signal(s) • If this argument contains <code>SIG_NONE</code>, do not wait—simply return from this call For details, see the <i>Intel® Internet Exchange Architecture Optimized Data Plane Libraries Reference Manual</i> located on the IXA SDK Tools CD.
<code>START_LW</code>	The start word—a constant.
<code>NUM_LW</code>	The number of longwords to flush—a constant.

Estimated Size

Three to twelve instructions.

2.2.1.9 dl_meta_get_hw_next[]

Gets the next pointer in the meta data. In the case of chained (linked list) buffer, this next pointer indicates the next buffer in the chain. The format of this pointer (eop, sop, cellcount (6 bits), offset (24 bits)) is such that it can be directly used by the SRAM Q-Array to queue this buffer.

Microengine Assembler Syntax

```
#macro dl_meta_get_hw_next[hw_next]
```

Output

hw_next	The next pointer (32 bits). The format of this pointer should be as specified above.
---------	--

2.2.1.10 dl_meta_set_hw_next[]

Sets the next pointer in the meta data. In the case of chained (linked list) buffer, this next pointer indicates the next buffer in the chain. The format of this pointer (eop, sop, cellcount (6 bits), offset (24 bits)) is such that it can be directly used by the SRAM Q-Array to queue this buffer.

Microengine Assembler Syntax

```
#macro dl_meta_set_hw_next[hw_next]
```

Output

hw_next	The next pointer (32 bits). The format of this pointer should be as specified above.
---------	--

2.2.1.11 dl_meta_load_cache[]

Loads the meta data from SRAM and caches it in registers.

Microengine Assembler Syntax

```
#macro dl_meta_load_cache[buffer_handle, dl_meta, signal_number, \
    START_LW, NUM_LW]
```


Input

buffer_handle	The buffer handle specifying the buffer containing the metadata to load into a read transfer register.
signal_number	The signal number to use in I/O.
START_LW	The starting longword within the buffer specifying the metadata to load.
NUM_LW	The number of longwords of metadata to load.

Output

dl_meta	Read transfer register where meta data is loaded.
---------	---

Estimated Size

At most two instructions.

2.2.1.12 dl_meta_get_buffer_next[]

Returns the next pointer in the meta data. In case of a chained (linked list) buffer this next pointer indicates the next buffer in the chain. The format of this pointer is such that it can be directly used by the SRAM Q-array to queue this buffer.

Microengine Assembler Syntax

```
#macro dl_meta_get_buffer_next[hw_next]
```

Output

hw_next The next pointer—32 bits in length. The format of this pointer should be:

- SRAM Offset—24 bits
- Cellcount—6 bits
- SOP bit
- EOP bit

[illegible]

Estimated Size

N/A

2.2.1.13 dl_meta_set_buffer_next[]

Sets the next pointer in the meta data. In the case of a chained (linked list) buffer, this next pointer indicates the next buffer in the chain. The format of this pointer is such that it can be directly used by the SRAM Q-array to queue this buffer.

Microengine Assembler Syntax

```
#macro dl_meta_set_buffer_next[hw_next]
```

Input

hw_next	<p>The next pointer—32 bits total length. The format of this pointer is:</p> <ul style="list-style-type: none"> • EOP bit • SOP bit • Cellcount—6 bits • Offset—24 bits
---------	---

Estimated Size

N/A

2.2.1.14 dl_meta_get_offset[]

Returns the offset within the buffer where the packet data begins. Typically the data is not at an offset of zero relative to the start of the buffer. There is some headroom at the beginning of these buffers so that headers can be easily prepended.

Microengine Assembler Syntax

```
#macro dl_meta_get_offset[offset]
```

Output

offset	<p>The offset from the start of the buffer to the element where the packet data begins.</p>
--------	---

Estimated Size

One instruction.

2.2.1.15 dl_meta_set_offset[]

Sets the offset within the buffer where the packet data begins. Typically the data is not at an offset of zero relative to the start of the buffer. There is some headroom at the beginning of these buffers so that headers can be easily prepended.

Microengine Assembler Syntax

```
#macro dl_meta_set_offset[offset]
```

Input

`offset` The offset from the start of the buffer where the packet data begins.

Estimated Size

One instruction.

2.2.1.16 dl_meta_get_free_list[]

Returns the free list—buffer pool—from which the buffer pointed to by `dl_buf_handle` was allocated.

Microengine Assembler Syntax

```
#macro dl_meta_get_free_list[free_list]
```

Output

`free_list` The free list ID—a 4-bit value.

Estimated Size

One instruction.

2.2.1.17 **dl_meta_set_free_list[]**

Sets the free list (buffer pool) from which this buffer was allocated.

Microengine Assembler Syntax

```
#macro dl_meta_set_free_list[free_list]
```

Input

free_list The free list ID—a 4-bit quantity.

Estimated Size

Two instructions.

2.2.1.18 **dl_meta_get_rx_stat[]**

Returns the receive status for this buffer.

Microengine Assembler Syntax

```
#macro dl_meta_get_rx_stat[rx_stat]
```

Output

rx_stat Receive status—a 4-bit value.

Estimated Size

One instruction.

2.2.1.19 dl_meta_set_rx_stat[]

Sets the receive status (meta data) for this buffer.

Microengine Assembler Syntax

```
#macro dl_meta_set_rx_stat[rx_stat]
```

Input

rx_stat Receive status—a 4-bit value.

Estimated Size

Two instructions.

2.2.1.20 dl_meta_get_buffer_size[]

Returns the buffer size. Buffer size refers to the length of the data present in this buffer and *only* in this buffer.

Microengine Assembler Syntax

```
#macro dl_meta_get_buffer_size[buf_len]
```

Output

buf_len Length of the data in the buffer—a 16-bit value.

Estimated Size

One instruction.

2.2.1.21 `dl_meta_set_buffer_size[]`

Sets the buffer size. Buffer size refers to the length of the data present in this buffer and *only* in this buffer.

Microengine Assembler Syntax

```
#macro dl_meta_set_buffer_size[buf_len]
```

Input

<code>buf_len</code>	The length of the data in the buffer—a 16-bit value.
----------------------	--

Estimated Size

One instruction.

2.2.1.22 `dl_meta_get_input_port[]`

Returns the input port through which this packet was received.

Microengine Assembler Syntax

```
#macro dl_meta_get_input_port[input_port]
```

Output

<code>input_port</code>	The input port number—a 16-bit value.
-------------------------	---------------------------------------

Estimated Size

One instruction.

2.2.1.23 dl_meta_set_input_port[]

Sets the input port through which packets are received.

Microengine Assembler Syntax

```
#macro dl_meta_set_input_port[input_port]
```

Input

input_port The input port number—a 16-bit value.

Estimated Size

Two instructions.

2.2.1.24 dl_meta_get_packet_size[]

Returns the packet size. Packet size refers to the total length of the data present across a chain of buffers. A large packet is split across multiple buffers with each buffer linked to the next. If the packet is small enough to be present in a single buffer, then this value is the same as `dl_get_buf_size`.

Microengine Assembler Syntax

```
#macro dl_meta_get_packet_size[pkt_len]
```

Output

pkt_len Total length of the data in the packet—a 16-bit value.

Estimated Size

One instruction.

2.2.1.25 `dl_meta_set_packet_size[]`

Sets the packet size. Packet size refers to the total length of data present across a chain of buffers. A large packet is split across multiple buffers with each buffer linked to the next. If the packet is small enough to be present in a single buffer, then this value is the same as the buffer size.

Microengine Assembler Syntax

```
#macro dl_meta_set_packet_size[pkt_len]
```

Input

<code>pkt_len</code>	Total length of the data in the packet—a 16-bit value.
----------------------	--

Estimated Size

One instruction.

2.2.1.26 `dl_meta_get_nexthop_id[]`

Returns the next hop ID.

Microengine Assembler Syntax

```
#macro dl_meta_get_nexthop_id[nexthop]
```

Output

<code>nexthop</code>	The ID of the next hop—a 16-bit value.
----------------------	--

Estimated Size

One instruction.

2.2.1.27 `dl_meta_set_nexthop_id[]`

Sets the next hop ID.

Microengine Assembler Syntax

```
#macro dl_meta_set_nexthop_id[nexthop]
```

Input

`nexthop` The ID of the next hop—a 16-bit value.

Estimated Size

One instruction.

2.2.1.28 `dl_meta_get_output_port[]`

Returns the output port destination. For a two-chip configuration, this is the destination port on the egress processor.

Microengine Assembler Syntax

```
#macro dl_meta_get_output_port[oport]
```

Output

`oport` Output port for egress—a 16-bit value.

Estimated Size

One instruction.

2.2.1.29 dl_meta_set_output_port[]

Sets the output port number for egress.

Microengine Assembler Syntax

```
#macro dl_meta_set_output_port[oport_egress]
```

Input

oport Output port—a 16-bit value.

Estimated Size

Two instructions.

2.2.1.30 dl_meta_get_fabric_port[]

Returns the output port for the switch fabric. This is also called the blade ID—when multiple line cards/blades are connected through a switch fabric this ID refers to the blade ID.

Microengine Assembler Syntax

```
#macro dl_meta_get_fabric_port[oport_fabric]
```

Output

oport_fabric The output port for the switch fabric—an 8-bit value.

Estimated Size

One instruction.

2.2.1.31 dl_meta_set_fabric_port[]

Sets the output port for the switch fabric. This is also called the blade ID.

Microengine Assembler Syntax

```
#macro dl_meta_set_fabric_port[oport_fabric]
```

Input

`oport_fabric` The output port for the switch fabric—an 8-bit value.

Estimated Size

One instruction.

2.2.1.32 dl_meta_get_flow_id[]

Returns the flow ID.

Microengine Assembler Syntax

```
#macro dl_meta_get_flow_id[flow_id]
```

Output

`flow_id` The flow ID—a 32-bit value.

Estimated Size

One instruction.

2.2.1.33 dl_meta_set_flow_id[]

Sets the flow ID.

Microengine Assembler Syntax

```
#macro dl_meta_set_flow_id[flow_id]
```

Input

flow_id The flow ID—a 32-bit value.

Estimated Size

One instruction

2.2.1.34 dl_meta_get_class_id[]

Returns the class ID.

Microengine Assembler Syntax

```
#macro dl_meta_get_class_id[class_id]
```

Output

class_id The class ID—a 16-bit value.

Estimated Size

One instruction.

2.2.1.35 `dl_meta_set_class_id[]`

Sets the class ID.

Microengine Assembler Syntax

```
#macro dl_meta_set_class_id[class_id]
```

Input

`class_id` The class ID—a 16-bit value.

Estimated Size

One instruction.

2.2.1.36 `dl_buf_set_SOP[]`

Sets the Start-of-Packet (SOP) bit in `buf_handle` indicating that this buffer contains the SOP.

Microengine Assembler Syntax

```
#macro dl_buf_set_SOP[ ]
```

Output

`buf_handle` The buffer handle.

Estimated Size

One instruction.

2.2.1.37 `dl_buf_set_EOP[]`

Sets the End-of-Packet (EOP) bit in the `buf_handle` indicating that this buffer contains the EOP.

Microengine Assembler Syntax

```
#macro dl_buf_set_EOP[buf_handle]
```

Input

`buf_handle` The buffer handle.

Estimated Size

One instruction.

2.2.1.38 `dl_buf_get_cell_count[]`

Get the cellcount in `buf_handle` indicating how many cells are in this buffer.

Microengine Assembler Syntax

```
#macro dl_buf_get_cell_count[buf_handle, cell_count]
```

Input

`buf_handle` The buffer handle.

Output

`cell_count` Cell Count. (0 -> 1 cell, 1->2 cells, etc.) The maximum cell count is 64.

Estimated Size

Two instructions.

2.2.1.39 `dl_buf_set_cell_count[]`

Sets the cellcount in `buf_handle` indicating how many cells are in this buffer.

Note: Application code can safely assume that bits 29 through 24 in `buf_handle` are already zero, as guaranteed by `dl_buf_alloc[]`, and so an application can save one instruction by not clearing these bits to zero before setting them to cell count.

Microengine Assembler Syntax

```
#macro dl_buf_set_cell_count[buf_handle, cell_count]
```

Input

<code>buf_handle</code>	The buffer handle.
<code>cell_count</code>	The cell count to set in the buffer handle.

Estimated Size

Two instructions.

2.2.1.40 `dl_set_exception[]`

Sets the exception code.

Microengine Assembler Syntax

```
#macro dl_set_exception[block_id, exception_code]
```

Input

<code>block_id</code>	The ID of the calling microblock.
<code>exception_code</code>	The exception code.

Estimated Size

N/A

2.2.1.41 `dl_meta_get_nexthop_id_type[]`

Returns the next hop ID type which specifies the appropriate lookup table to use to lookup the next hop ID.

Microengine Assembler Syntax

```
#macro dl_meta_get_nexthop_id_type[nhid_type]
```


Input

`nhid_type` The next hop ID type—a 4-bit value.

Estimated Size

One instruction.

2.2.1.42 `dl_meta_set_nexthop_id_type[]`

Sets the next hop ID type which specifies the appropriate lookup table to use to lookup the next hop ID.

Microengine Assembler Syntax

```
#macro dl_meta_set_nexthop_id_type[nhid_type]
```

Output

`nhid_type` The next hop ID type—a 4-bit value.

Estimated Size

At most two instructions.

2.2.2 Dispatch Loop API Functions for Extended Meta Data

This section describes the API functions available to access extended meta data.

The following sets of macros operate on extended meta data which includes the child and parent buffer formats. All the macros take the format type value as an argument. Valid values for format type are:

- `META_TYPE_IGNORE` – This value is specified when accessing certain fields in extended meta data layout whose layout is the same.
- `META_PARENT_PACKET_MODE` – When this value is specified, the underlying meta data is assumed to be parent meta data in `PACKET` mode.
- `META_PARENT_CELL_MODE` - When this value is specified, the underlying meta data is assumed to be parent meta data in `CELL` mode.
- `META_CHILD_PACKET_MODE` - When this value is specified, the underlying meta data is assumed to be child meta data in `PACKET` mode.
- `META_CHILD_CELL_MODE` - When this value is specified, the underlying meta data is assumed to be child meta data in `CELL` mode.

The macros also perform a validity check on the type value passed when applicable. It is assumed that the caller has enough knowledge about the type of meta data being operated on. As a result, the macros are invoked by specifying the appropriate format type.

2.2.2.1 `dl_meta_parent_get_ref_cnt[]`

Obtains the reference count value from the parent meta data.

Microengine Assembler Syntax

```
#macro dl_meta_parent_get_ref_cnt[MODE, cnt]
```

Input

MODE This argument is ignored. The caller can specify META_TYPE_IGNORE.

Output

cnt The reference count.

Estimated Size

One instruction.

2.2.2.2 `dl_meta_child_get_child_offset[]`

Obtains the offset in bytes of the start of data in the child buffer. All copy specific data is stored in child buffers.

Microengine Assembler Syntax

```
#macro dl_meta_child_get_child_offset[MODE, child_offset]
```

Input

MODE Either META_CHILD_CELL_MODE or META_CHILD_PACKET_MODE.

Output

child_offset The offset in bytes.

Estimated Size

One instruction.

2.2.2.3 `dl_meta_child_set_child_offset[]`

Sets the offset in bytes of the start of the data in the child buffer.

Microengine Assembler Syntax

```
#macro dl_meta_child_set_child_offset[MODE, child_offset]
```

Input

`MODE` Either `META_CHILD_CELL_MODE` or `META_CHILD_PACKET_MODE`.

`child_offset` The start of data in the child buffer, in bytes.

Estimated Size

Three instructions.

2.2.2.4 `dl_meta_child_get_child_buffer_size[]`

Obtains the size of data in child buffer in bytes. This can be viewed as the size of copy specific data added to the packet.

Microengine Assembler Syntax

```
#macro dl_meta_child_get_child_buffer_size[MODE, buf_size]
```

Input

`MODE` Either `META_CHILD_CELL_MODE` or `META_CHILD_PACKET_MODE`.

Output

`buf_size` The start of data in the child buffer, in bytes.

Estimated Size

One instruction.

2.2.2.5 `dl_meta_child_set_child_buffer_size[]`

Sets the size of the data in the child buffer in bytes.

Microengine Assembler Syntax

```
#macro dl_meta_child_set_child_buffer_size(MODE, buf_size)
```

Input

MODE Either META_CHILD_CELL_MODE or META_CHILD_PACKET_MODE.

buf_size The size of data in the child buffer, in bytes.

Estimated Size

Three instructions.

2.2.2.6 dl_meta_child_get_child_freelist_id[]

Obtains the freelist ID of the child buffer.

Microengine Assembler Syntax

```
#macro dl_meta_child_get_child_freelist_id(MODE, freelist_id)
```

Input

MODE Either META_CHILD_CELL_MODE or META_CHILD_PACKET_MODE.

Output

freelist_id The freelist ID of the child buffer.

Estimated Size

One instruction.

2.2.2.7 dl_meta_child_set_child_freelist_id[]

Sets the freelist ID of the child buffer in the meta data.

Microengine Assembler Syntax

```
#macro dl_meta_child_set_child_freelist_id(MODE, freelist_id)
```


Input

MODE	Either META_CHILD_CELL_MODE or META_CHILD_PACKET_MODE.
freelist_id	The freelist ID of the child buffer.

Estimated Size

Three instructions.

2.2.2.8 dl_meta_child_get_parent_offset[]

Obtains the offset in bytes of the start of data in the parent buffer pointed by the child buffer. This buffer contains the original packet data.

Microengine Assembler Syntax

```
#macro dl_meta_child_get_parent_offset [MODE, parent_offset]
```

Input

MODE	This argument is ignored. The caller can specify META_TYPE_IGNORE.
------	--

Output

parent_offset	The start of data in the parent buffer, in bytes.
---------------	---

Estimated Size

One instruction.

2.2.2.9 dl_meta_child_set_parent_offset[]

Sets the offset in bytes of the start of data in the parent buffer pointed by the child buffer.

Microengine Assembler Syntax

```
#macro dl_meta_child_set_parent_offset [MODE, parent_offset]
```


Input

`MODE` This argument is ignored. The caller can specify `META_TYPE_IGNORE`.

`parent_offset` The start of data in the parent buffer, in bytes.

Estimated Size

One instruction.

2.2.2.10 dl_meta_child_get_parent_buffer_size[]

Obtains the size in bytes of the data in the parent buffer pointed by the child buffer. This buffer contains the original packet data.

Microengine Assembler Syntax

```
#macro dl_meta_child_get_parent_buffer_size [MODE, buf_size]
```

Input

`MODE` This argument is ignored. The caller can specify `META_TYPE_IGNORE`.

Output

`buf_size` The size of the parent buffer, in bytes.

Estimated Size

One instruction.

2.2.2.11 dl_meta_child_set_parent_buffer_size[]

Sets the size in bytes of the data in the parent buffer pointed by the child buffer.

Microengine Assembler Syntax

```
#macro dl_meta_child_set_parent_buffer_size [MODE, buf_size]
```


Input

MODE	This argument is ignored. The caller can specify META_TYPE_IGNORE.
buf_size	The size of the parent buffer, in bytes.

Estimated Size

One instruction.

2.2.2.12 dl_meta_child_get_header_type[]

Obtains the header type of the packet.

Microengine Assembler Syntax

```
#macro dl_meta_child_get_header_type [MODE, header_type]
```

Input

MODE	This argument is ignored. The caller can specify META_TYPE_IGNORE.
------	--

Output

header_type	The packet header type. If child buffer data exists, this is the child buffer header type. Otherwise, this is the parent buffer header type.
-------------	--

Estimated Size

One instruction.

2.2.2.13 dl_meta_child_set_header_type[]

Sets the header type of the packet.

Microengine Assembler Syntax

```
#macro dl_meta_child_set_header_type [MODE, header_type]
```


Input

MODE	This argument is ignored. The caller can specify META_TYPE_IGNORE.
header_type	The packet header type. If child buffer data exists, this is the child buffer header type. Otherwise, this is the parent buffer header type.

Estimated Size

One instruction.

2.2.2.14 dl_meta_child_get_parent_free_list[]

Obtains the free list ID of the parent buffer pointed by the child buffer.

Microengine Assembler Syntax

```
#macro dl_meta_child_get_parent_free_list [MODE, free_list]
```

Input

MODE	This argument is ignored. The caller can specify META_TYPE_IGNORE.
------	--

Output

free_list	The freelist ID of the parent buffer.
-----------	---------------------------------------

Estimated Size

One instruction.

2.2.2.15 dl_meta_child_set_parent_free_list[]

Sets the free list ID of the parent buffer pointed by the child buffer.

Microengine Assembler Syntax

```
#macro dl_meta_child_set_parent_free_list [MODE, free_list]
```


Input

MODE	This argument is ignored. The caller can specify META_TYPE_IGNORE.
free_list	The freelist ID of the parent buffer.

Estimated Size

One instruction.

2.2.2.16 dl_meta_child_get_rx_stat[]

Obtains the receive status of the packet.

Microengine Assembler Syntax

```
#macro dl_meta_child_get_rx_stat [MODE, rx_stat]
```

Input

MODE	This argument is ignored. The caller can specify META_TYPE_IGNORE.
------	--

Output

rx_stat	The packet receive status. This field also includes the receive status of the original packet.
---------	--

Estimated Size

One instruction.

2.2.2.17 dl_meta_child_set_rx_stat[]

Sets the receive status of the packet.

Microengine Assembler Syntax

```
#macro dl_meta_child_set_rx_stat [MODE, rx_stat]
```


Input

MODE	This argument is ignored. The caller can specify META_TYPE_IGNORE.
rx_stat	The packet receive status.

Estimated Size

One instruction.

2.2.2.18 dl_meta_child_get_packet_size[]

Obtains the size in bytes of the packet across all buffers, including the child buffer.

Microengine Assembler Syntax

```
#macro dl_meta_child_get_packet_size [MODE, pkt_size]
```

Input

MODE	This argument is ignored. The caller can specify META_TYPE_IGNORE.
------	--

Output

pkt_size	The size of the packet, in bytes.
----------	-----------------------------------

Estimated Size

One instruction.

2.2.2.19 dl_meta_child_set_packet_size[]

Sets the size in bytes of the packet across all buffers, including the child buffer.

Microengine Assembler Syntax

```
#macro dl_meta_child_set_packet_size [MODE, pkt_size]
```

Input

MODE	This argument is ignored. The caller can specify META_TYPE_IGNORE.
pkt_size	The size of the packet, in bytes.

Estimated Size

One instruction.

2.2.2.20 `dl_meta_child_get_output_port[]`

Obtains the output port number for this packet.

Microengine Assembler Syntax

```
#macro dl_meta_child_get_output_port [MODE, out_port]
```

Input

MODE This argument is ignored. The caller can specify META_TYPE_IGNORE.

Output

out_port The output port number.

Estimated Size

One instruction.

2.2.2.21 `dl_meta_child_set_output_port[]`

Sets the output port number for this packet.

Microengine Assembler Syntax

```
#macro dl_meta_child_set_output_port [MODE, out_port]
```

Input

MODE This argument is ignored. The caller can specify META_TYPE_IGNORE.

out_port The output port number.

Estimated Size

One instruction.

2.2.2.22 `dl_meta_child_get_input_port[]`

Obtains the input port number for this packet.

Microengine Assembler Syntax

```
#macro dl_meta_child_get_input_port [MODE, in_port]
```

Input

MODE This argument is ignored. The caller can specify META_TYPE_IGNORE.

Output

in_port The input port number.

Estimated Size

One instruction.

2.2.2.23 dl_meta_child_set_input_port[]

Sets the input port number for this packet.

Microengine Assembler Syntax

```
#macro dl_meta_child_set_input_port [MODE, in_port]
```

Input

MODE This argument is ignored. The caller can specify META_TYPE_IGNORE.

in_port The input port number.

Estimated Size

One instruction.

2.2.2.24 dl_meta_child_get_nexthop_id[]

Obtains the next hop ID for this packet.

Microengine Assembler Syntax

```
#macro dl_meta_child_get_nexthop_id [MODE, nhid]
```


Input

MODE This argument is ignored. The caller can specify META_TYPE_IGNORE.

Output

nhid The next hop ID.

Estimated Size

One instruction.

2.2.2.25 dl_meta_child_set_nexthop_id[]

Sets the next hop ID for this packet.

Microengine Assembler Syntax

```
#macro dl_meta_child_set_nexthop_id [MODE, nhid]
```

Input

MODE This argument is ignored. The caller can specify META_TYPE_IGNORE.

nhid The next hop ID.

Estimated Size

One instruction.

2.2.2.26 dl_meta_child_get_fabric_port[]

Obtains the fabric port number for this packet.

Microengine Assembler Syntax

```
#macro dl_meta_child_get_fabric_port [MODE, fabric_port]
```

Input

MODE This argument is ignored. The caller can specify META_TYPE_IGNORE.

Output

`fabric_port` The fabric port number.

Estimated Size

One instruction.

2.2.2.27 `dl_meta_child_set_fabric_port[]`

Sets the fabric port number for this packet.

Microengine Assembler Syntax

```
#macro dl_meta_child_set_fabric_port [MODE, fabric_port]
```

Input

`MODE` This argument is ignored. The caller can specify `META_TYPE_IGNORE`.

`fabric_port` The fabric port number.

Estimated Size

One instruction.

2.2.2.28 `dl_meta_child_get_nexthop_id_type[]`

Obtains the nexthop ID type for this packet. This type indicates how the next hop ID field has to be interpreted.

Microengine Assembler Syntax

```
#macro dl_meta_child_get_nexthop_id_type [MODE, nhid_type]
```

Input

`MODE` This argument is ignored. The caller can specify `META_TYPE_IGNORE`.

Output

`nhid_type` The next hop ID type.

Estimated Size

One instruction.

2.2.2.29 `dl_meta_child_set_nexthop_id_type[]`

Sets the next hop ID type for this packet.

Microengine Assembler Syntax

```
#macro dl_meta_child_set_nexthop_id_type [MODE, nhid_type]
```

Input

MODE This argument is ignored. The caller can specify META_TYPE_IGNORE.

nhid_type The next hop ID type.

Estimated Size

One instruction.

2.2.2.30 `dl_meta_child_get_flow_id[]`

Obtains the flow ID for this packet. This is typically obtained after packet classification and is stored in meta data.

Microengine Assembler Syntax

```
#macro dl_meta_child_get_flow_id [MODE, flow_id]
```

Input

MODE This argument is ignored. The caller can specify META_TYPE_IGNORE.

Output

flow_id The flow identifier.

Estimated Size

One instruction.

2.2.2.31 `dl_meta_child_set_flow_id[]`

Sets the flow ID for this packet.

Microengine Assembler Syntax

```
#macro dl_meta_child_set_flow_id [MODE, flow_id]
```

Input

MODE This argument is ignored. The caller can specify META_TYPE_IGNORE.

flow_id The flow identifier.

Estimated Size

One instruction.

2.2.2.32 dl_meta_child_get_color[]

Obtains the color of this packet. This is typically obtained after the packet is flow metered and is stored in meta data.

Microengine Assembler Syntax

```
#macro dl_meta_child_get_color [MODE, color]
```

Input

MODE This argument is ignored. The caller can specify META_TYPE_IGNORE.

Output

color The color of the packet.

Estimated Size

One instruction.

2.2.2.33 dl_meta_child_set_color[]

Sets the color of this packet.

Microengine Assembler Syntax

```
#macro dl_meta_child_set_color [MODE, color]
```


Input

MODE	This argument is ignored. The caller can specify META_TYPE_IGNORE.
color	The color of the packet.

Estimated Size

One instruction.

2.2.2.34 dl_meta_child_get_class_id[]

Obtains the class ID of this packet. This is typically obtained after the packet classification and is stored in meta data.

Microengine Assembler Syntax

```
#macro dl_meta_child_get_class_id [MODE, class_id]
```

Input

MODE	This argument is ignored. The caller can specify META_TYPE_IGNORE.
------	--

Output

class_id	The class identifier.
----------	-----------------------

Estimated Size

One instruction.

2.2.2.35 dl_meta_child_set_class_id[]

Sets the class ID of this packet.

Microengine Assembler Syntax

```
#macro dl_meta_child_set_class_id [MODE, class_id]
```


Input

MODE	This argument is ignored. The caller can specify META_TYPE_IGNORE.
class_id	The class identifier.

Estimated Size

One instruction.

2.2.2.36 dl_meta_child_get_parent_buffer_id[]

Obtains the parent buffer ID to which this child buffer is linked.

Microengine Assembler Syntax

```
#macro dl_meta_child_get_parent_buffer_id [MODE, buf_id]
```

Input

MODE	This argument is ignored. The caller can specify META_TYPE_IGNORE.
------	--

Output

buf_id	The parent buffer identifier.
--------	-------------------------------

Estimated Size

One instruction.

2.2.2.37 dl_meta_child_set_parent_buffer_id[]

Sets the parent buffer ID to which this child buffer is linked.

Microengine Assembler Syntax

```
#macro dl_meta_child_set_parent_buffer_id [MODE, buf_id]
```

Input

MODE	This argument is ignored. The caller can specify META_TYPE_IGNORE.
buf_id	The parent buffer identifier.

Estimated Size

Four instructions (maximum).

2.2.2.38 `dl_meta_child_get_buffer_next[]`

Obtains the next buffer handle for this child buffer.

Microengine Assembler Syntax

```
#macro dl_meta_child_get_buffer_next [MODE, buf_next]
```

Input

MODE Must be META_CHILD_CELL_MODE.

Output

buf_next The next buffer in the chain.

Estimated Size

One instruction.

2.2.2.39 `dl_meta_child_set_buffer_next[]`

Sets the next buffer handle for this child buffer.

Microengine Assembler Syntax

```
#macro dl_meta_child_set_buffer_next [MODE, buf_next]
```

Input

MODE Must be META_CHILD_CELL_MODE.

buf_next The next buffer in the chain.

Estimated Size

One instruction.

2.2.2.40 `dl_meta_child_get_packet_next[]`

Obtains the next packet handle for this child buffer. This is not required to be processed by software as the Q-array hardware manipulates this field.

Microengine Assembler Syntax

```
#macro dl_meta_child_get_packet_next [MODE, pkt_next]
```

Input

MODE Must be META_CHILD_PACKET_MODE.

Output

pkt_next The next packet handle, in hierarchical queueing.

Estimated Size

One instruction.

2.2.2.41 dl_meta_child_set_packet_next[]

Sets the next packet handle for this child buffer.

Microengine Assembler Syntax

```
#macro dl_meta_child_set_packet_next [MODE, pkt_next]
```

Input

MODE Must be META_CHILD_PACKET_MODE.

pkt_next The next packet handle, in hierarchical queueing.

Estimated Size

One instruction.

The Resource Manager is used as a programming interface between Intel XScale® core applications and microcode running on the microengines of the Intel® IXP2400 and IXP2800 Network Processors.

The Resource Manager functionality includes:

- Hardware resource allocation, initialization and configuration for
 - Memory—SRAM, DRAM, Scratch, and Local Memory
 - Hardware Queues and Rings
- Microengine Management including
 - Loading
 - Patching symbols
 - Enable
 - Disable
- Buffer Management
- Communication with Microblocks

Note: This API is *not* backward compatible with the API supported by the IXA SDK 2.0 Resource Manager. In part, this is due to a number of feature changes between the Intel® IXP1200 Network Processor and the Intel® IXP2400 and IXP2800 Network Processors. Also, with the modularization of the SDK infrastructure, the scope and requirements for the IXA SDK Resource Manager API have changed.

The IXA SDK Resource Manager API may be functionally grouped as shown in [Table 3-1](#).

Table 3-1. Resource Manager API Functional Groups

Resource Manager API Group	Description
"System API" on page 70	Functions to initialize and terminate the API, get and set the system hardware configuration, and so on.
"Microengine API" on page 86	Functions for microengine management.
"Hardware Resource Management API" on page 94	Functions to manage hardware rings and queues.
"Buffer Management API" on page 112	Functions for managing buffer freelists and for accessing packet descriptors and data.
"Communication API" on page 134	Functions to communicate with the microengines, other core components in the system, and the peer subsystem in a dual ingress/egress system.
"Remote Communication Extension API" on page 160	Functions to communicate with remote systems.
"Memory Management API" on page 167	Functions to manage non operating system memory.
"System Repository API" on page 187	Functions to centrally manage configuration properties.
"64-Bit Counters API" on page 200	Functions to manage 64-bit provisioning counters.
"Services API" on page 206	Functions for software services including atomic operations, fast memory copy operations, and so on.
"Hash API" on page 220	Functions to support 48-, 64-, and 128-bit hash operations.
"Microengine Services API" on page 227	Functions that coordinate operations between the Intel XScale [®] core and microengines.
"Debug Support API" on page 235	Functions that provide debugging capabilities.

Resource Manager Error Codes are listed in [Table 3-4](#).

3.1 Defined Types, Enumerations, and Data Structures

The Resource Manager defines a set of basic types used across the entire SDK. These types are defined to ease portability of the API across different operating systems, compilers and so on. These types are listed in [Table 3-2](#).

Table 3-2. Basic Types Supported by the Resource Manager

Basic Types	Description
<code>ix_int8</code>	An 8-bit signed integer.
<code>ix_uint8</code>	An 8-bit unsigned integer.
<code>ix_int16</code>	A 16-bit signed integer.
<code>ix_uint16</code>	A 16-bit unsigned integer.
<code>ix_int32</code>	A 32-bit signed integer.
<code>ix_uint32</code>	A 32-bit unsigned integer.
<code>ix_int64</code>	A 64-bit signed integer.
<code>ix_uint64</code>	A 64-bit unsigned integer.
<code>ix_uint128</code>	A 128-bit unsigned integer. This type does not support full arithmetic operations. However a set of macros has been provided to support this type.
<code>ix_bit_mask8</code>	An 8-bit bit mask.
<code>ix_bit_mask16</code>	A 16-bit bit mask.
<code>ix_bit_mask32</code>	A 32-bit bit mask.
<code>ix_bit_mask64</code>	A 64-bit bit mask.
<code>ix_handle</code>	Generic handle type used throughout the framework API. All handle types are aliases of this handle type.
<code>ix_error</code>	Error token used through out the IX SDK 3.0. All IXA SDK 3.0 functions return this type. This is a 32-bit unsigned integer that has packed a 16-bit error code, an 8-bit error group and an 8-bit error level. Macros are provided for creating an error token and for accessing the different fields.
<code>_IX_OS_TYPE_</code>	Preprocessor symbol that indicates for which operating system the SDK is currently compiled. At the moment just four values are defined, but others may be added later: <ul style="list-style-type: none"> <code>_IX_OS_VXWORKS_</code> <code>_IX_OS_LINUX_KERNEL_</code> <code>_IX_OS_LINUX_USER_</code> <code>_IX_OS_WIN32_</code> NOTE: <code>_IX_OS_WIN32_</code> is used for debug with foreign model or for Win32 simulation.

3.2 System API

The Resource Manager System API provides functions to initialize and terminate the Resource Manager, to get and set the hardware configuration, and so on. [Table 3-3](#) lists the data types and functions in this API.

Table 3-3. Resource Manager System API

Name	Description
ix_rm_error_code	The enumerated type listing error numbers specific to the Resource Manager.
ix_phy_type	The enumerated type specifying the values for physical layer interfaces that could be present on the board.
ix_port_type	The enumerated type specifying the different types of physical interface.
ix_port	The structure specifying a type and number for a physical interface.
ix_subsystem_type	For a system composed of an ingress and egress subsystem, this enumerated type defines the subsystem type.
ix_sys_config	The structure specifying system configuration.
ix_memory_reserved_area	Describes a microengine memory area to reserve at initialization time.
ix_rm_init()	Initializes the Resource Manager API.
ix_rm_term()	Terminates the Resource Manager API.
ix_rm_version_get_string()	This function returns the Resource Manager library version information string.
ix_rm_sys_config_get()	Returns the board-specific configuration.
ix_rm_error_get_string()	Returns the error string corresponding to an ix_rm_error_code .
ix_rm_sys_config_set()	Sets the system configuration.

3.2.1 Defined Types, Enumerations, and Data Structures

3.2.1.1 ix_rm_error_code

This enumerated data type lists Resource Manager specific error codes. An error code is obtained from an `ix_error` token by calling the `IX_ERROR_GET_CODE()` macro. See the *Intel® Internet Exchange Architecture (IXA) Software Reference Manual* for details on this macro.

The error code can be passed to the `ix_rm_error_get_string()` function to get an associated error string. Table 3-4 lists error codes that could be returned by the Resource Manager library.

Table 3-4. Error Codes for the Resource Manager (Sheet 1 of 6)

Error Code	Description
IX_RM_ERROR_SUCCESS	Not an error code; this value indicates success.
IX_RM_ERROR_ASSERTION	Assertion error code
Memory Manager error codes	
IX_RM_ERROR_MEMORY_INIT_FAILED	Memory API initialization operation failed.
IX_RM_ERROR_MEMORY_FINI_FAILED	Memory API termination operation failed.
IX_RM_ERROR_BAD_MEMORY_TYPE	Invalid memory type was passed.
IX_RM_ERROR_CANNOT_CREATE_CELL	Internal memory block information structure couldn't be allocated.
IX_RM_ERROR_DATA_ALREADY_RELEASED	Memory was already released.
IX_RM_ERROR_BAD_MEMORY_ADDRESS	Invalid memory address was passed (not in range)
IX_RM_ERROR_MEMORY_IN_USE	Passed memory address is already used by another entity in the system.
IX_RM_ERROR_NULL_MEMORY_ADDRESS	Passed memory address is a null value.
IX_RM_ERROR_CANNOT_ALLOCATE_SIZE	Requested size can't be allocated.
IX_RM_ERROR_CANNOT_RESERVE_MEMORY	Requested amount of memory cannot be reserved due to the fact that part of the requested memory is already in use or the requested size is too large.
IX_RM_ERROR_CELL_SPLIT_FAILED	Internal memory block split operation failed.
IX_RM_ERROR_MEMORY_BAD_OFFSET	Invalid memory offset was passed.
General Library error codes	
IX_RM_ERROR_LIB_INIT_FAILED	Resource Manager library initialization operation didn't complete successfully.
IX_RM_ERROR_LIB_FINI_FAILED	Resource Manager library termination operation didn't complete successfully.
Communication API error codes	
IX_RM_ERROR_PKT_DISP_ENG_INIT_FAILED	Initialization operation for packet dispatch engine failed.
IX_RM_ERROR_MSG_DISP_ENG_INIT_FAILED	Initialization operation for message dispatch engine failed.
IX_RM_ERROR_PKT_DISP_ENG_FINI_FAILED	Termination operation for packet dispatch engine failed.
IX_RM_ERROR_MSG_DISP_ENG_FINI_FAILED	Termination operation for message dispatch engine failed.
IX_RM_ERROR_PKT_DISP_ENG_START_FAILED	Packet dispatch engine start operation failed.
IX_RM_ERROR_MSG_DISP_ENG_START_FAILED	Message dispatch engine start operation failed.

Table 3-4. Error Codes for the Resource Manager (Sheet 2 of 6)

Error Code	Description
IX_RM_ERROR_PKT_DISP_ENG_STOP_FAILED	Packet dispatch engine stop operation failed.
IX_RM_ERROR_MSG_DISP_ENG_STOP_FAILED	Message dispatch engine stop operation failed.
IX_RM_ERROR_COMM_API_INIT_FAILED	Communication API initialization operation didn't complete successfully.
IX_RM_ERROR_COMM_API_FINI_FAILED	Communication API termination operation didn't complete successfully.
IX_RM_ERROR_COMM_QUEUE_FULL	Communication ID internal queue is full.
IX_RM_ERROR_COMM_QUEUE_EMPTY	Communication ID internal queue is empty.
IX_RM_ERROR_COMM_NOTIFICATION_ELEM_ALLOCATION_FAILED	Allocation operation for communication ID notification element failed.
IX_RM_ERROR_COMM_SEM_POOL_INIT_FAILED	Communication API semaphore pool initialization operation failed.
IX_RM_ERROR_COMM_SEM_NODE_ALLOCATION_FAILED	Semaphore pool node allocation operation failed.
IX_RM_ERROR_COMM_SEM_GET_FAILED	Notification semaphore retrieve operation failed.
IX_RM_ERROR_COMM_ISR_SETUP_FAILED	Interrupt service routine (ISR) setup operation failed.
IX_RM_ERROR_COMM_DISPATCH_TH_INIT_FAILED	Dispatch thread engine initialization operation failed.
IX_RM_ERROR_COMM_EP_DATA_ALLOCATION_FAILED	Internal communication ID queue allocation operation failed.
IX_RM_ERROR_COMM_EP_SEMA_CREATION_FAILED	End point synchronization semaphore creation operation failed.
IX_RM_ERROR_COMM_EP_MGR_INIT_FAILED	End point manager initialization operation failed.
IX_RM_ERROR_COMM_EP_MGR_FINI_FAILED	End point manager termination operation failed.
IX_RM_ERROR_COMM_EP_PCI_PROXIES_INIT_FAILED	PCI end point proxies initialization operation failed.
IX_RM_ERROR_COMM_EP_PCI_PROXIES_FINI_FAILED	PCI end point proxies termination operation failed.
IX_RM_ERROR_COMM_EP_REMOTE	Communication ID is remote.
IX_RM_ERROR_COMM_EP_UBLOCK	Communication ID corresponds to a microblock.
IX_RM_ERROR_COMM_EP_CC	Communication ID corresponds to a core component.
IX_RM_ERROR_COMM_EP_RECV_CALLBACK	Communication ID is in callback receive mode.
IX_RM_ERROR_COMM_EP_RECV_GET_SELECT	Communication ID is in get/select receive mode.
IX_RM_ERROR_COMM_EP_MULTIPLE_CONSUMER	Communication ID is in multiple consumer mode.
IX_RM_ERROR_COMM_EP_CC_MODE_ENABLED	Communication ID has core component mode enabled.
IX_RM_ERROR_COMM_EP_CC_MODE_DISABLED	Communication ID has core component mode disabled.
IX_RM_ERROR_REMOTE_COMM_API_INIT_FAILED	Remote communication API initialization operation didn't complete successfully.
IX_RM_ERROR_REMOTE_COMM_API_FINI_FAILED	Remote communication API termination operation didn't complete successfully.
IX_RM_ERROR_REMOTE_COMM_SERVICE_ACTIVE	Another remote communication service is active.
IX_RM_ERROR_COMM_PCI_INIT_FAILED	PCI initialization operation didn't complete successfully.
IX_RM_ERROR_COMM_PCI_FINI_FAILED	PCI termination operation didn't complete successfully.
IX_RM_ERROR_COMM_PCI_GET_REMOTE_DEV_FAILED	PCI remote device retrieve operation failed.
IX_RM_ERROR_COMM_PCI_TX_INIT_FAILED	PCI transmit engine initialization operation didn't complete successfully.

Table 3-4. Error Codes for the Resource Manager (Sheet 3 of 6)

Error Code	Description
IX_RM_ERROR_COMM_PCI_RX_INIT_FAILED	PCI receive engine initialization operation didn't complete successfully.
IX_RM_ERROR_COMM_PCI_TX_START_FAILED	PCI transmit engine start didn't complete successfully.
IX_RM_ERROR_COMM_PCI_RX_START_FAILED	PCI receive engine start didn't complete successfully.
IX_RM_ERROR_COMM_PCI_TX_STOP_FAILED	PCI transmit engine stop didn't complete successfully.
IX_RM_ERROR_COMM_PCI_RX_STOP_FAILED	PCI receive engine stop didn't complete successfully.
IX_RM_ERROR_COMM_PCI_RX_FINI_FAILED	PCI receive termination operation didn't complete successfully.
IX_RM_ERROR_COMM_PCI_TX_FINI_FAILED	PCI transmit termination operation didn't complete successfully.
IX_RM_ERROR_COMM_PCI_INT_CONNECT_FAILED	PCI communication interrupt ISR registration failed.
IX_RM_ERROR_COMM_PCI_INT_ENABLE_FAILED	PCI communication interrupt ISR enable failed.
IX_RM_ERROR_COMM_PCI_INT_DISABLE_FAILED	PCI communication interrupt ISR disable failed.
IX_RM_ERROR_COMM_PCI_RX_TH_INIT_FAILED	PCI receive thread initialization operation failed.
IX_RM_ERROR_COMM_PCI_TX_TH_INIT_FAILED	PCI transmit thread initialization operation failed.
IX_RM_ERROR_COMM_PCI_RX_SEMA_CREATION_FAILED	PCI receive semaphore creation operation failed.
IX_RM_ERROR_COMM_PCI_TX_SEMA_CREATION_FAILED	PCI transmit semaphore creation operation failed.
IX_RM_ERROR_COMM_PCI_TX_CTRL_INFO_ALLOC_FAILED	PCI transmit control information allocation operation failed.
IX_RM_ERROR_COMM_PCI_TX_CTRL_INFO_GET_PHYS_OFFSET_FAILED	Retrieve operation for PCI transmit control info offset failed.
IX_RM_ERROR_COMM_PCI_RX_HW_FL_CREATE_FAILED	PCI receive hardware free list create operation failed.
IX_RM_ERROR_COMM_PCI_RX_SW_FL_CREATE_FAILED	PCI receive software free list create operation failed.
IX_RM_ERROR_COMM_PCI_TX_CTRL_INFO_FREE_FAILED	PCI transmit control information deallocation operation failed.
IX_RM_ERROR_COMM_PCI_TX_SEMA_FINI_FAILED	PCI transmit semaphore termination operation failed.
IX_RM_ERROR_COMM_PCI_RX_SW_FL_DELETE_FAILED	PCI receive software free list delete operation failed.
IX_RM_ERROR_COMM_PCI_RX_HW_FL_DELETE_FAILED	PCI receive hardware free list delete operation failed.
IX_RM_ERROR_COMM_PCI_RX_SEMA_FINI_FAILED	PCI receive semaphore termination operation failed.
IX_RM_ERROR_COMM_PCI_UNSUPPORTED_MEMORY_TYPE	Unsupported memory type was passed for the PCI communication received buffer.
Hardware API error codes	
IX_RM_ERROR_HW_API_INIT_FAILED	Hardware API initialization operation didn't complete successfully.
IX_RM_ERROR_HW_API_FINI_FAILED	Hardware API termination operation didn't complete successfully.
IX_RM_ERROR_HW_QARRAY_MGR_ALLOC_FAILED	QArray manager allocation operation failed, therefore QArray manager is not initialized.
IX_RM_ERROR_HW_QARRAY_MGR_DEALLOC_FAILED	QArray manager deallocation operation failed.
IX_RM_ERROR_HW_QUEUE_CREATE_FAILED	Hardware queue create operation failed.
IX_RM_ERROR_HW_RING_CREATE_FAILED	Hardware scratch ring create operation failed.
IX_RM_ERROR_HW_RING_DELETE_FAILED	Hardware scratch ring delete operation failed.
IX_RM_ERROR_HW_SRAM_RING_PUT_FAILED	SRAM ring put operation failed.

Table 3-4. Error Codes for the Resource Manager (Sheet 4 of 6)

Error Code	Description
IX_RM_ERROR_HW_SCRATCH_RING_PUT_FAILED	Scratch ring put operation failed.
IX_RM_ERROR_HW_SRAM_RING_GET_FAILED	SRAM ring retrieve operation failed.
IX_RM_ERROR_HW_SCRATCH_RING_GET_FAILED	Scratch ring retrieve operation failed.
Buffer API error codes	
IX_RM_ERROR_BUF_API_INIT_FAILED	Buffer API initialization operation didn't complete successfully.
IX_RM_ERROR_BUF_API_FINI_FAILED	Buffer API termination operation didn't complete successfully.
IX_RM_ERROR_BUF_QUEUES_ALLOC_FAILED	Allocation of hardware queues for buffer free lists failed.
IX_RM_ERROR_BUF_HANDLE_INVALID	Invalid handle was passed.
IX_RM_ERROR_BUF_FREE_LIST_INVALID	Invalid free list handle was passed.
IX_RM_ERROR_BUF_FREE_LIST_ID_INVALID	Invalid free list identifier was passed.
IX_RM_ERROR_BUF_FREE_LIST_BAD_SIZE	Invalid size of free list was passed.
IX_RM_ERROR_BUF_FREE_LIST_SIZE_TOO_LARGE	Passed value for free list size is larger than legal value.
IX_RM_ERROR_BUF_FREE_LIST_BAD_META_SIZE	Passed value for free list meta data is invalid.
IX_RM_ERROR_BUF_FREE_LIST_ALL_IN_USE	Free list can't be created because all free lists are already in use.
IX_RM_ERROR_BUF_FREE_LIST_META_ALLOCATION_FAILED	Allocation operation for meta data didn't complete successfully.
IX_RM_ERROR_BUF_FREE_LIST_INTERNAL_META_ALLOCATION_FAILED	Allocation operation for free list internal meta memory block didn't complete successfully. This error is returned only if IX_RM_SPLIT_META_DATA is defined.
IX_RM_ERROR_BUF_FREE_LIST_DEBUG_INFO_ALLOCATION_FAILED	Allocation for free list debugging information memory block didn't complete successfully. This error is returned only if _IX_RM_BUFFER_DEBUG_ is defined.
IX_RM_ERROR_BUF_FREE_LIST_DATA_ALLOCATION_FAILED	Free list data memory block allocation operation failed.
IX_RM_ERROR_BUF_FREE_LIST_FINI_FAILED	Free list termination operation failed.
IX_RM_ERROR_BUF_FREE_LIST_CREATION_FAILED	Free list creation operation failed.
IX_RM_ERROR_BUF_NO_FREE_BUFFERS	Operation didn't complete because no buffers are available.
IX_RM_ERROR_BUF_ALREADY_LINKED	Link operation failed because the first buffer parameter already has a link.
System Repository API error codes	
IX_RM_ERROR_CP_API_INIT_FAILED	System repository API initialization operation didn't complete successfully.
IX_RM_ERROR_CP_API_FINI_FAILED	System repository API termination operation didn't complete successfully.
IX_RM_ERROR_CP_HANDLE_NULL	Passed configuration property handle is null.
IX_RM_ERROR_CP_NAME_TOO_LONG	Passed configuration property name is longer than legal value.
IX_RM_ERROR_CP_NAME_NULL	Passed configuration property name is null.
IX_RM_ERROR_CP_BAD_NAME	Passed configuration property name is invalid.
IX_RM_ERROR_CP_ALREADY_EXISTS	Configuration property cannot be created because the requested name already exists at the same node level.

Table 3-4. Error Codes for the Resource Manager (Sheet 5 of 6)

Error Code	Description
IX_RM_ERROR_CP_CREATION_FAILED	Configuration property creation operation failed.
IX_RM_ERROR_CP_FAILED_TO_ACCESS_ATTRIBUTES	Configuration property attributes can't be accessed.
IX_RM_ERROR_CP_CONSTANT_DELETE	Constant configuration property can't be deleted.
IX_RM_ERROR_CP_CONSTANT_SET_VALUE	Value for constant configuration property can't be set.
IX_RM_ERROR_CP_FAILED_TO_LINK	Configuration property can't be linked.
IX_RM_ERROR_CP_DATA_UINT32	Data type of configuration property can't be changed directly from 32 bit to opaque.
IX_RM_ERROR_CP_DATA_OPAQUE	Data type of configuration property can't be changed directly from opaque to 32 bit.
IX_RM_ERROR_CP_BUFFER_GET_NEXT_FAILED	Retrieval of next configuration property failed.
IX_RM_ERROR_CP_BUFFER_TOO_SMALL	Configuration property internal buffer size is smaller than the passed data.
IX_RM_ERROR_CP_FAILED_TO_ALLOCATE_NOTIFICATION_TOKEN	Configuration property notification token wasn't allocated.
IX_RM_ERROR_CP_NOTIFICATION_ACTIVE	Operation can't be performed while notification is in progress.
IX_RM_ERROR_CP_PROPERTY_NOT_FOUND	Requested configuration property is not in the database.
64-Bit Counter API error codes	
IX_RM_ERROR_64BIT_COUNTER_INIT_FAILED	64-bit counter API initialization operation didn't complete successfully.
IX_RM_ERROR_64BIT_COUNTER_FINI_FAILED	64-bit counter API termination operation didn't complete successfully.
IX_RM_ERROR_64BIT_COUNTER_MEMORY_INIT_FAILED	Memory initialization for 64-bit counter didn't complete successfully.
IX_RM_ERROR_64BIT_COUNTER_BLOCK_INIT_FAILED	Block initialization for 64-bit counter didn't complete successfully.
IX_RM_ERROR_64BIT_COUNTER_ALL_BLOCKS_IN_USE	Block can't be created because all blocks are currently in use.
IX_RM_ERROR_64BIT_COUNTER_INACTIVE	64-bit counter is inactive.
IX_RM_ERROR_64BIT_COUNTER_HANDLE_INVALID	Passed handle is invalid.
IX_RM_ERROR_64BIT_COUNTER_UPDATE_TH_INIT_FAILED	Initialization of 64-bit counter update thread failed.
IX_RM_ERROR_64BIT_COUNTER_UPDATE_SEM_INIT_FAILED	Initialization of 64-bit counter update semaphore failed.
IX_RM_ERROR_64BIT_COUNTER_BAD_OVERFLOW_TIME	Passed value for overflow time is invalid.
IX_RM_ERROR_64BIT_COUNTER_BAD_INTERNAL_MEMORY	Passed value for type of internal memory is invalid.
IX_RM_ERROR_64BIT_COUNTER_NOT_ENOUGH_COUNTERS	Operation failed because too few counters are available.
IX_RM_ERROR_64BIT_COUNTER_SWAP_MEM_ALLOC_FAILED	64-bit counter atomic memory swap operation failed.
Microengine API error codes	
IX_RM_ERROR_ME_API_INIT_FAILED	Microengine API initialization operation didn't complete successfully.
IX_RM_ERROR_ME_API_FINI_FAILED	Microengine API termination operation didn't complete successfully.
IX_RM_ERROR_ME_MAP_UCODE_FAILED	Microcode mapping of UOF code to memory failed.

Table 3-4. Error Codes for the Resource Manager (Sheet 6 of 6)

Error Code	Description
IX_RM_ERROR_ME_LOAD_UCODE_FAILED	Microcode loading of UOF code to memory failed.
IX_RM_ERROR_ME_LOAD_TO_USTORE_FAILED	Microcode loading of UOF code to microstore failed.
IX_RM_ERROR_ME_LOAD_UCODE_FROM_WB_FAILED	Microcode loading of UOF code from workbench failed.
IX_RM_ERROR_ME_START_FAILED	Microengine start operation failed.
IX_RM_ERROR_ME_STOP_FAILED	Microengine stop operation failed.
IX_RM_ERROR_ME_RESET_FAILED	Microengine reset operation failed.
IX_RM_ERROR_ME_ENABLE_CTX_FAILED	Enable operation for microengine contexts failed.
IX_RM_ERROR_ME_DISABLE_CTX_FAILED	Disable operation for microengine contexts failed.
IX_RM_ERROR_ME_REGISTER_INTERNAL_SYMBOLS_FAILED	Internal symbol registration operation failed.
IX_RM_ERROR_ME_PATCH_SYMBOLS_FAILED	Symbol patch to microcode failed.
Hash API error codes	
IX_RM_ERROR_HASH_API_INIT_FAILED	Hash API initialization operation didn't complete successfully.
IX_RM_ERROR_HASH_API_FINI_FAILED	Hash API termination operation didn't complete successfully.
Microengine Services API error codes	
IX_RM_ERROR_ME_XSCALE_LOCK_INIT_FAILED	ME to XScale lock API initialization operation didn't complete successfully.
IX_RM_ERROR_ME_XSCALE_LOCK_FINI_FAILED	ME to XScale lock API API termination operation didn't complete successfully.
IX_RM_ERROR_ME_XSCALE_LOCK_MEM_ALLOC_FAILED	Scratch memory allocation for locks failed.
IX_RM_ERROR_ME_XSCALE_LOCK_OUT_OF_LOCKS	Operation failed because no more locks are available.
IX_RM_ERROR_ME_XSCALE_LOCK_NOT_ACTIVE	Operation failed because passed lock can't be used.
IX_RM_ERROR_ME_XSCALE_LOCK_IS_LOCKED	Operation failed because lock is already enabled.
IX_RM_ERROR_ME_XSCALE_LOCK_BAD_MEMORY	Passed scratch memory value is invalid.
IX_RM_ERROR_ME_XSCALE_LOCK_IS_LOCKED_BY_ME	Operation failed because lock is enabled by microengine.
IX_RM_ERROR_UNKNOWN	Error has no text message associated with it.

3.2.1.2 ix_phy_type

This enumerated type lists the physical layer interfaces recognized by the board.

C Syntax

```
typedef enum ix_e_phy_type {
    IX_PHY_TYPE_FIRST = 0,
    IX_PHY_TYPE_OTHER = IX_PHY_TYPE_FIRST,
    IX_PHY_TYPE_IX_BUS,
    IX_PHY_TYPE_UTOPIA_3,
    IX_PHY_TYPE_SPI_4,
    IX_PHY_TYPE_LAST
} ix_phy_type;
```


Defined Values`IX_PHY_TYPE_OTHER``IX_PHY_TYPE_IX_BUS``IX_PHY_TYPE_UTOPIA_3``IX_PHY_TYPE_SPI_4`

These values specify types for various physical layer interfaces recognized by the Intel® IXP2400 and IXP2800 Network Processors.

3.2.1.3 ix_port_type

This enumerated type lists the link layer interfaces present on the board.

C Syntax

```
typedef enum ix_e_port_type {  
    IX_PORT_TYPE_OTHER = 0,  
    IX_PORT_TYPE_FAST_ETHERNET,  
    IX_PORT_TYPE_GIGABIT_ETHERNET,  
    IX_PORT_TYPE_ATM,  
    IX_PORT_TYPE_POS,  
    IX_PORT_TYPE_FRAME_RELAY,  
    IX_PORT_TYPE_CSIX_FABRIC,  
    IX_PORT_TYPE_SOFTWARE_LOOPBACK,  
    IX_PORT_TYPE_LAST  
} ix_port_type;
```

Defined Values

IX_PORT_TYPE_OTHER	These values specify types for various link layer interfaces supported by the Intel® IXP2400 and IXP2800 Network Processors.
IX_PORT_TYPE_FAST_ETHERNET	
IX_PORT_TYPE_GIGABIT_ETHERNET	
IX_PORT_TYPE_ATM	
IX_PORT_TYPE_POS	
IX_PORT_TYPE_FRAME_RELAY	
IX_PORT_TYPE_CSIX_FABRIC	
IX_PORT_TYPE_SOFTWARE_LOOPBACK	

3.2.1.4 ix_port

This structure represents a link layer interface to the Intel® IXP2400 and IXP2800 Network Processors.

Microengine C Syntax

```
typedef struct ix_s_port
{
    ix_uint32      m_Number;
    ix_port_type   m_vLinkType;
    ix_phy_type    m_vPhyType;
    ix_port_speed  m_PortSpeed;
} ix_port;
```

Data Members

m_vNumber	The port number.
m_vLinkType	The link-layer type. See Section 3.2.1.3 , “ix_port_type.”
m_vPhyType	The physical-layer type. See Section 3.2.1.2 , “ix_phy_type.”
m_PortSpeed	The port speed.

3.2.1.5 ix_subsystem_type

This enumerated type lists the supported subsystem types.

C Syntax

```
typedef enum ix_e_comm_subsystem_type {
    IX_SUBSYSTEM_TYPE_FIRST = 0,
    IX_SUBSYSTEM_TYPE_INGRESS = IX_SUBSYSTEM_TYPE_FIRST,
    IX_SUBSYSTEM_TYPE_EGRESS,
    IX_SUBSYSTEM_TYPE_LAST
} ix_subsystem_type;
```

Defined Values

IX_SUBSYSTEM_TYPE_INGRESS	Specifies an ingress processor.
IX_SUBSYSTEM_TYPE_EGRESS	Specifies an egress processor.

3.2.1.6 ix_sys_config

This structure describes the current system—that is, board—configuration containing the number of ports physically present on the system as well as the number of microengines, amount of SRAM, and number of SRAM channels.

C Syntax

```
typedef struct ix_s_sys_config {  
    /* This section is constant and can not be modified once initialized */  
    ix_uint32      m_SystemId;  
    ix_subsystem_type m_SubsystemType;  
    ix_uint32      m_NumberOfMicroengines;  
    ix_uint32      m_NumberOfSramChannels;  
    ix_uint32      m_NumberOfDramChannels;  
    ix_uint32      m_ChipId;  
    ix_uint32      m_NumberOfPorts;  
    ix_port        m_aPorts[IX_MAX_NUM_PORTS];  
    //  
    /* This is the section that can be modified after initialization */  
    ix_uint32      m_Endianess;      /* 0 for big and 1 little.      */  
} ix_sys_config;
```


Data Members

<code>m_SystemId</code>	The system identifier for a multi-system configuration—a 32-bit unsigned integer.
<code>m_SubsystemType</code>	The subsystem type. See Section 3.2.1.5 , “ ix_subsystem_type .”
<code>m_NumberOfMicroengines</code>	The number of microengines in the system—a 32-bit unsigned integer.
<code>m_NumberOfSramChannels</code>	The number of SRAM memory channels—a 32-bit unsigned integer.
<code>m_NumberOfDramChannels</code>	The number of DRAM memory channels—a 32-bit unsigned integer.
<code>m_ChipId</code>	The chip identifier—a 32-bit unsigned integer.
<code>m_NumberOfPorts</code>	The number of ports physically present in this system—a 32-bit unsigned integer.
<code>m_aPorts[IX_MAX_NUM_PORTS]</code>	The array of physical ports. See Section 3.2.1.4 , “ ix_port .”
<code>m_Endianess</code>	<p>The byte ordering for multibyte words—a 32-bit unsigned integer.</p> <ul style="list-style-type: none"> • 0—big-endian ordering • 1—little-endian ordering <p>NOTE: This element may be changed after initialization.</p>

3.2.1.7 ix_memory_reserved_area

This structure describes a microengine memory area to reserve at initialization time. This allows the Resource Manager to respect this memory used by microcode such that there is no requirement that microcode have any knowledge of the existence of the Resource Manager.

Microengine C Syntax

```
typedef struct ix_s_memory_reserved_area {
    ix_memory_type  m_Type;
    ix_uint32       m_Channel;
    ix_uint32       m_StartOffset;
    ix_uint32       m_Size;
} ix_memory_reserved_area;
```

Data Members

m_Type	The type of memory to reserve.
m_Channel	The channel of the memory to reserve.
m_StartOffset	The offset to the base address for the memory area to reserve—in bytes.
m_Size	The size of the memory area to reserve—in bytes.

3.2.2 API Functions

3.2.2.1 ix_rm_init()

This function initializes the Resource Manager. The calling application must call this function before making any other Resource Manager call. Applications *must* call this function before utilizing any of the Resource Manager services.

The `arg_pReservedAreas` parameter is the beginning of an array listing memory areas that need to be reserved. The end of the array is signaled by an entry of size zero. If this parameter is `NULL` then it signals that there is no memory area that needs to be reserved.

The main reason for the `arg_pReservedAreas` parameter is that Microengine C generated microcode might need some areas of the memory controlled by the Resource Manager for creating variables. As there is no coordination between the Microengine C compiler, the loader, and the Resource Manager two logical modules may access the same memory location with different logic. At the time the microcode is loaded in memory, the Resource Manager checks if any memory areas are needed by the microcode, and if there are such areas, the Resource Manager checks if these areas have been reserved at initialization time.

If areas needed by microcode are not reserved at initialization time an error is signaled and the areas that must be reserved for microcode use are displayed. To correct this condition the calling program must create an array of reserved areas and pass this array to the initialization function.

The first time this function is called in a system, the Resource Manager performs internal initialization. Any other subsequent call to the `ix_rm_init()` function results in an increment of an internal counter with no initialization done. In this case the passed parameter is disregarded. A number of `ix_rm_init()` calls must be followed by the same number of `ix_rm_term()` calls in order to bring the system to its initial state.

Note: If any other Resource Manager call is made prior to this one, the result is unpredictable.

C Syntax

```
ix_error ix_rm_init(const ix_memory_reserved_area* arg_pReservedAreas);
```

Input

<code>arg_pReservedAreas</code>	A pointer to the beginning of an array of memory areas to be reserved at initialization time.
---------------------------------	---

Output/Returns

Return Value	Returns <code>IX_SUCCESS</code> if successful and a valid <code>ix_error</code> otherwise.
--------------	--

3.2.2.2 `ix_rm_term()`

This operation should be called when the Resource Manager services are no longer needed. It provides all necessary cleanup. Applications *must* call this function when services from the Resource Manager are no longer used.

Note: The calling application must not call any Resource Manager functions after this call returns.

C Syntax

```
ix_error ix_rm_term(void);
```

Output

Return Value	Returns <code>IX_SUCCESS</code> if successful and a valid <code>ix_error</code> otherwise.
--------------	--

3.2.2.3 `ix_rm_error_get_string()`

This is a support function that retrieves a string representation of error codes specific to the Resource Manager and specified by `ix_rm_error_code`. Resource Manager specific error codes are passed within the `ix_error` return value for all interface functions. This function can be used to get a string representation of the value of that error code.

C Syntax

```
const char* ix_rm_error_get_string (
    ix_rm_error_code arg_vRmError);
```

Input

<code>ix_rm_error_code</code>	The Resource Manager-specific error code whose string representation is to be returned.
-------------------------------	---

Output

Return Value	The string representation corresponding to the error code.
--------------	--

3.2.2.4 `ix_rm_sys_config_get()`

Returns the board-specific configuration. The calling application is responsible for allocating the `arg_pSysConfig` structure and passing a pointer to it into the function.

Output/Returns

Return Value	Returns <code>IX_SUCCESS</code> if the operation is successful and a valid <code>ix_error</code> value otherwise.
<code>arg_pSysConfig</code>	The location where the system configuration is written.

C Syntax

```
ix_error ix_rm_sys_config_get(
    ix_sys_config* arg_pSysConfig);
```

3.2.2.5 `ix_rm_version_get_string()`

This function returns the library version description—a string.

C Syntax

```
const char* ix_rm_version_get_string(void);
```


Output/Returns

Return Value Returns the product version description—a string.

3.2.2.6 `ix_rm_sys_config_set()`

Sets the system configuration. Only certain parameters of the system configuration can be set, the others are ignored.

C Syntax

```
ix_error ix_rm_sys_config_set(const ix_sys_config* arg_pSysConfig);
```

Input

`arg_pSysConfig` The location of the system configuration structure specifying system parameter values to set.

Output/Returns

Return Value `IX_SUCCESS` if successful or a valid `ix_error` token for failure.

3.3 Microengine API

Table 3-5 lists the functions and data structures in the microengine API.

Table 3-5. Resource Manager Microengine API

Name	Description
<code>ix_imported_symbol</code>	The structure represents a microcode symbol.
<code>ix_rm_ueng_set_ucode()</code>	Sets the microcode image for microengines from a file.
<code>ix_rm_ueng_map_ucode()</code>	Sets the microcode image for microengines from a buffer.
<code>ix_rm_ueng_reset_all()</code>	Stops and resets all active microengines.
<code>ix_rm_ueng_patch_symbols()</code>	Patches symbols.
<code>ix_rm_ueng_load()</code>	Loads the microcode into the microstore of the microengines.
<code>ix_rm_ueng_start()</code>	Starts the specified microengines.
<code>ix_rm_ueng_stop()</code>	Stops the specified microengines.
<code>ix_rm_ueng_reset()</code>	Resets the specified microengines.
<code>ix_rm_ueng_enable()</code>	Enables the specified microengines.
<code>ix_rm_ueng_disable()</code>	Disables the specified microengines.

3.3.1 Defined Types, Enumerations, and Data Structures

3.3.1.1 `ix_imported_symbol`

This data structure represents a microcode symbol as used in the Resource Manager. This structure defines a name-value pair where the name is the imported symbol's name and the value is the value to patch into microcode.

C Syntax

```
typedef struct ix_imported_symbol {
    ix_uint32 m_Value;
    char* m_Name;
} ix_imported_symbol;
```

Data Members

`m_Value` The name for a name-value pair.

`m_Name` The value for a name-value pair.

3.3.2 API Functions

3.3.2.1 `ix_rm_ueng_set_ucose()`

This function sets the microcode image for microengines from a file. The `arg_pImageName` represents the image name containing the microcode in UOF format. The images are not loaded into the microengines' microstore because symbols still need to be patched using the function `ix_rm_ueng_patch_symbols()`. The images are loaded at the time of the call to `ix_rm_ueng_load()`.

C Syntax

```
ix_error ix_rm_ueng_set_ucose(  
    const char* arg_pImageName);
```

Input

`arg_pImageName` The name the image containing the microcode in UOF format.

Output/Returns

Return Value Returns `IX_SUCCESS` if successful and a valid `ix_error` value otherwise.

3.3.2.2 `ix_rm_ueng_map_ucode()`

Sets the microcode image for microengines from a buffer. The image is taken from a buffer containing the microcode in UOF format. The images are not loaded with this call so that the calling application can first patch symbols using a call to `ix_rm_ueng_patch_symbols()`. The microstore is loaded with an image at the time of the call to `ix_rm_ueng_load()`. The content of the buffer might be modified during this call—`arg_pUOFImage` should point to a writeable memory location rather than a read-only memory location. The buffer pointed to by `arg_pUOFImage` should be valid as long as this microcode is in use. The buffer can be released after a call to `ix_rm_ueng_reset_all()` or `ix_rm_term()` or after any other buffer or microcode file becomes the active image.

C Syntax

```
ix_error ix_rm_ueng_map_ucode(
    void* arg_pUOFImage,
    ix_int32 arg_Size);
```

Input

<code>arg_pUOFImage</code>	A pointer to the image buffer containing the microcode to map.
<code>arg_Size</code>	The size of the image buffer.

Output/Returns

Return Value	Returns <code>IX_SUCCESS</code> if successful and a valid <code>ix_error</code> value otherwise.
--------------	--

3.3.2.3 `ix_rm_ueng_reset_all()`

This function stops and resets all active microengines. It removes the active UOF image from memory and brings the microengine system into initial state. This call is useful when the calling application must change the microcode image run by the microengines at runtime.

C Syntax

```
ix_error ix_rm_ueng_reset_all();
```

Output/Returns

Return Value	Returns <code>IX_SUCCESS</code> if successful and a valid <code>ix_error</code> value otherwise.
--------------	--

3.3.2.4 `ix_rm_ueng_patch_symbols()`

This function patches imported symbols into the microcode. Applications can use this call to specify which imported variables should be patched and with what value. The call also specifies the microengine number so that values may be different for different microengines.

Note: Patching should only be done between the time the microcode image has been loaded into memory using the `ix_rm_ueng_set_ucode()` or `ix_rm_ueng_map_ucode()` calls, until the time the microcode is loaded into the microstore using the `ix_rm_ueng_load()` call. Symbols to be patched are stored in memory until they are actually bound to microcode through the call to `ix_rm_ueng_load()`.

C Syntax

```
ix_error ix_rm_ueng_patch_symbols (
    ix_uint32          arg_UengNumber,
    ix_uint32          arg_SymbolsNumber,
    ix_imported_symbol arg_aSymbols[]);
```

Input

<code>arg_UengNumber</code>	The microengine number identifying the microengine whose microcode is to be patched. Allowed values for the microengine number are 0x00 through 0x03 and 0x10 through 0x13 for the Intel® IXP2400 Network Processor and 0x00 through 0x07 and 0x10 through 0x17 for the Intel® IXP2800 Network Processor. The validity of the microengine number is checked only in debug mode.
<code>arg_SymbolsNumber</code>	The number of symbols in the <code>arg_Symbols</code> array.
<code>arg_Symbols</code>	The array of symbols to patch into the microcode.

Output/Returns

Return Value	Returns <code>IX_SUCCESS</code> if successful and a valid <code>ix_error</code> value otherwise.
--------------	--

3.3.2.5 `ix_rm_ueng_load()`

Load the microcode into the microengine store. This function loads microcode for all microengines. Call `ix_rm_ueng_set_ucode()` or `ix_rm_ueng_map_ucode()` before calling this function.

C Syntax

```
ix_error ix_rm_ueng_load(void);
```

Output/Returns

Return Value	Returns <code>IX_SUCCESS</code> if successful and a valid <code>ix_error</code> value otherwise.
--------------	--

3.3.2.6 `ix_rm_ueng_start()`

This function starts the specified microengine and enables the selected contexts.

C Syntax

```
ix_error ix_rm_ueng_start(
    ix_uint32 arg_MENumber,
    ix_bit_mask32 arg_EnableContextMask);
```

Input

<code>arg_MENumber</code>	The microengine's number for it to be started. Allowed values for the microengine number are 0x00 through 0x03 and 0x10 through 0x13 for the Intel® IXP2400 Network Processor and 0x00 through 0x07 and 0x10 through 0x17 for the Intel® IXP2800 Network Processor. The validity of the microengine number is checked only in debug mode.
<code>arg_EnableContextMask</code>	Mask specifying the microengine contexts to be enabled at start time.

Output/Returns

Return Value	Returns <code>IX_SUCCESS</code> if successful and a valid <code>ix_error</code> value otherwise.
--------------	--

3.3.2.7 `ix_rm_ueng_stop()`

This function stops the specified microengine with all active contexts.

C Syntax

```
ix_error ix_rm_ueng_stop(
    ix_uint32 arg_MENumber);
```

Input

<code>arg_MENumber</code>	The microengine's number for the microengine to be stopped. Allowed values for the microengine number are 0x00 through 0x03 and 0x10 through 0x13 for the Intel® IXP2400 Network Processor and 0x00 through 0x07 and 0x10 through 0x17 for the Intel® IXP2800 Network Processor. The validity of the microengine number is checked only in debug mode.
---------------------------	---

Output/Returns

Return Value	Returns <code>IX_SUCCESS</code> if successful and a valid <code>ix_error</code> value otherwise.
--------------	--

3.3.2.8 `ix_rm_ueng_reset()`

This function resets the specified microengine. If the `arg_ClearRegisters` is non-zero then all the microengine registers are cleared and the microengine is in the same state as at startup.

C Syntax

```
ix_error ix_rm_ueng_reset(
    ix_uint32 arg_MENumber,
    ix_uint32 arg_ClearRegisters)
```

Input

<code>arg_MENumber</code>	The number specifying the microengine to be reset. Allowed values for the microengine number are 0x00 through 0x03 and 0x10 through 0x13 for the Intel® IXP2400 Network Processor and 0x00 through 0x07 and 0x10 through 0x17 for the Intel® IXP2800 Network Processor. The validity of the microengine number is checked only in debug mode.
<code>arg_ClearRegisters</code>	Flag specifying whether all registers should be cleared on reset.

Output/Returns

Return Value	Returns <code>IX_SUCCESS</code> if successful and a valid <code>ix_error</code> value otherwise.
--------------	--

3.3.2.9 `ix_rm_ueng_enable()`

This function enables the contexts specified by the `arg_EnableContextMask` parameter. A context represents a thread in the microengine. Only the enabled threads run for the specified microengine. The microengine should be started in order for this call to succeed.

C Syntax

```
ix_error ix_rm_ueng_enable(
    ix_uint32 arg_MENumber,
    ix_bit_mask32 arg_EnableContextMask);
```


Input

<code>arg_MENumber</code>	The microengine's number for context to be enabled. Allowed values for the microengine number are 0x00 through 0x03 and 0x10 through 0x13 for the Intel® IXP2400 Network Processor and 0x00 through 0x07 and 0x10 through 0x17 for the Intel® IXP2800 Network Processor. The validity of the microengine number is checked only in debug mode.
<code>arg_EnableContextMask</code>	Mask specifying the microengine context to be enabled.

Output/Returns

Return Value	Returns <code>IX_SUCCESS</code> if successful and a valid <code>ix_error</code> value otherwise.
--------------	--

3.3.2.10 `ix_rm_ueng_disable()`

This function disables one or more contexts of the specified microengine. The `arg_DisableContextMask` parameter specifies which contexts should be disabled. The microengine should be started in order for this call to succeed.

C Syntax

```
ix_error ix_rm_ueng_disable(
    ix_uint32 arg_MENumber,
    ix_bit_mask32 arg_DisableContextMask);
```

Input

<code>arg_MENumber</code>	The microengine's number for context to be disabled. Allowed values for the microengine number are 0x00 through 0x03 and 0x10 through 0x13 for the Intel® IXP2400 Network Processor and 0x00 through 0x07 and 0x10 through 0x17 for the Intel® IXP2800 Network Processor. The validity of the microengine number is checked only in debug mode.
<code>arg_DisableContextMask</code>	Mask specifying the microengine context to be disabled.

Output/Returns

Return Value	Returns <code>IX_SUCCESS</code> if successful and a valid <code>ix_error</code> value otherwise.
--------------	--

3.4 Hardware Resource Management API

This section discusses API calls to manage MEv2 hardware features including SRAM queues, rings, scratch rings, and so on. This API may be extended in the future to support reservation of RBUFs, TBUFs, and other hardware features.

Table 3-6 lists the functions and data structures in the Hardware API.

Table 3-6. Resource Manager Hardware API

Name	Description
<code>ix_hw_queue_handle</code>	A generic queue handle for hardware queues.
<code>ix_hw_ring_handle</code>	A generic handle for hardware rings.
<code>ix_sram_ring_size</code>	Enumerated type specifying the supported hardware SRAM ring sizes.
<code>ix_scratch_ring_size</code>	Enumerated type specifying the supported hardware scratch ring sizes.
<code>ix_rm_hw_queue_create()</code>	Creates a hardware queue.
<code>ix_rm_hw_queue_delete()</code>	Deletes a hardware queue.
<code>ix_rm_hw_queue_array_get_base_address()</code>	Returns the virtual base address for an SRAM Q-array allocated for a specific channel.
<code>ix_rm_hw_enqueue()</code>	Enqueues an element to a hardware queue.
<code>ix_rm_hw_dequeue()</code>	Dequeues an element from a hardware queue.
<code>ix_rm_hw_sram_ring_create()</code>	Creates a hardware ring in SRAM memory.
<code>ix_rm_hw_scratch_ring_create()</code>	Creates a hardware ring in scratch memory.
<code>ix_rm_hw_ring_delete()</code>	Deletes a hardware ring.
<code>ix_rm_hw_ring_put()</code>	Puts an element into a hardware ring.
<code>ix_rm_hw_ring_get()</code>	Returns an element from a hardware ring.

3.4.1 SRAM Queues

The SRAM controllers for the IXP2400 and IXP2800 processor series support a data structure called Q-array, which provides hardware-supported basic queue management. These hardware-supported queues enable faster turn-around time for packets in the fast path. See *Intel® IXP2400 Network Processor Hardware Reference Manual* or *Intel® IXP2800 Network Processor Hardware Reference Manual*.

Each element in the Q-array is a queue descriptor used to point to a queue—a singly linked list, ring, or a journal. The Q-array supports up to 64 on-chip queue descriptors on each SRAM controller.

There are two ways of using entries in the queue array. For designs requiring a large number of packet queues, 16 entries of the Q-array are used as a cache. In this case, the entire queue structure resides in SRAM—including the queue descriptor and queue elements—and the hardware Q-array is used as a cache for the queue descriptors. Looking up a particular queue requires CAM support, which can handle up to 16 entries. This implies that the maximum number of queue descriptors which can be cached is 16. The number of queues in SRAM is only limited by the size of the SRAM available.

The other way of using entries in the Q-array—more appropriate for buffer free lists, and so on—is to allocate an entry to be solely owned by a single queue or ring. In this case, the total number of queues or rings supported cannot exceed 64.

The Resource Manager reserves entries in the Q-array for queues and rings. For rings, the Resource Manager allocates memory for the entries in the ring. Apart from applications, the Queue Manager building block described in *Intel® Internet Exchange Architecture Software Building Blocks Developer's Manual* uses the Resource Manager API to reserve up to 16 entries in the Q-array. The Resource Manager Buffer API uses this API to allocate buffer free lists and reserve up to 48 entries in the Q-array.

Note: The allocation of queue descriptors for the packet queues in SRAM is done by the Queue Manager.

When a queue or ring is created, the Resource Manager returns a handle. Subsequently, this handle should be used to access the entity from the Intel XScale® core. Encoded in the handle is an index into the Q-array. This index may be passed onto the microblock—either through an imported variable or through the control block. If the application requests more than one queue the returned handle indicates the base of a newly created array. For example, if the base handle returned is 0x5 for a ten queue array, then the queues are accessed with handles 0x5 for the first queue in the array, 0x6 for the second queue, 0x7 for the third queue, and so on.

The handle is an alias of the generic `ix_handle` type, and is encoded as described in [Section 3.4.1.1.1, “ix_hw_queue_handle.”](#)

3.4.1.1 Defined Types, Enumerations, and Data Structures

3.4.1.1.1 `ix_hw_queue_handle`

A generic type providing a handle to a hardware supported queue. Two parameters, *Channel number* and *Queue index*, are encoded in the handle.

Handle Parameters

`Channel number` The SRAM channel number.

- For IXP2400—0 or 1
- For IXP2800—0 to 3

Represented by bits 30 and 31.

`Queue index` Points to an entry in the Q-array and can have values between 0 and 63.

Represented by bits 0..5.

Bits 6 to 29 are reserved and are set to 0.

The handle encoding is shown in [Figure 3-1](#).

Figure 3-1. Hardware Queue Handle Encoding

Chnl ID		Reserved																										Queue Index					
3	3																											5	4	3	2	1	0
1	0																																

C Syntax

```
typedef ix_handle ix_hw_queue_handle;
```


3.4.1.1.2 `ix_hw_ring_handle`

A generic ring handle.

Data Members

Channel number	<p>The SRAM channel number.</p> <ul style="list-style-type: none"> For the IXP2400—0 or 1 For the IXP2800—0 to 3 For scratch memory on either processor—0x0
Size	<p>Size is the size of the ring expressed as an enumerated value of <code>ix_sram_ring_size</code> and <code>ix_scratch_ring_size</code> types based on the value of Memory Type, either SRAM or SCRATCH.</p>
Memory Type	<p>This type is either <code>IX_MEMORY_TYPE_SRAM</code> or <code>IX_MEMORY_TYPE_SCRATCH</code>.</p>
Ring index	<p>Points to a ring element. The range of values is:</p> <ul style="list-style-type: none"> For SRAM—0 to 63 For scratch memory—0 to 15

The handle is encoded is shown in [Figure 3-2](#).

Figure 3-2. Ring Handle Encoding

Channel								Size								Memory Type								Ring Index							
3							2	2							1	1							8	7							0
1							4	3							6	5															

C Syntax

```
typedef ix_handle ix_ring_handle;
```


3.4.1.2 API Functions

3.4.1.2.1 `ix_rm_hw_queue_create()`

This function reserves `arg_vQueuesNumber` entries in the Q-array of the SRAM controller on the specified channel. `arv_pQueueHandleBase` has the handle of the first queue upon return. A macro is provided to get the n^{th} handle, when required.

This function reserves entries in the Q-array of the SRAM controller on the specified channel.

Note: No queue descriptors are actually allocated by this call. The real allocation is done by the Queue Manager microcode at startup. All queue descriptors must be reserved before the microcode is loaded into the microstore.

A handle identifies each queue created. The handle is used on the Intel XScale® core for enqueue and dequeue operations. If more than one queue is to be created a contiguous area of memory is allocated and the first base handle is returned. If queues are not available contiguously, an error is returned.

Macros are provided (see below) to get the Q-array index from a handle and to get a specific handle from a base.

C Syntax

```
ix_error ix_rm_queue_create (
    ix_uint32 arg_Channel,
    ix_uint32 arg_QueuesNumber,
    ix_uint32 arg_Flags,
    ix_queue_handle* arv_pQueueHandleBase);
```

Input

<code>arg_Channel</code>	The channel of the SRAM controller in question.
<code>arg_QueuesNumber</code>	The number of queues to be allocated.
<code>arg_Flags</code>	The flags modifying the request.

Output/Returns

Return Value	Returns <code>IX_SUCCESS</code> if successful and a valid <code>ix_error</code> value otherwise.
<code>arg_pQueueHandleBase</code>	The address where the first queue handle is stored.

Macros

The following macros are provided for hardware queue support and provide increased portability.

`IX_RM_HW_QUEUE_GET_NEXT_N_HANDLE()`

Returns a specific handle from a base.

```
#define IX_RM_HW_QUEUE_GET_NEXT_N_HANDLE(arv_hHwQueueBase, arg_Index)
```

`IX_RM_HW_QUEUE_GET_INDEX()`

```
#define IX_RM_HW_QUEUE_GET_INDEX(arg_hHwQueue)
```

`IX_RM_HW_QUEUE_SET_INDEX()`

```
#define IX_RM_HW_QUEUE_SET_INDEX(arg_hHwQueue, arg_QueueIndex)
```

`IX_RM_HW_QUEUE_GET_CHANNEL()`

```
#define IX_RM_HW_QUEUE_GET_CHANNEL(arg_hHwQueue)
```

`IX_RM_HW_QUEUE_SET_CHANNEL()`

```
#define IX_RM_HW_QUEUE_SET_CHANNEL(arg_hHwQueue, arg_QueueChannel)
```

`IX_RM_HW_QUEUE_CREATE_HANDLE()`

```
#define IX_RM_HW_QUEUE_CREATE_HANDLE(arv_QueueChannel, arv_QueueIndex)
```


3.4.1.2.2 `ix_rm_hw_queue_delete()`

This function deletes the specified queue. The Q-array entry is freed.

C Syntax

```
ix_error ix_rm_hw_queue_delete(
    ix_hw_queue_handle arg_hHwQueue);
```

Input

`arg_hHwQueue` The queue handle pointing to the queue to be deleted.

Output/Returns

Return Value Returns `IX_SUCCESS` if successful and a valid `ix_error` value otherwise.

3.4.1.2.3 `ix_rm_hw_queue_array_get_base_address()`

This function returns the virtual base address of the core-allocated Q-array descriptor array for each SRAM memory channel. Using this address the user can get the physical offset for the queue descriptor array by a call to `ix_rm_get_phys_offset()` call. This value has to be patched into the microcode that performs the other half of the Q-array initialization on each SRAM channel. The physical offset corresponding to this address is used by the Queue Manager microcode to finish the initialization of the SRAM Q-arrays.

C Syntax

```
ix_error ix_rm_hw_queue_array_get_base_address(
    ix_uint32 arg_Channel,
    ix_uint32** arg_pQArrayBaseAddress);
```

Input

`arg_Channel` Represents the SRAM channel of interest.

`arg_pQArrayBaseAddress` Represents the address where the queue descriptor array base address is stored upon return.

Output/Returns

Return Value `IX_SUCCESS` if successful or a valid `ix_error` token for failure.

3.4.1.2.4 `ix_rm_hw_enqueue()`

This function enqueues an SRAM entry into the specified queue. The first 4 bytes of each entry is reserved for hardware use. The calling application needs to set these entries in the format specified in *Intel® IXP2400 Network Processor Hardware Reference Manual* or *Intel® IXP2800 Network Processor Hardware Reference Manual*, as appropriate. The least significant 24 bits are used as a next address—that is, as a channel longword offset—and the eight most significant bits are used as a cell count, EOP, and SOP flags. By default these are set to 0xff. For details see the Buffer Management API.

Note: It is important to note that the Intel XScale® core can only enqueue to a queue if the queue has a static entry in the Q-array. If the queue is a cached queue, managed by the Queue Manager, then this call should not be used.

Note: The hardware uses the first 4 bytes of buffer as explained in the Buffer Management API chapter.

C Syntax

```
ix_error ix_rm_hw_enqueue(
    ix_hw_queue_handle arg_hHwQueue,
    ix_buffer_handle arg_hHwBuffer)
```

Input

<code>arg_hHwQueue</code>	The queue handle.
<code>arg_hHwBuffer</code>	The entry to be enqueued of type <code>ix_buffer_handle</code> . Only hardware buffer handles can be interpreted by the hardware to support enqueue and dequeue operations.

Output/Returns

Return Value	Returns <code>IX_SUCCESS</code> if successful and a valid <code>ix_error</code> value otherwise.
--------------	--

3.4.1.2.5 `ix_rm_hw_dequeue()`

This function dequeues an SRAM entry from the specified queue. The `arg_pHwBufferHandle` points to the dequeued entry—though only frame mode dequeues are supported in this release of the API. If there is nothing to dequeue, NULL is returned in the buffer.

C Syntax

```
ix_error ix_rm_hw_dequeue(  
    ix_hw_queue_handle arg_hHwQueue,  
    ix_buffer_handle* arg_pHwBufferHandle);
```

Input

<code>arg_hHwQueue</code>	The queue handle specifying the queue from which to dequeue the SRAM entry.
---------------------------	---

Output/Returns

Return Value	Returns <code>IX_SUCCESS</code> if successful and a valid <code>ix_error</code> value otherwise.
<code>arg_pHwBufferHandle</code>	The location to which the SRAM entry should be returned. For queues the dequeued value is a 32-bit integer with a predefined packing which is modeled by the hardware <code>ix_buffer_handle</code> type.

3.4.2 SRAM and Scratch Rings

The scratch pad memory in the Intel® IXP2400 and Intel® IXP2800 Network Processors support rings of various sizes. During scratch ring creation, the Resource Manager initializes the ring registers with appropriate base address and size fields. The scratch memory needed for this ring is also allocated. Once the ring is created, the applications can call Resource Manager functions to put and get data stored in these rings. These rings are accessible from the microengines also. A total of 16 rings are supported by the hardware with ring numbers of zero through fifteen.

The scratchpad memory is 16KB and the scratchpad can be accessed in longwords only. This implies that many combinations of rings are not possible. For example, the total scratchpad memory allocated to support the required rings cannot exceed 4K longwords. If applications need to access scratchpad memory by means other than rings, the space available for rings is further reduced. Access to the ring data are purely under the control of software, and the hardware doesn't prevent accesses to other regions of scratchpad memory. Hence the applications on the Intel XScale® core are required to use Resource Manager functions to at least reserve their requirements for scratch memory.

SRAM rings are supported by the Q-array as described in [Section 3.4.1, “SRAM Queues.”](#) The number of rings supported is restricted only by the entries free in the Q-array.

When a ring is created, the Resource Manager returns a handle. Subsequently, this handle should be used to access the ring from the Intel XScale® core. Encoded in the ring handle is an index into the SRAM Q-array which can be passed to microblocks.

3.4.2.1 Defined Types, Enumerations, and Data Structures

3.4.2.1.1 Handles

The handle is a 32-bit longword and is encoded as described for `ix_hw_ring_handle`. Both SRAM and scratch rings are represented by the same handle type, `ix_hw_ring_handle`.

3.4.2.1.2 `ix_sram_ring_size`

This enumerated type describes the allowed sizes for an SRAM ring.

C Syntax

```
typedef enum ix_e_sram_ring_size {  
    IX_SRAM_RING_SIZE_FIRST = 0,  
    IX_SRAM_RING_SIZE_512 = IX_SRAM_RING_SIZE_FIRST,  
    IX_SRAM_RING_SIZE_1K,  
    IX_SRAM_RING_SIZE_2K,  
    IX_SRAM_RING_SIZE_4K,  
    IX_SRAM_RING_SIZE_8K,  
    IX_SRAM_RING_SIZE_16K,  
    IX_SRAM_RING_SIZE_32K,  
    IX_SRAM_RING_SIZE_64K,  
    IX_SRAM_RING_SIZE_LAST  
} ix_sram_ring_size;
```

Defined Values

<code>IX_SRAM_RING_SIZE_512</code>	The system-defined SRAM-ring sizes.
<code>IX_SRAM_RING_SIZE_1K</code>	
<code>IX_SRAM_RING_SIZE_2K</code>	
<code>IX_SRAM_RING_SIZE_4K</code>	
<code>IX_SRAM_RING_SIZE_8K</code>	
<code>IX_SRAM_RING_SIZE_16K</code>	
<code>IX_SRAM_RING_SIZE_32K</code>	
<code>IX_SRAM_RING_SIZE_64K</code>	

3.4.2.1.3 `ix_scratch_ring_size`

This enumerated type describes the allowed sizes for a *scratch* ring

C Syntax

```
typedef enum ix_e_scratch_ring_size {  
    IX_SCRATCH_RING_SIZE_FIRST = 0,  
    IX_SCRATCH_RING_SIZE_128 = IX_SCRATCH_RING_SIZE_FIRST,  
    IX_SCRATCH_RING_SIZE_256,  
    IX_SCRATCH_RING_SIZE_512,  
    IX_SCRATCH_RING_SIZE_1K,  
    IX_SCRATCH_RING_SIZE_LAST  
} ix_scratch_ring_size;
```

Defined Values

`IX_SCRATCH_RING_SIZE_128` The system-defined scratch-ring sizes.

`IX_SCRATCH_RING_SIZE_256`

`IX_SCRATCH_RING_SIZE_512`

`IX_SCRATCH_RING_SIZE_1K`

3.4.2.2 API Functions

3.4.2.2.1 Bit-Field Macros

The following macros are used for accessing the corresponding bit fields into the handle:

```
#define IX_RM_HW_RING_GET_CHANNEL(arg_hHwRing)
#define IX_RM_HW_RING_SET_CHANNEL(arg_hHwRing, arg_HwRingChannel)
#define IX_RM_HW_RING_GET_SIZE(arg_hHwRing)
#define IX_RM_HW_RING_SET_SIZE(arg_hHwRing, arg_HwRingSize)
```

The ring size returned is one of the enumerated values for `ix_sram_ring_size` or `ix_scratch_ring_size` types, based on the type of the ring. The same applies to the `arg_HwRingSize` parameter for the `IX_RM_HW_RING_SET_SIZE` macro.

3.4.2.2.2 Memory Type Macros

```
#define IX_RM_HW_RING_GET_MEMORY_TYPE(arg_hHwRing)
#define IX_RM_HW_RING_SET_MEMORY_TYPE(arg_hHwRing, arg_HwRingMemoryType)
```

The memory type for the above macros could be one of the enumerated values `IX_MEMORY_TYPE_SRAM` or `IX_MEMORY_TYPE_SCRATCH`.

3.4.2.2.3 Ring Index Macros

```
#define IX_RM_HW_RING_GET_INDEX(arg_hHwRing)
#define IX_RM_HW_RING_SET_INDEX(arg_hHwRing, arg_HwRingIndex)
```


3.4.2.2.4 `ix_rm_hw_sram_ring_create()`

Creates a ring in SRAM memory on a specified channel. The ring size is limited to the values defined in the enumerated types `ix_sram_ring_size`. The Resource Manager allocates memory to hold the ring data in SRAM. The handle returned can be used on the Intel XScale[®] core to consume and produce data within this ring.

For SRAM rings, the handle returned can be used to obtain an index into the SRAM Q-array. This index can be passed to the microengines to access the same ring.

C Syntax

```
ix_error ix_rm_hw_sram_ring_create(
    ix_uint32 arg_Channel,
    ix_sram_ring_size arg_RingSize,
    ix_hw_ring_handle* arg_pHwRingHandle);
```

Input

<code>arg_Channel</code>	The channel for the memory.
<code>arg_RingSize</code>	The ring element size whose value must be one of those specified by <code>ix_sram_ring_size</code> .
<code>arg_pHwRingHandle</code>	Represents the address where the created ring handle should be stored.

Output/Returns

Return Value	Returns <code>IX_SUCCESS</code> if successful and a valid <code>ix_error</code> value otherwise.
<code>arg_pHwRingHandle</code>	Contains the created ring handle.

Helper Macro

```
#define IX_RM_GET_RING_INDEX(arg_vRingHandle)
```


3.4.2.2.5 `ix_rm_hw_scratch_ring_create()`

Creates a *scratch* ring on the specified channel. The ring size is limited to several values as specified by the `ix_scratch_ring_size` enumerated type. The Resource Manager allocates scratch memory needed to hold the ring data. The returned handle can be used on the core to consume and produce data on this ring. Unlike SRAM rings, the scratch ring ID is passed in as an parameter whose value is zero to fifteen. This ID value can be passed to the microengines to access the same ring. If the ring ID passed in by the calling application is in-use the function fails—this is due to the microengine instruction limitation which requires that the ring ID is a compile time constant.

The function allocates all the required memory to hold the ring data in scratch memory. The memory is allocated on the requested channel but, as of now, there is only one scratch memory channel and its logical channel ID is zero.

C Syntax

```
ix_error ix_rm_hw_scratch_ring_create(
    ix_uint32 arg_Channel,
    ix_scratch_ring_size arg_RingSize,
    ix_uint32 arg_RingInternalId,
    ix_hw_ring_handle* arg_pHwRingHandle);
```

Input

<code>arg_Channel</code>	Specifies the memory channel—this is zero at all times.
<code>arg_RingSize</code>	Represents the ring element size as specified by the <code>ix_scratch_ring_size</code> enumerated type.
<code>arg_RingInternalId</code>	Specifies the internal scratch ring ID to be associated with the ring. If the ID specified is in-use then the call fails—this is due to the microengine instruction limitation which requires that the ring ID is a compile time constant.

Output/Returns

<code>arg_pHwRingHandle</code>	A pointer used to return the address of the newly created ring handle.
Return Value	Returns <code>IX_SUCCESS</code> if successful and a valid <code>ix_error</code> value otherwise.

Macros

Returns the ring index.

```
#define IX_RM_GET_RING_INDEX(arg_hHwRing)
```


3.4.2.2.6 `ix_rm_hw_ring_delete()`

Deletes the specified ring. The ring handle and associated Q-array entry is freed—if the ring is in SRAM and the Resource Manager reclaims allocated memory—either SRAM or SCRATCH.

C Syntax

```
ix_error ix_rm_ring_delete (ix_ring_handle arg_hHwRing);
```

Input

<code>arg_hHwRing</code>	Represents the ring to be deleted.
--------------------------	------------------------------------

Output/Returns

Return Value	Returns <code>IX_SUCCESS</code> if successful and a valid <code>ix_error</code> value otherwise.
--------------	--

3.4.2.2.7 `ix_rm_hw_ring_put()`

Produces `*arg_pDataSize` 32-bit words located at `arg_pData` in the specified ring. On return the value of `*arg_pDataSize` specifies the number of 32-bit words that have been put on the ring. If the ring cannot accommodate the entirety of the data, then the function returns an error.

C Syntax

```
ix_error ix_rm_hw_ring_put(  
    ix_hw_ring_handle arg_hHwRing,  
    const ix_uint32* arg_pData,  
    ix_uint32* arg_pDataSize);
```

Input

<code>arg_hHwRing</code>	The ring handle where the data are to be stored.
<code>arg_pData</code>	The location of the data to be stored on the ring.

Input/Output

<code>arg_pDataSize</code>	The number of 32-bit words to be written on the ring. On return this argument stores the actual number of 32-bit words produced.
----------------------------	--

Output/Returns

Return Value	Returns <code>IX_SUCCESS</code> if successful and a valid <code>ix_error</code> value otherwise. If the ring does not accommodate the entire data, then the function succeeds but the actual produced size is less than the requested data size.
--------------	--

3.4.2.2.8 `ix_rm_hw_ring_get()`

This function consumes from the ring `*arg_pDataSize` 32-bit words and store them at the location specified by `arg_pData`. On return the location referred by `arg_pDataSize` stores the actual number of 32-bit words consumed. If there is not enough data, then `arg_vDataSize` is less than the actual size requested.

C Syntax

```
ix_error ix_rm_hw_ring_get(
    ix_hw_ring_handle arg_hHwRing,
    ix_uint32* arg_pDataSize,
    ix_uint32* arg_pData);
```

Input

`arg_hHwRing` The ring handle specifying from where to consume the data.

Input/Output

`arg_pDataSize` On input this parameter specifies the data size in 32-bit words. On output this parameter returns the number of 32-words consumed.

Output/Returns

Return Value Returns `IX_SUCCESS` if successful and a valid `ix_error` value otherwise. If not enough data are to be consumed then the call is successful but the consumed size is less than the requested one.

`arg_pData` The location where the data should be stored.
NOTE: The memory buffer *must* be large enough to hold all requested data.

3.5 Buffer Management API

The buffer API has two parts:

- An API that is used to create buffer free lists of varying sizes
This API is generic and independent of the packet descriptor layout defined by the microblock infrastructure.
- An API that is used to access fields in the packet descriptor—the packet metadata
This API is specific to the layout and fields of the packet descriptor.

3.5.1 Generic Buffers

The buffer API supports both hardware and software buffers. Both hardware and software buffers have similar structure but they are different in the way they are managed: the hardware buffers are handled with direct hardware support, whereas the software buffers are handled entirely by software. The other major difference is that at this time only hardware buffers can be accessed by both the microengine and Intel XScale® core side. In consequence these are the only ones that should be used in core to microengine communication. The software and hardware buffers are both composed of meta and payload data but they differ in the way the information in the buffer handles is packed and in how the required metadata are laid out. This API is generic and independent of packet descriptor layout defined by the microblock infrastructure or by the software.

The type of a buffer is decided by the free list type that allocates the buffer. The free lists can be hardware or software and different creation functions exist for both types. All other API functions encapsulate the hardware and software details—so only one set of functions is required. However for Intel XScale® core to microengine communication only hardware buffers can be used. The maximum number of hardware freelists that can be created on each SRAM channel is defined by the `IX_BUF_MAX_HW_FL_NUMBER` symbol. In the case that multiple channel hardware free lists are used, then the total number of hardware free lists will be multiplied by the number of channels. The total number of free lists that can be created—hardware and software—is defined by the `IX_BUF_MAX_FL_NUMBER` symbol.

The Resource Manager library can be built in two modes with the respect to hardware free lists. The first mode allows creation of hardware free lists on just one SRAM channel. This mode is less flexible but provides simplicity, as there will be no complication to retrieve the meta data for a buffer, based on the buffer handle. The handle contains an SRAM offset for the corresponding meta data. For the case when hardware free lists are supported on a single channel, then the meta data will be uniquely determined. This mode has been extended to support hardware free lists on any of the existing SRAM channels, rather than on only SRAM channel 0 only.

By default, the SRAM channel 0 is chosen, but using the preprocessor symbol `IX_HW_FL_META_CHANNEL` that configuration can be changed at compile time. The channel must be specified as a compile-time symbol definition as follows: `-D IX_HW_FL_META_CHANNEL=1` (or the desired channel number). In this single channel mode, the corresponding microcode has to be adjusted to use the same SRAM channel for hardware free list as the Resource Manager library.

The second mode allows hardware free lists to be created on all channels. This mode imposes a minimum size of 32 bytes for the meta data, and consequently uses bits 1 and 2 of the buffer handle to specify the channel number of the owning hardware free list. In order to compile the Resource Manager library in this mode, the `IX_RM_MULTIPLE_CHANNEL_HW_FREE_LIST` preprocessor symbol must be defined in the compilation.

This section describes the generic buffer API independent of the microblock packet descriptor layout. Table 3-7 lists the functions and data structures in the Buffer Management API.

Table 3-7. Resource Manager Buffer Management API

Name	Description
<code>ix_buffer_handle</code>	A buffer handle.
<code>ix_buffer_free_list_handle</code>	A buffer free list handle.
<code>ix_buffer_free_list_info</code>	Buffer free list data structure.
<code>ix_buffer_type</code>	Enumerated type specifying the type of the buffer—hardware or software.
<code>ix_rm_hw_buffer_free_list_create()</code>	Creates a hardware buffer free list.
<code>ix_rm_sw_buffer_free_list_create()</code>	Creates a software buffer free list.
<code>ix_rm_buffer_free_list_delete()</code>	Deletes a buffer free list.
<code>ix_rm_buffer_free_list_get_info()</code>	Retrieves information about a free list.
<code>ix_rm_buffer_alloc()</code>	Allocates a buffer.
<code>ix_rm_buffer_free()</code>	Frees a buffer.
<code>ix_rm_buffer_free_chain()</code>	Frees and returns a buffer in a chain to the correct buffer free list.
<code>ix_rm_buffer_get_meta()</code>	Returns the metadata for a buffer.
<code>ix_rm_buffer_get_data()</code>	Returns the data associated with a buffer.
<code>ix_rm_buffer_is_eop()</code>	Determines if the buffer is the last one in a chain.
<code>ix_rm_buffer_is_sop()</code>	Determines if the buffer is the first one in a chain.
<code>ix_rm_buffer_get_type()</code>	Determines the type of a buffer—either hardware or software.
<code>ix_rm_buffer_get_next()</code>	Returns the next buffer in a chain.
<code>ix_rm_buffer_link()</code>	Links two buffers into a chain.
<code>ix_rm_buffer_unlink()</code>	Breaks a linked list chain.

Note: These functions are for use in the Intel XScale® core. In the microengines, the XBUF macros are used to allocate and access buffers. (For details, see the *Intel® Internet Exchange Architecture Optimized Data Plane Libraries Reference Manual* located on the IXA SDK Tools CD.)

3.5.1.1 Defined Types, Enumerations, and Data Structures

3.5.1.1.1 ix_buffer_handle

A generic type for buffer handles.

To support hardware free lists on multiple SRAM channels, the structure of the hardware buffer handle is slightly modified. In this case, the meta data for the buffers in this case is assumed to be at least 32 bytes long. That way, all meta addresses will be 32 bytes aligned and the long word (LW) offset address that exists in the handle will always have the last 3 bits set to 0. While the least significant bit must be 0 at all times, as it is used to distinguish between hardware and software buffers, bits 1 and 2 are used to store the SRAM channel number where the meta data resides for the current buffer. The idea is that any time a buffer is allocated, the SRAM channel information is stored into the handle, and when a buffer is released, the channel information is retrieved and masked from the handle.

C Syntax

```
typedef ix_handle ix_buffer_handle;
```

Predefined Value

`IX_NULL_BUFFER_HANDLE` This symbol defines a null buffer handle.

The following is the mapping of buffer handles for hardware and software buffers. The hardware buffer handle mapping is imposed by the hardware design. Both hardware and software buffer handle designs are subject to change and presented here for reference.

Figure 3-3. Hardware Buffer Bit-Field Mapping

E O P	S O P	Cell Count						Address																													
		3	2	1	0	3	2	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0						
3 1	3 0	2 9					2 4	2 3																										0			

Hardware Buffer Bit Fields

EOP	The EOP flag indicates end-of-packet. This is a 1-bit field.
SOP	The SOP flag indicates start-of-packet. This is a 1-bit field.
Cell Count	The cell count for the buffer. This is a 6-bit field.
Address	The address in SRAM of the buffer descriptor expressed as a longword index offset into SRAM channel 0. This is a 24-bit field.

Figure 3-4. Software Buffer Bit-Field Mapping

E O P	S O P	FL Index								Cell Index																							C T
		3	3	2						2	2																					1	0
1	0	9								2	2																						

Software Buffer Bit Fields

EOP	The EOP flag indicates end-of-packet. This is a 1-bit field.
SOP	The SOP flag indicates start-of-packet. This is a 1-bit field.
FL Index	The free list index that owns this buffer. This is an 8-bit field.
Cell Index	The cell index inside specified freelist. This is an 21-bit field.
C T	The cell type. This bit is one at all times to differentiate this buffer from a hardware buffer whose handle has the last bit set to zero at all times—that is, 8-bit meta alignment. This is a 1-bit field.

3.5.1.1.2 ix_buffer_free_list_handle

A generic type for a buffer free list handle.

C Syntax

```
typedef ix_handle ix_buffer_free_list_handle;
```


3.5.1.1.3 ix_buffer_free_list_info

This structure contains all the information associated with a buffer free list. Information about start addresses of the controlled meta and data memory and buffer element sizes for both meta and data are included. This structure is used to return useful information about a buffer free list.

C Syntax

```
typedef struct ix_s_buffer_free_list_info {
    ix_uint32          m_FreeListType;

    #if defined(_IX_RM_SPLIT_META_DATA_)
        ix_uint32*      m_pInternalMetaBaseAddress;
        ix_memory_type  m_InternalMetaMemoryType;
        ix_uint32       m_InternalMetaMemoryChannel;
        ix_uint32       m_InternalMetaElementSize;
    #endif /* defined(_IX_RM_SPLIT_META_DATA_) */

    ix_uint32*      m_pMetaStart;
    ix_memory_type  m_MetaMemoryType;
    ix_uint32       m_MetaMemoryChannel;
    ix_uint32       m_MetaElementSize;
    ix_uint32*      m_pDataStart;
    ix_memory_type  m_DataMemoryType;
    ix_uint32       m_DataMemoryChannel;
    ix_uint32       m_DataElementSize;
    ix_uint32       m_NumElements;
    ix_uint32       m_FreeListInfo;
    ix_uint32       m_FreeListInfo1;
} ix_buffer_free_list_info;
```

Data Members

m_FreeListType	Specifies if the free list is software or hardware based.
m_pInternalMetaBaseAddress	Specifies the start of the internal controlled meta memory.
m_InternalMetaMemoryType	Specifies internal meta block memory type.
m_InternalMetaMemoryChannel	Specifies internal meta block memory channel.
m_InternalMetaElementSize	Specifies the buffer internal meta element size.
m_pMetaStart	A pointer to the start of the controlled meta memory.
m_MetaMemoryType	The meta-block memory type.
m_MetaMemoryChannel	The meta-block memory channel.
m_MetaElementSize	Specifies the buffer meta-element size.
m_pDataStart	Specifies the start of the controlled data memory.

Data Members (Continued)

<code>m_DataMemoryType</code>	The data-block memory type.
<code>m_DataMemoryChannel</code>	The meta-block memory channel.
<code>m_DataElementSize</code>	Specifies the buffer data-element size.
<code>m_NumElements</code>	Specifies the number of buffer elements in the list.
<code>m_FreeListInfo</code>	Specifies the SRAM controller Q-array index for the corresponding hardware free list and the free list index for the software free list.
<code>m_FreeListInfo1</code>	Specifies the free list ID for hardware free lists.

3.5.1.1.4 `ix_buffer_type`

This enumerated type differentiates between software and hardware buffers. The hardware ones have hardware support for managing them and can be used by both microcode and core API. The software buffers are managed totally in software and must be used just by the core applications.

C Syntax

```
typedef enum ix_e_buffer_type {
    IX_BUFFER_TYPE_FIRST = 0,
    IX_BUFFER_TYPE_HARDWARE = IX_BUFFER_TYPE_FIRST,
    IX_BUFFER_TYPE_SOFTWARE,
    IX_BUFFER_TYPE_LAST
} ix_buffer_type;
```


3.5.1.2 API Functions

3.5.1.2.1 `ix_rm_hw_buffer_free_list_create()`

Creates a hardware buffer free list with `arg_ElementsNumber` entries. Memory is allocated in SRAM and DRAM. A handle to the new created free list is returned. A combination of an `arg_SRAMSize` of zero and an `arg_DRAMSize` of zero is not accepted and the function returns an error. The data and meta element size is always rounded to the next power of two—it is better to pass in these preferred sizes. A maximum of `IX_BUF_MAX_HW_FL_NUMBER` hardware free lists can be created. The minimum accepted size for hardware buffer meta element is the size of the `ix_hw_buffer_meta` structure.

Note: This function should be called before calling `ix_rm_ueng_load()`. In part this is because part of the hardware free list initialization is done by the microcode—Intel XScale® core can not write Q-array descriptors into the SRAM. Allocate hardware buffers after the `ix_rm_ueng_load()` call as well—allowing some time for the microengines to perform the initialization of the Q-array descriptors through the SRAM controller.

In the case of single SRAM channel hardware free list support, the `arg_SRAMChannel` parameter should be the same as the value of `IX_HW_FL_META_CHANNEL` preprocessor symbol.

In the case of split meta data configuration, the first part of the meta data will be created on the `IX_HW_FL_META_CHANNEL` SRAM channel and the second part of the meta data will be created on the SRAM channel specified by the `arg_SRAMChannel` argument.

C Syntax

```
ix_error ix_rm_hw_buffer_free_list_create(
    ix_uint32 arg_ElementsNumber,
    ix_uint32 arg_SRAMSize,
    ix_uint32 arg_DRAMSize,
    ix_uint32 arg_SRAMChannel,
    ix_uint32 arg_DRAMChannel,
    ix_buffer_free_list_handle* arg_pFreeListHandle);
```

Input

<code>arg_ElementsNumber</code>	The number of free list entries to allocate.
<code>arg_SRAMSize</code>	The size of SRAM to allocate for each entry.
<code>arg_DRAMSize</code>	The size of DRAM to allocate for each entry.
<code>arg_SRAMChannel</code>	The SRAM channel where the memory is allocated.
<code>arg_DRAMChannel</code>	The DRAM channel where the memory is allocated.

Output/Returns

<code>arg_pFreeListHandle</code>	The pointer to the address used to return the handle to the newly created free list.
Return Value	Returns <code>IX_SUCCESS</code> if successful and a valid <code>ix_error</code> value otherwise. If not enough data are to be consumed then the call is successful but the consumed size is less than the requested one.

3.5.1.2.2 `ix_rm_sw_buffer_free_list_create()`

Creates a software buffer free list that allocates and manages `arg_NumElements` elements. The caller has to specify the sizes of the meta and data parts of the buffer. The sizes are always rounded to the next power of two—preferred sizes should be passed in.

The meta and data blocks can be allocated in a flexible way in different types of memory, but SRAM and especially scratch memory should be used only if badly needed. Memory is allocated by the Resource Manager as requested.

The sizes of the metadata and data can not both be zero. A maximum of `IX_BUF_MAX_SW_FL_NUMBER` software free lists can be created. A handle to the free list is returned.

C Syntax

```
ix_error ix_rm_sw_buffer_free_list_create(
    ix_uint32 arg_MetaElementSize,
    ix_memory_type arg_MetaMemoryType,
    ix_uint32 arg_DataElementSize,
    ix_memory_type arg_DataMemoryType,
    ix_uint32 arg_NumElements,
    ix_buffer_free_list_handle* arg_pPoolManagerHandle);
```

Input

<code>arg_MetaElementSize</code>	The size of a meta element. The size is rounded to the next power of two.
<code>arg_MetaMemoryType</code>	The type of the meta memory. This value is one of {SRAM, DRAM, or SCRATCH}. NOTE: SRAM and especially SCRATCH should be used with great care.

Input

<code>arg_DataElementSize</code>	The size of a data element. The size is rounded to the next power of two.
<code>arg_DataMemoryType</code>	The type of the data memory. This value is one of {SRAM, DRAM, or SCRATCH}. NOTE: SRAM and especially SCRATCH should be used with great care.
<code>arg_NumElements</code>	The number of memory-buffer elements that this manager is to allocate and manage.

Output/Returns

Return Value	Returns <code>IX_SUCCESS</code> if successful or a valid <code>ix_error</code> token for failure.
<code>arg_pPoolManagerHandle</code>	Upon return, the address where the handle of the new created pool manager is stored.

3.5.1.2.3 `ix_rm_buffer_free_list_delete()`

Deletes the specified buffer free list. All associated memory is freed by and available from the Resource Manager.

C Syntax

```
ix_error
ix_rm_buffer_free_list_delete(
    ix_buffer_free_list_handle arg_hFreeList);
```

Input

`arg_hFreeList` The handle of the free list to be deleted.

Returns

Return Value Returns `IX_SUCCESS` if successful and a valid `ix_error` value otherwise.

3.5.1.2.4 `ix_rm_buffer_free_list_get_info()`

This function retrieves buffer free list information. Once a buffer free list is created the caller has no control over where the required memory is allocated.

This function retrieves information specific to a buffer free list. For example, the calling application might later need the start addresses for a buffer free list to pass these to a microengine.

C Syntax

```
ix_error ix_rm_buffer_free_list_get_info(
    ix_buffer_free_list_handle arg_hFreeList,
    ix_buffer_free_list_info* arg_pFreeListInfo);
```

Input

`arg_hFreeList` Specifies the free list handle about which to retrieve information.

Output/Returns

Return Value Returns `IX_SUCCESS` if successful or a valid `ix_error` for failure.

`arg_pFreeListInfo` A pointer to a structure where the function has stored the returned free list information.

3.5.1.2.5 `ix_rm_buffer_alloc()`

Allocate a buffer from the specified buffer free list. The buffer handle is returned in a `arg_pBufferHandle`. This buffer handle can be used to get a pointer to the meta information and data area for the buffer. Based on the type of free list used a hardware or software buffer is created.

C Syntax

```
ix_error ix_rm_buffer_alloc(
    ix_buffer_free_list_handle arg_hFreeList,
    ix_buffer_handle* arg_pBufferHandle);
```

Input

`arg_hFreeList` The free list handle.

Output/Returns

Return Value Returns `IX_SUCCESS` if successful and a valid `ix_error` value otherwise.

`arg_pBufferHandle` The newly allocated buffer handle.

3.5.1.2.6 `ix_rm_buffer_free()`

This function frees the buffer specified by `arg_hBuffer`. The buffer is returned to the correct buffer free list which is encoded in the handle itself. When the buffer is part of a chain only the buffer passed in through `arg_hBuffer` is freed and the chain becomes inconsistent.

C Syntax

```
ix_error ix_rm_buffer_free(
    ix_buffer_handle arg_hBuffer);
```

Input

`arg_hBuffer` The handle of the buffer to be freed.

Returns

Return Value Returns `IX_SUCCESS` if successful and a valid `ix_error` value otherwise.

3.5.1.2.7 `ix_rm_buffer_free_chain()`

This function frees the buffer chain starting from `arg_hBuffer`. The buffers are returned to appropriate freelists. If the buffer is the head of a buffer chain then the entire chain is freed and the `SOP` flag is set. If this buffer is in the middle of a chain then the end of the chain is freed starting from this buffer. The chain has to be unlinked at this buffer boundary in order to avoid chain inconsistency.

C Syntax

```
ix_error ix_rm_buffer_free_chain(
    ix_buffer_handle arg_hBuffer);
```

Input

<code>arg_hBuffer</code>	The handle of the buffer head of chain to be.
--------------------------	---

Output/Returns

Return Value	Returns <code>IX_SUCCESS</code> if successful and a valid <code>ix_error</code> value otherwise.
--------------	--

3.5.1.2.8 `ix_rm_buffer_get_meta()`

Returns the memory base location of metadata for the buffer specified by `arg_hBuffer`.

In the case of split meta data, the address to the real buffer meta data structure is returned, so the field `m_HwNext` will not be accessible.

C Syntax

```
ix_error ix_rm_buffer_get_meta(
    ix_buffer_handle arg_hBuffer,
    void** arg_pMetaData);
```

Input

<code>arg_hBuffer</code>	The handle of the buffer for which the metadata are requested.
--------------------------	--

Output/Returns

Return Value	Returns <code>IX_SUCCESS</code> if successful and a valid <code>ix_error</code> value otherwise.
--------------	--

<code>arg_pMetaData</code>	The location where the metadata address is written.
----------------------------	---

3.5.1.2.9 `ix_rm_buffer_get_data()`

Returns the base of the packet data portion of the buffer specified by `arg_pData`.

C Syntax

```
ix_error ix_rm_buffer_get_data (
    ix_buffer_handle arg_hBuffer,
    void** arg_pData);
```

Input

`arg_hBuffer` The handle of the buffer.

Output/Returns

`arg_pData` The location where the data address is written.

Return Value Returns `IX_SUCCESS` if successful and a valid `ix_error` value otherwise.

3.5.1.2.10 `ix_rm_buffer_is_eop()`

This function returns, in the argument `*arg_pIsEOP`, non-zero if this buffer is the last in a list of concatenated memory buffers and zero otherwise. This is the equivalent of the end-of-packet property of a buffer. Usually only the first buffer from a packet is passed for processing, so the processing entity has to process all buffers in the chain. This flag signals that we reached the end of the chain. Single buffers have this flag and the SOP—start-of-packet—flag set and the next buffer is set to `IX_NULL_BUFFER_HANDLE`.

C Syntax

```
ix_error ix_rm_buffer_is_eop(
    ix_buffer_handle arg_hBuffer,
    ix_uint32* arg_pIsEOP);
```

Input

`arg_hBuffer` The handle to a buffer whose EOP property is of interest.

Output/Returns

Return Value	Returns <code>IX_SUCCESS</code> if successful or a valid <code>ix_error</code> token for failure.
<code>arg_pIsEOP</code>	The address where the EOP property of the buffer is stored. On return <code>*arg_pIsEOP</code> is zero if the buffer is not the last in a chain or non-zero otherwise.

3.5.1.2.11 `ix_rm_buffer_is_sop()`

This function returns—in the argument `*arg_pIsSOP`—non-zero if this buffer is the first in a list of concatenated memory buffers and zero otherwise. This is the equivalent of the start-of-packet property of a buffer. This function determines if the buffer is the first buffer in a chain.

C Syntax

```
ix_error ix_rm_buffer_is_sop(
    ix_buffer_handle arg_hBuffer,
    ix_uint32* arg_pIsSOP);
```

Input

<code>arg_hBuffer</code>	The handle to a buffer whose SOP property is of interest.
--------------------------	---

Output/Returns

Return Value	Returns <code>IX_SUCCESS</code> if successful or a valid <code>ix_error</code> token for failure.
<code>arg_pIsSOP</code>	The address where the SOP property of the buffer is stored. On return <code>*arg_pIsSOP</code> is zero if the buffer is not the first in a chain or non-zero otherwise.

3.5.1.2.12 `ix_rm_buffer_get_type()`

This function retrieves the type of the buffer for the handle passed as the first parameter. This function determines if a buffer is a software or a hardware buffer.

C Syntax

```
ix_error ix_rm_buffer_get_type(
    ix_buffer_handle arg_hBuffer,
    ix_buffer_type* arg_pBufferType);
```

Input

<code>arg_hBuffer</code>	The handle of the buffer whose type we want to retrieve.
--------------------------	--

Output/Returns

Return Value	Returns <code>IX_SUCCESS</code> if successful or a valid <code>ix_error</code> token for failure. On return the <code>arg_pBufferType</code> variable contains the type of the buffer.
--------------	--

<code>arg_pBufferType</code>	The address where the buffer type is stored on return.
------------------------------	--

3.5.1.2.13 `ix_rm_buffer_get_next()`

This function returns the next link for this buffer. If there are no buffers linked to this one the `arg_pNextBufferHandle` location is set to `IX_NULL_BUFFER_HANDLE`. This function retrieves the next buffer handle relative to our base buffer. If this buffer is the last in the chain then `IX_NULL_BUFFER_HANDLE` is returned.

C Syntax

```
ix_error ix_rm_buffer_get_next(
    ix_buffer_handle arg_hBuffer,
    ix_buffer_handle* arg_pNextBufferHandle);
```

Input

<code>arg_hBuffer</code>	The handle of the buffer for which the next link is needed.
--------------------------	---

Output/Returns

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

<code>arg_pNextBufferHandle</code>	The location where the next link for this buffer is stored.
------------------------------------	---

3.5.1.2.14 `ix_rm_buffer_link()`

This function links the buffer specified by `arg_pNextBufferHandle` to the buffer specified by `arg_pBufferHandle`. The buffers form two links in a linked list data structure. If the current buffer, specified by `arg_pBufferHandle`, already has a link—that is, if it already is part of a linked list—then this function returns an error. On return the handles of the two buffers are modified to specify the new EOP and SOP state.

Note: If the buffer specified by the argument `arg_pBufferHandle` is not an EOP buffer, then the call returns an error.

Note: On return, EOP and SOP flags are appropriately set for both buffer handles.

C Syntax

```
ix_error ix_rm_buffer_link(
    ix_buffer_handle* arg_pBufferHandle,
    ix_buffer_handle* arg_pNextBufferHandle);
```

Input

<code>arg_pBufferHandle</code>	The reference to the handle of the buffer to which to link.
<code>arg_pNextBufferHandle</code>	The reference to the handle of the buffer to be added to the linked list.

Output/Returns

Return Value	Returns <code>IX_SUCCESS</code> if successful or a valid <code>ix_error</code> token for failure. If the current buffer specified by <code>arg_pBufferHandle</code> already has a link—that is, if it is not EOP—then the function returns an error.
--------------	--

3.5.1.2.15 `ix_rm_buffer_unlink()`

This function unlinks the buffer specified by `arg_pBufferHandle`. If the buffer has a link that link is set to `IX_NULL_BUFFER_HANDLE` and the unlinked buffer handle is returned at the location specified by `arg_pUnlinkedBufferHandle`. For this function to work `arg_pBufferHandle` should have the SOP flag set, meaning that only the beginning of a chain can be unlinked.

Note: If the buffer specified by the first argument is not an SOP buffer this call fails.

Note: On return the handles of the two buffers are modified to specify the new EOP and SOP states.

C Syntax

```
ix_error ix_rm_buffer_unlink(
    ix_buffer_handle* arg_pBufferHandle,
    ix_buffer_handle* arg_pUnlinkedBufferHandle);
```

Input

<code>arg_pBufferHandle</code>	The reference to the handle of the buffer that is to change from a linked to an unlinked state.
--------------------------------	---

Output/Returns

Return Value	Returns <code>IX_SUCCESS</code> if successful or a valid <code>ix_error</code> token for failure.
<code>arg_pUnlinkedBufferHandle</code>	The location where the handle of the newly unlinked buffer is stored.

3.5.2 Framework Buffer Structure

For hardware free lists, a typical DRAM buffer size is 2048 bytes and the default SRAM buffer descriptor (meta data) size is 32 bytes, which is the size of `ix_hw_buffer_meta` structure. For software buffers, there is a minimum size of the buffer descriptor of 16 bytes imposed by the size of `ix_sw_buffer_meta` structure. The developer may change these values at compile time or run time. The SRAM and DRAM base addresses and size values for the hardware free list are patched into the microcode for use in the dispatch loop macros. The dispatch loop macros use these imported variables in calling the IXP buffer macros. For POS and ATM—where packets may be greater than 2048 bytes—packets are stored in multiple buffers chained together.

In order to improve the performance of the system, the way the hardware buffers are created have been extended to allow split meta data. The split meta data configuration for hardware buffers can be selected by compiling the Resource Manager library with the `_IX_RM_SPLIT_META_DATA_` preprocessor symbol defined.

The 32 byte SRAM buffer descriptor pre-defined by the Resource Manager is described in [Table 3-8, “Resource Manager Packet Metadata Definitions.”](#) Some of these fields are common to all applications while others of these fields are specific to particular categories of applications.

Calling applications may add—or trim—fields to customize the metadata to their application category with the exception of the first member, `m_HwNext`, that is required by the hardware and `m_BufferInfo` that is used by software to determine the Q-array descriptor index corresponding to the free list that owns the buffer. The position in the structure of `m_HwNext` should not change at any time. The position of `m_BufferInfo` can change, but it needs to contain the `free_list_id` bit field in the same position and there should be agreement between microcode and Intel XScale® core code.

The IXA Software Framework is dependent on this pre-defined 32 byte SRAM buffer descriptor. [Section 3.5.2.1, “Packet Metadata Description,”](#) discusses application use of this metadata. [Table 3-8](#) lists these packet metadata fields. If a calling application changes this layout, then the associated dispatch loop macros and the Resource Manager library must also be updated by the application developer. [Section 3.5.2.1.2, “Extending Packet Metadata,”](#) describes how applications can extend this core data structure so that the dispatch loop macros and the Resource Manager library do not need to be updated by the application developer.

3.5.2.1 Packet Metadata Description

The structure, `ix_hw_buffer_meta`, can be extended, but the Resource Manager code expects the first 32 bytes of SRAM buffer descriptor to conform to the field layout described in [Section 3.5.2.1.3, “IX_DECLARE_HW_BUFFER_META_DATA for Common Meta Data.”](#)

[Section 3.5.2.1.1, “ix_hw_buffer_meta”](#) describes the use of the base 32-byte SRAM buffer descriptor as-is. [Section 3.5.2.1.2, “Extending Packet Metadata,”](#) describes application-specific extensions of this common core.

3.5.2.1.1 ix_hw_buffer_meta

This structure defines the information and the layout of the common core of buffer metadata using a macro—see [Section 3.5.2.1.3, “IX_DECLARE_HW_BUFFER_META_DATA for Common Meta Data.”](#) Use the data structure declaration shown in this section if application code can use the pre-defined 32-byte SRAM buffer descriptor as-is. To add—or trim—application-specific fields see [Section 3.5.2.1.2, “Extending Packet Metadata.”](#)

C Syntax

```
typedef struct ix_s_hw_buffer_meta {
    IX_DECLARE_HW_BUFFER_META_DATA
} ix_hw_buffer_meta;
```

3.5.2.1.2 Extending Packet Metadata

Calling applications requiring extra fields can do so in the following manner without the expense of an extra indirection.

C Syntax

```
typedef struct ix_s_atm_buffer_meta {
    IX_DECLARE_HW_BUFFER_META_DATA
    ix_uint32      m_CellHeader;
} ix_atm_buffer_meta;
```


3.5.2.1.3 IX_DECLARE_HW_BUFFER_META_DATA for Common Meta Data

The symbol `IX_DECLARE_HW_BUFFER_META_DATA` declares all the fields and the layout of the common part of the buffer meta data. These fields are described in [Table 3-8](#).

Table 3-8. Resource Manager Packet Metadata Definitions

LW	Bits	Size	Data Member	Field	Description
0	31:00	32	m_HwNext		The buffer handle of the next buffer in the chain.
1	31:16	16	m_BufferSize		The buffer size—in bytes.
	15:00	16	m_Offset		The offset of the start of data in the buffer—in bytes.
2	31:28	16	m_PacketSize		The size of the entire packet across buffers—in bytes.
	15:12	4	m_BufferInfo ¹	free_list_id ¹	The free list ID for the buffer.
	11:08	4		rx_stat ¹	The receive status flag.
	07:00	8		header_type ¹	The type of header at m_Offset bytes into the packet.
3	31:16	16	m_InputPort		The input port on the ingress processor.
	15:00	16	m_OutputPort		The output port on the egress processor.
4	31:16	16	m_NextHopID		The next hop ID.
	15:08	8	m_FabricPort		The output port for the switch fabric indicating the destination blade.
	07:00	8	m_Reserved1		Reserved.
5	31:00	32	m_FlowID		The flow ID—a QoS flow ID or an MPLS label or flow ID.
6	31:16	16	m_ClassID		The class ID.
	15:00	16	m_Reserved2		Reserved.
7	31:00	32	m_PacketNext		A pointer to next the packet—unused in cell mode.

1. The data member, `m_BufferInfo`, is a packed buffer containing the bit fields `free_list_id`, `rx_stat`, and `header_type`. The macros in [Section 3.5.2.3](#) should be used to extract these bit fields from the packed buffer.

C Syntax

```
#define IX_DECLARE_HW_BUFFER_META_DATA \
    /* longword 0 */ \
    ix_uint32    m_HwNext; \
    /* longword 1 */ \
    ix_uint16    m_BufferSize; \
    ix_uint16    m_Offset; \
    /* longword 2 */ \
    ix_uint16    m_PacketSize; \
    \
    /** \
     * The following represents packed buffer information. \
     * 15:12 4 bits    free_list_id  Free list ID for the buffer \
     * 11:8   4 bits    rx_stat      Receive status flag
```



```

        *   7:0   8 bits   header_type   Type of header at "offset" bytes
        *                                           into the packet
        */ \
ix_uint16      m_BufferInfo; \
/* longword 3 */ \
ix_uint16      m_InputPort; \
ix_uint16      m_OutputPort; \
/* longword 4 */ \
ix_uint16      m_NextHopID; \
ix_uint8       m_FabricPort; \
ix_uint8       m_Reserved1; \
/* longword 5 */ \
ix_uint32      m_FlowID; \
/* longword 6 */ \
ix_uint16      m_ClassID; \
ix_uint16      m_Reserved2; \
/* longword 7 */ \
ix_uint32      m_PacketNext; \

```

3.5.2.2 Split Meta Data Configuration Details

When a split meta data configuration is used, then the meta data is allocated in two parts: one part contains the `m_HwNext` field plus the Intel XScale® core address to the second part of the meta data, and the second part which is similar to the original meta data with the exception that the field in the position of `m_HwNext` field is reserved.

3.5.2.2.1 ix_hw_internal_buffer_meta

This is the internal HW meta data structure when split meta data is used. This portion should be accessible only by the resource manager, because the other applications should not care about the `m_HwNext` field.

C Syntax

```

typedef struct ix_s_hw_internal_buffer_meta
{
    /* long word 0 */
    ix_uint32      m_HwNext; /* reserved hardware link to the next data */
    /* long word 1 */
    ix_hw_buffer_meta* m_pMeta; /* address of the proper meta data */
} ix_hw_internal_buffer_meta;

```

3.5.2.2.2 IX_DECLARE_HW_BUFFER_META_DATA for Split Meta Data

The symbol `IX_DECLARE_HW_BUFFER_META_DATA` declares the fields and the layout of the buffer meta data when the split meta data configuration is used, as described in [Table 3-9](#).

In this case, the Intel XScale® core will have to patch into the microcode more information for the microcode to be able to access both types of meta data. When the common meta data is used, the microcode accesses the meta data based on the address offset contained in the buffer handle. In the split meta data case, that offset points to the internal meta data that contains the `m_HwNext` field and the Intel XScale® core address to the real buffer meta data. The same strategy used to find the data portion corresponding to the buffer can be used to retrieve the meta data portion as well.

In the case that the support for hardware free lists on multiple channels is enabled, then the first part of the meta data will have an allocated size of 32 bytes to accommodate the alignment requirements.

Table 3-9. Resource Manager Packet Metadata Definitions for Split Meta Data

LW	Bits	Size	Data Member	Field	Description
0	31:00	32	m_Reserved0		Reserved hardware link to the next data.
1	31:16	16	m_BufferSize		The buffer size—in bytes.
	15:00	16	m_Offset		The offset of the start of data in the buffer—in bytes.
2	31:28	16	m_PacketSize		The size of the entire packet across buffers—in bytes.
	15:12	4	m_BufferInfo ¹	free_list_id ¹	The free list ID for the buffer.
	11:08	4		rx_stat ¹	The receive status flag.
	07:00	8		header_type ¹	The type of header at m_Offset bytes into the packet.
3	31:16	16	m_InputPort		The input port on the ingress processor.
	15:00	16	m_OutputPort		The output port on the egress processor.
4	31:16	16	m_NextHopID		The next hop ID.
	15:08	8	m_FabricPort		The output port for the switch fabric indicating the destination blade.
	07:00	8	m_Reserved1		Reserved.
5	31:00	32	m_FlowID		The flow ID—a QoS flow ID or an MPLS label or flow ID.
6	31:16	16	m_ClassID		The class ID.
	15:00	16	m_Reserved2		Reserved.
7	31:00	32	m_PacketNext		A pointer to next the packet—unused in cell mode.

1. The data member, m_BufferInfo, is a packed buffer containing the bit fields free_list_id, rx_stat, and header_type. The macros in [Section 3.5.2.3](#) should be used to extract these bit fields from the packed buffer.

C Syntax

```
#define IX_DECLARE_HW_BUFFER_META_DATA \
/* long word 0 */ \
ix_uint32      m_Reserved0; \
/* long word 1 */ \
ix_uint16      m_BufferSize; \
ix_uint16      m_Offset; \
/* long word 2 */ \
ix_uint16      m_PacketSize; \
\
\
/** \
 * The following represents packed buffer information. \
 * +-----+-----+-----+ \
 * | Free List ID | RxStat | Header type | \
 * | 15:12 4 bit | 11:8 4 bit | 7:0 8 bit | \
```



```

* +-----+-----+-----+ \
* 15:12 4 bits   free_list_id  Free list ID for the buffer
* 11:8  4 bits   rx_stat       Receive status flag
* 7:0   8 bits   header_type   Type of header at "offset" bytes
*                               into the packet
*/ \
ix_uint16      m_BufferInfo; \
/* long word 3 */ \
ix_uint16      m_InputPort; \
ix_uint16      m_OutputPort; \
/* long word 4 */ \
ix_uint16      m_NextHopID; \
ix_uint8       m_FabricPort; \
ix_uint8       m_Reserved1; \
/* long word 5 */ \
ix_uint32      m_FlowID; \
/* long word 6 */ \
ix_uint16      m_ClassID; \
ix_uint16      m_Reserved2; \
/* long word 7 */ \
ix_uint32      m_PacketNext; \

```

3.5.2.3 Packed Field Macros

The following macros should be used for accessing and modifying the packet metadata packed fields—`free_list_id`, `header_type`, and `rx_stat`.

`IX_RM_HW_META_GET_FREE_LIST_ID`

Returns the `free_list_id` bit field value. `arg_BufferInfo` should be a 16-bit value.

```
#define IX_RM_HW_META_GET_FREE_LIST_ID(arg_BufferInfo)
```

`IX_RM_HW_META_SET_FREE_LIST_ID`

Sets the `free_list_id` bit field value. `arg_BufferInfo` should be a 16-bit value.

```
#define IX_RM_HW_META_SET_FREE_LIST_ID(arg_BufferInfo, arg_FreeListID)
```

`IX_RM_HW_META_GET_HEADER_TYPE`

Returns the `header_type` bit field value. `arg_BufferInfo` should be a 16-bit value.

```
#define IX_RM_HW_META_GET_HEADER_TYPE(arg_BufferInfo)
```

`IX_RM_HW_META_SET_HEADER_TYPE`

Sets the `header_type` bit field value. `arg_BufferInfo` should be a 16-bit value.

```
#define IX_RM_HW_META_SET_HEADER_TYPE(arg_BufferInfo, arg_HeaderType)
```

`IX_RM_HW_META_GET_RX_STAT`

Returns the `rx_stat` bit field value. `arg_BufferInfo` should be a 16-bit value.

```
#define IX_RM_HW_META_GET_RX_STAT(arg_BufferInfo)
```


`IX_RM_HW_META_SET_RX_STAT`

Sets the `rx_stat` bit field value. `arg_BufferInfo` should be a 16-bit value.

```
#define IX_RM_HW_META_SET_RX_STAT(arg_BufferInfo, arg_RxStat)
```

3.6 Communication API

The Resource Manager Communication API implements the mechanism to transport packets and control messages between the core components or core components and microblocks through an abstraction called a communication ID. A communication ID represents a destination where messages and packets can be sent. On a local system there is a limited—this limit is compile-time configurable—number of communication IDs expressed by the `IX_COMM_LOCAL_ID_NUMBER` symbol. Out of this number of communication IDs `IX_COMM_UBLOCK_ID_NUMBER` IDs are reserved for communication with the microblocks, and the remaining ones are dedicated to inter core component communication. For the core communication ID core components can choose to listen for incoming messages and packets through several mechanisms.

- The calling application polls a communication ID for messages or packets
- The calling application retrieves messages and packets from a communication ID using a synchronous function that blocks until a message or a packet arrives or the call times out
- The calling application specifies interest in several communication IDs and the call returns only when one of the specified IDs receives data
- The calling application registers a callback function that is called whenever new data arrives on a communication ID

All of these options apply only to the core communication IDs. These communication IDs can be regarded as gateways. For the corresponding microblock IDs, one side of this gateway is in the microcode but this microcode has no visibility into the core application.

These communication IDs are represented by a generic type `ix_communication_id` that is an unsigned 32-bit handle defining a destination. The communication ID mechanism allows for local communication as well as remote communication with other systems. If the destination is not intended for the local system then the messages and packets are forwarded to a proxy that routes the data through PCI or another communication path to the remote system. A portion of the communication ID specifies if the destination is on the current subsystem or—in the case of a dual ingress/egress network processor system—on the peer subsystem.

A zero value for this bit field at the time of creation of the communication ID specifies a destination on the local subsystem. A non-zero value at the time of creation specifies a destination on the peer subsystem. Each system that works in a group should have a unique identifier.

The communication between microblocks and core is handled in the following way. The microcode queues data onto a common hardware ring and signals the Intel XScale® core that data has been sent. For packet communication, the scratch ring with the ID of zero is used to queue data, and the core is signaled through `Thread_Interrupt_A_#`. (For details, see the *Intel XScale® core Gasket* section in the *Intel® IXP2400 Network Processor Hardware Reference Manual* or the *Intel® IXP2800 Network Processor Hardware Reference Manual*.)

For message communication, the scratch ring with ID of one is used to queue data, and the Intel XScale® core is signaled through `Thread_Interrupt_B_#`. On the core side, the registered ISRs awaken the corresponding dispatch threads that dequeue the data and send it to the requested destination. The message dispatch thread has higher priority than the packet dispatch thread.

Currently, the microcode queues two 32-bit words for each packet or message sent to the core. The first 32-bit word is a hardware buffer handle. The second 32-bit word represents several pieces of information as shown in Table 3-10.

Table 3-10. Second 32-bit Word For Resource Manager Communication Signaling to the Core

R ¹		Exception Code																						Destination Local Identifier									
3	1	3	0																					1	9								0

1. This bit is **reserved** and is always set to one.

Bit 31 of this second word must be always set to one. The bits zero through nine—a total of 10 bits—represent the destination local identifier. This is the same as the value used to create a communication ID. If this value is less than `IX_COMM_UBLOCK_ID_NUMBER` then the data is routed back to the microcode as these corresponding communications IDs are reserved for core to microblock communication. The bits 10 through 30—a total of 21 bits—are used to communicate an exception code.

Both of these 32-bit words must be non-zero. The dispatch thread extracts the destination and exception code, and send the buffer handle and the exception code as user data to the specified destination communication ID. If that communication ID has a callback registered for processing, then that function is invoked in the context of the dispatch thread. The programmer should take care about data consistency in this case. If that communication ID is in the get-select mode, then the dispatch thread queues the data into an internal queue, awakens all threads waiting for data on that communication ID and resumes data processing immediately. In this case, the processing of the message or packet is done in a different thread than the dispatch thread.

To send data from the core to the microblocks application code creates—using the provided macros—a communication ID for the destination microblock. There is a one-to-one correspondence between microblock IDs and the local communication IDs. Part of the initialization of core to microblock communication is left as an application responsibility. This spares system resources and provides extended flexibility.

There are two ways to setup a microblock communication ID for core to microcode communication.

In the first approach the calling application creates a hardware scratch ring—or an SRAM ring, but this option is slower—and assigns it as the communication ring dedicated to communication with a particular microblock using the `ix_rm_ublock_packet_comm_init()` or `ix_rm_ublock_message_comm_init()` functions. At this point a specific data handler is registered with the communication ID. When data is sent to the communication ID using the regular send functions—`ix_rm_packet_send()`, `ix_rm_packet_send_wait()`, `ix_rm_message_send()`, or `ix_rm_message_send_wait()`, the registered data handler is invoked storing the passed buffer handle and user data onto the associated ring.

Packet and message communication to the microblocks must be done through different rings—the same ring cannot be used for both packet and message communication. However, all communication IDs can perform packet or message communication—but *not message and packet communication*—through the same hardware ring. Before the microcode is loaded into the microstore, the ring ID must be patched into the microcode. If a specific core to microblock communication is not properly initialized, then calls to the send functions result in the sent buffers being returned to the free lists from which they originated.

The second method requires application code to register a custom packet or message handler with the microblock communication ID. Using this approach, for each packet or message, any amount of data can be transferred to the microcode through any means available.

It is up to a product development team to decide what kind of data are put on the rings. Usually the buffer handle from the head of a chain—representing a packet—suffices.

Table 3-11 lists the functions and data structures in the Communication API.

Table 3-11. Resource Manager Communication API

Name	Description
<code>ix_comm_data_handler</code>	Generic function type for data communication handler.
<code>ix_communication_id</code>	Generic type used for the identification of a communication point.
<code>ix_comm_select_action_set</code>	Array of core local communication ID masks that is passed to the select function.
<code>ix_comm_id_mode</code>	Enumerated type expressing a communication ID receive mode.
<code>IX_RM_COMM_ID_GET_LOCAL_ID()</code>	Returns the local ID for a communication ID.
<code>IX_RM_COMM_ID_GET_SYSTEM_TYPE()</code>	Returns the subsystem type for a communication ID.
<code>IX_RM_COMM_ID_GET_BLADE_ID()</code>	Returns the blade ID for a communication ID.
<code>IX_RM_COMM_MAKE_ID()</code>	Creates a communication ID.
<code>IX_RM_COMM_MAKE_LOCAL_ID()</code>	Creates a local subsystem communication ID.
<code>ix_rm_packet_set_receive_mode()</code>	This function sets the packet receive mode for the communication ID.
<code>ix_rm_message_set_receive_mode()</code>	This function sets the message receive mode for the communication ID.
<code>ix_rm_packet_set_consumer_mode()</code>	This function sets the packet consumer mode for the communication ID.
<code>ix_rm_message_set_consumer_mode()</code>	This function sets the message consumer mode for the communication ID.
<code>ix_rm_packet_set_producer_mode()</code>	This function sets the packet producer mode for the communication ID.
<code>ix_rm_message_set_producer_mode()</code>	This function sets the message producer mode for the communication ID.
<code>ix_rm_packet_handler_register()</code>	Registers a packet handler with a core communication ID.
<code>ix_rm_message_handler_register()</code>	Registers a message handler with a core communication ID.
<code>ix_rm_packet_handler_unregister()</code>	Puts the packet processing for the communication ID in default mode—it drops packets.
<code>ix_rm_message_handler_unregister()</code>	Puts the message processing for the communication ID in default mode—it drops packets.
<code>ix_rm_packet_send()</code>	Sends a packet to a destination.
<code>ix_rm_packet_send_wait()</code>	Sends a packet to a destination in a blocking mode.
<code>ix_rm_message_send()</code>	Sends a message to a destination.
<code>ix_rm_message_send_wait()</code>	Sends a message to a destination in a blocking mode.
<code>ix_rm_packet_peek()</code>	This function retrieves the number of packets stored in the internal queue for the specified communication ID.
<code>ix_rm_packet_get()</code>	Retrieves a packet from a communication ID in a non-blocking mode.
<code>ix_rm_packet_get_wait()</code>	Retrieves a packet from a communication ID in a blocking mode.
<code>ix_rm_message_peek()</code>	This function retrieves the number of messages stored in the internal queue for the specified communication ID.

Table 3-11. Resource Manager Communication API (Continued)

Name	Description
<code>ix_rm_message_get()</code>	Retrieves a message from a communication ID in a non-blocking mode.
<code>ix_rm_message_get_wait()</code>	Retrieves a message from a communication ID in a blocking mode.
<code>ix_rm_comm_select()</code>	Waits on a set of communication IDs for data to be available.
<code>ix_rm_ublock_packet_comm_init()</code>	Initializes the packet communication to a microblock.
<code>ix_rm_ublock_message_comm_init()</code>	Initializes the message communication to a microblock.

For the communication between core components there are two mutually exclusive ways to receive packets and messages:

- Through callbacks
- By waiting for data then retrieving data from the communication IDs

The second case is at all times buffered—that is, the data are temporarily stored in an internal queue. On the other hand, callbacks may be buffered or unbuffered based on the implementation.

At the creation time, the core communication IDs are in the callback receive mode—`IX_COMM_ID_MODE_CALLBACK`—and drops the packets and messages received. From this default state, they can be put in either callback mode—`IX_COMM_ID_MODE_CALLBACK`—or get-select mode—`IX_COMM_ID_MODE_GET_SELECT`—by calls to `ix_rm_packet_set_receive_mode()` and `ix_rm_message_set_receive_mode()` functions. Usually there is no need to go from one mode to another but if that is required it can be done.

If a communication ID is in get-select mode and it is switched to callback mode, all buffers in the internal queue are dropped and all waiting tasks are awakened and the communication ID is set to the callback mode. A separate call to `ix_rm_packet_handler_register()` or `ix_rm_message_handler_register()` should be made in order to install the desired callback function.

When the communication ID is switched from the callback mode to the get-select mode, all data from that point on is queued in the internal queue associated with the communication ID. Once in the get-select mode, calls to `ix_rm_packet_get()`, `ix_rm_packet_get_wait()`, `ix_rm_message_get()`, `ix_rm_message_get_wait()`, `ix_rm_packet_peek()`, `ix_rm_message_peek()`, and `ix_rm_comm_select()` functions are allowed.

In get-select receiving mode, a communication ID can be in different consumer/producer modes. At the time a communication ID is put in get-select receiving mode the producer/consumer mode is automatically set to multiproducer/multiconsumer.

Note: When in the get-select mode, the Resource Manager assures data consistency where multiple threads send to one communication ID at the same time and where multiple threads consume from one communication ID at the same time. The get-select mode has speed penalties due to the need to lock access to the internal queue for all put and get operations.

Through calls to `ix_rm_packet_set_consumer_mode()`, `ix_rm_packet_set_producer_mode()`, `ix_rm_message_set_consumer_mode()`, and `ix_rm_message_set_producer_mode()` functions, the default producer/consumer mode can be changed.

Note: A communication ID can be in one mode for packet communication and another for message communication.

The communication modes are strictly related to the receive side of a communication ID but they affect the send-side behavior.

3.6.1 Defined Types, Enumerations, and Data Structures

This section defines the data types and structures used in the Communication API.

3.6.1.1 `ix_comm_data_handler`

This is the prototype for the generic data handler function pointer. The data handler function takes as parameters a buffer handle—which in general refers to a packet or message handle, a 32-bit unsigned integer where extra user data are passed to the handler, and a context pointer that is passed to Resource Manager at the time the data handler is registered with a particular communication ID and that is passed as an argument to the data handler upon each invocation.

When data are sent to a communication ID through `send_packet` or `send_message` functions, on the receiving side, the registered handler is invoked. The `send_packet` and `send_message` have an `arg_UserData` parameter that is passed as the `arg_UserData` argument to the registered handler for the packet or message communication ID. This type can be used as a parameter for all registration functions but in the actual implementation of the handler the first parameter can be specialized to alias `ix_handle` types in order to avoid any confusion for the programmer. This type describes the data handler function type that can be registered as a callback with a communication ID.

C Syntax

```
typedef ix_error (*ix_comm_data_handler)(
    ix_buffer_handle arg_hBuffer,
    ix_uint32 arg_UserData,
    void* arg_pContext);
```

3.6.1.2 `ix_communication_id`

This type describes a communication ID. The communication ID includes three bit fields—a local communication ID, a system type and the blade ID. Macros are provided for constructing communication IDs and for accessing different bit fields.

The local communication IDs are used in the following manner: the IDs ranging from zero to `IX_COMM_LAST_UBLOCK_ID`, inclusive, are reserved for communication with the microblocks. A packet or message whose destination is an ID in this range is sent to the microblock corresponding to the ID number.

Internally this is a 32-bit unsigned integer that has three bit fields packed within it—the local identifier, the subsystem identifier, and the blade identifier (also known as the system identifier).

The ID range from `IX_COMM_LAST_UBLOCK_ID` plus one to `IX_COMM_LOCAL_ID_NUMBER`, inclusive, are used for communication with core components. Microcode sends the exception packets and messages to one of these IDs on which core components listens.

Note: A blade ID of zero always identifies the current system so it should not be used in identifying a particular system.

C Syntax

```
typedef ix_uint32 ix_communication_id;
```

Field Mapping

Table 3-12 shows the mapping of the bit fields contained within `ix_communication_id`.

Table 3-12. Bit Field Mapping for `ix_communication_id`

Field	Description	Bit Range ¹	
		Least Significant Byte	Most Significant Byte
Local ID	The local communication identifier.	IX_COMM_ID_LOCAL_ID_LSB	IX_COMM_ID_LOCAL_ID_MSB
Subsystem Type	The subsystem type used mainly for remote communication.	IX_COMM_ID_SUBSYSTEM_TYPE_LSB	IX_COMM_ID_SUBSYSTEM_TYPE_MSB
Blade ID	The blade identifier used mainly for remote communication.	IX_COMM_ID_BLADE_ID_LSB	IX_COMM_ID_BLADE_ID_MSB

1. The size of the bit field can be computed using the formula: `Most_Significant_Bit + 1 - Least_Significant_Bit`.

3.6.1.2.1 `ix_comm_select_action_set`

This type represents an array of core local communication ID masks that is passed to the select function—`ix_rm_comm_select()`. The index zero in the array corresponds to the first core communication ID—select does not work on the reserved microblock communication IDs. If the caller to `ix_rm_comm_select()` wants to be notified on the arrival of packets or messages on a communication ID then the corresponding mask should be set with the appropriate flags, described in this section. On return from the `ix_rm_comm_select()` function for each communication ID the action set is cleared and the flags in the set of masks are set only if data are available on that communication ID or when an error occurs on that ID.

C Syntax

```
typedef ix_bit_mask32 ix_comm_select_action_set[IX_COMM_CORE_ID_NUMBER];
```

`IX_COMM_SELECT_PACKET_MASK`

```
#define IX_COMM_SELECT_PACKET_MASK 0x1
```

This flag specifies interest in packet processing for a specific communication ID. If this flag is set in an `ix_comm_select_action_set` mask at the index corresponding to a communication ID then the function returns if packets have been received on that communication ID. On return the flag is set only if packets have been received on the specified communication ID.

IX_COMM_SELECT_MESSAGE_MASK

```
#define IX_COMM_SELECT_MESSAGE_MASK          0x2
```

This flag specifies interest in message processing for a specific communication ID. If this flag is set in an `ix_comm_select_action_set` mask at the index corresponding to a communication ID then the function returns if messages have been received on the communication ID. On return the flag is set only if messages have been received on the communication ID.

IX_COMM_SELECT_ERROR_MASK

```
#define IX_COMM_SELECT_ERROR_MASK           0x4
```

On return from the `ix_rm_comm_select()` call this flag specifies that an error occurred for the specified communication ID.

3.6.1.2.2 ix_comm_id_mode

This enumerated type describes communication ID receive modes. In the get-select mode the sent data is stored in an internal queue to be retrieved at a later time by the receiving side. In the callback mode the data is passed directly to a callback registered by the receiving side.

At initialization time all the communication IDs are in callback mode. Once the calling program registers a callback with the communication ID the callback is invoked for each buffer sent to that destination.

The get-select mode automatically registers an internal callback with the communication ID—this callback stores the sent buffer and application data using an internal queue and returns immediately. The data associated with that communication ID can be retrieved using `ix_rm_packet_get()`, `ix_rm_message_get()`, `ix_rm_comm_select()`, `ix_rm_packet_peek()`, and `ix_rm_message_peek()` calls referencing that communication ID.

C Syntax

```
typedef enum ix_e_comm_id_mode {
    IX_COMM_ID_MODE_FIRST = 0,
    IX_COMM_ID_MODE_CALLBACK = IX_COMM_ID_MODE_FIRST,
    IX_COMM_ID_MODE_GET_SELECT,
    IX_COMM_ID_MODE_LAST
} ix_comm_id_mode;
```

3.6.2 API Functions**3.6.2.1 Helper Macros**

The macros in this section are provided for creating communication IDs and for accessing the different bit fields.

3.6.2.1.1 IX_RM_COMM_ID_GET_LOCAL_ID()

This macro returns the local ID for a communication ID.

C Syntax

```
#define IX_RM_COMM_ID_GET_LOCAL_ID(arg_CommID)
```

3.6.2.1.2 `IX_RM_COMM_ID_GET_SYSTEM_TYPE()`

This macro returns the subsystem type for a communication ID.

C Syntax

```
#define IX_RM_COMM_ID_GET_SYSTEM_TYPE(arg_CommID)
```

3.6.2.1.3 `IX_RM_COMM_ID_GET_BLADE_ID()`

This macro returns the blade ID for a communication ID.

C Syntax

```
#define IX_RM_COMM_ID_GET_BLADE_ID(arg_CommID)
```

3.6.2.1.4 `IX_RM_COMM_MAKE_ID()`

This macro creates a communication ID. The caller specifies the local ID, subsystem type, and blade ID for the target of the communication.

Note: A blade ID of zero always identifies the current system so it should not be used in identifying a certain system.

C Syntax

```
#define IX_RM_COMM_MAKE_ID(arg_LocalID, arg_SubsystemType, arg_BladeID)
```

3.6.2.1.5 `IX_RM_COMM_MAKE_LOCAL_ID()`

This macro creates a local subsystem communication ID.

C Syntax

```
#define IX_RM_COMM_MAKE_LOCAL_ID(arg_LocalID)
```

3.6.2.2 `ix_rm_packet_set_receive_mode()`

This function sets the packet receive mode for the communication ID—either `IX_COMM_ID_MODE_CALLBACK` or `IX_COMM_ID_MODE_GET_SELECT`. The change from one mode to the other should be done explicitly. If this communication ID is in get-select mode, all buffers in the internal queue are dropped and all waiting tasks are awakened.

C Syntax

```
ix_error ix_rm_packet_set_receive_mode(
    ix_communication_id arg_CommunicationId,
    ix_comm_id_mode arg_ReceiveMode);
```


Input

<code>arg_CommunicationId</code>	The communication ID whose packet receive mode is to be set.
<code>arg_ReceiveMode</code>	The packet receive mode. The valid values are: <ul style="list-style-type: none">• <code>IX_COMM_ID_MODE_CALLBACK</code>—callback mode• <code>IX_COMM_ID_MODE_GET_SELECT</code>—get-select mode

Output/Returns

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

3.6.2.3 `ix_rm_message_set_receive_mode()`

This function sets the message receive mode for the communication ID—either `IX_COMM_ID_MODE_CALLBACK` or `IX_COMM_ID_MODE_GET_SELECT`. The change from one mode to the other should be done explicitly. If this communication ID is in get-select mode, all buffers in the internal queue are dropped and all waiting tasks are awakened.

C Syntax

```
ix_error ix_rm_message_set_receive_mode(
    ix_communication_id arg_CommunicationId,
    ix_comm_id_mode arg_ReceiveMode);
```

Input

<code>arg_CommunicationId</code>	The communication ID for which we are setting the message receive mode.
<code>arg_ReceiveMode</code>	The message receive mode. Valid values are: <ul style="list-style-type: none"> • <code>IX_COMM_ID_MODE_CALLBACK</code>—callback mode • <code>IX_COMM_ID_MODE_GET_SELECT</code>—get-select mode

Output/Returns

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

3.6.2.4 `ix_rm_packet_set_consumer_mode()`

This function sets the packet consumer mode for the communication ID. If `arg_ConsumerMode` is zero then the communication ID is set to single consumer mode. Otherwise the multiple consumer mode is set. If the communication ID is in callback receive mode then the function returns an error because this mode applies just to the get-select receive mode.

C Syntax

```
ix_error ix_rm_packet_set_consumer_mode(
    ix_communication_id arg_CommunicationId,
    ix_uint32 arg_ConsumerMode);
```


Input

<code>arg_CommunicationId</code>	The communication ID for which we are setting the packet consumer mode.
<code>arg_ConsumerMode</code>	The consumer mode for the communication ID. Valid values are: <ul style="list-style-type: none"> • zero—single consumer • non-zero—multiple consumer

Output/Returns

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

3.6.2.5 `ix_rm_message_set_consumer_mode()`

This function sets the message consumer mode for the communication ID. If `arg_ConsumerMode` is zero then the communication ID is set to single consumer mode. Otherwise the multiple consumer mode is set. If the communication ID is in callback receive mode then the function returns an error because this mode applies just to the get-select receive mode.

C Syntax

```
ix_error ix_rm_message_set_consumer_mode(
    ix_communication_id arg_CommunicationId,
    ix_uint32 arg_ConsumerMode);
```

Input

<code>arg_CommunicationId</code>	The communication ID for which we are setting the message consumer mode.
<code>arg_ConsumerMode</code>	The consumer mode for the communication ID. Valid values are: <ul style="list-style-type: none"> • zero—single consumer • non-zero—multiple consumer

Output/Returns

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

3.6.2.6 `ix_rm_packet_set_producer_mode()`

This function sets the packet producer mode for the communication ID. If `arg_ConsumerMode` is zero then the communication ID is set to single-producer mode. Otherwise the multiple-producer mode is set. If the communication ID is in callback receive mode then the function returns an error because this mode only applies to the get-select receive mode.

C Syntax

```
ix_error ix_rm_packet_set_producer_mode(
    ix_communication_id arg_CommunicationId,
    ix_uint32 arg_ProducerMode);
```

Input

<code>arg_CommunicationId</code>	The communication ID for which we are setting the packet producer mode.
<code>arg_ProducerMode</code>	The producer mode for the communication ID. Valid values are: <ul style="list-style-type: none"> • zero—single producer • non-zero—multiple producer

Output/Returns

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

3.6.2.7 `ix_rm_message_set_producer_mode()`

This function sets the message producer mode for the communication ID. If `arg_ConsumerMode` is zero then the communication ID is set to single-producer mode. Otherwise the multiple-producer mode is set. If the communication ID is in callback receive mode then the function returns an error because this mode only applies to the get-select receive mode.

C Syntax

```
ix_error ix_rm_message_set_producer_mode(
    ix_communication_id arg_CommunicationId,
    ix_uint32 arg_ProducerMode);
```

Input

<code>arg_CommunicationId</code>	The communication ID for which the message producer mode is set.
<code>arg_ProducerMode</code>	The producer mode for the communication ID. Valid values are: <ul style="list-style-type: none"> • zero—single-producer mode • non-zero—multiple-producer mode

Output/Returns

Return Value IX_SUCCESS if successful or a valid ix_error token for failure.

3.6.2.8 ix_rm_packet_handler_register()

This function registers a packet handler for an Intel XScale® core communication ID. From this point on, the newly registered handler is used for processing packets sent to this destination.

The calling application passes a data handler and an application-defined context pointer. If the call has been successful all packets sent to the specific communication ID are received by the registered callback function. The registered handler is invoked with the registered context as a parameter. The processing should be done as fast as possible due to the fact that this function might be executed from a context of the sender or of a processing engine. For more extensive processing, the packet should be queued for further handling in a separate task.

C Syntax

```
ix_error ix_rm_packet_handler_register(
    ix_communication_id arg_CommunicationId,
    ix_comm_data_handler arg_PacketHandler,
    void* arg_pContext);
```

Input

arg_CommunicationId	The communication ID for which to register the packet handler.
arg_PacketHandler	The packet handler to register.
arg_pContext	The context data to pass to the packet handler each time this function is invoked. The calling application must ensure that the context data are valid as long the packet handler is registered. The context is application-defined.

Output/Returns

Return Value Returns IX_SUCCESS if successful or a valid ix_error for failure.

3.6.2.9 `ix_rm_packet_handler_unregister()`

This function unregisters the current packet handler for the specified communication ID and installs the Resource Manager default handler. From this point on, the packets sent to this communication ID undergo the default processing—at this time the default is to drop the packet. The composing buffers are returned to the free list from which they were allocated.

C Syntax

```
ix_error ix_rm_packet_handler_unregister(
    ix_communication_id arg_CommunicationId);
```

Input

`arg_CommunicationId` The communication ID for which we unregister the packet handler.

Output/Returns

Return Value Returns `IX_SUCCESS` if successful or a valid `ix_error` for failure.

3.6.2.10 `ix_rm_message_handler_register()`

This function registers a message handler for a Intel XScale® core communication ID. The application passes a data handler and an application-defined context pointer. From this point on, the new registered handler is used for processing messages sent to this destination.

If this communication ID is in get-select mode then an error is returned. The receive mode for the communication ID should be changed explicitly.

If a call to this function has been successful all messages sent to the corresponding communication ID are received by the registered callback function. The registered handler is invoked with the registered context as a parameter. The processing should be done as fast as possible due to the fact that this function might be executed from a context of the sender or of a processing engine. For more extensive processing, the message should be queued for further handling in a separate task.

C Syntax

```
ix_error ix_rm_message_handler_register(
    ix_communication_id arg_CommunicationId,
    ix_comm_data_handler arg_MessageHandler,
    void* arg_pContext);
```


Input

<code>arg_CommunicationId</code>	The communication ID for which the message handler is registered.
<code>arg_MessageHandler</code>	The message handler to register.
<code>arg_pContext</code>	The context data that is passed to the message handler on each invocation. The application must ensure that the context data are valid as long the message handler is registered.

Output/Returns

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

3.6.2.11 `ix_rm_message_handler_unregister()`

This function unregisters the current message handler for the specified communication ID and installs the Resource Manager default handler. From this point on, the messages sent to this communication ID undergo the default processing which at this time is to drop the message. The composing buffers are returned to the free list from which they were allocated.

C Syntax

```
ix_error ix_rm_message_handler_unregister(
    ix_communication_id arg_CommunicationId);
```

Input

<code>arg_CommunicationId</code>	The communication ID from which to unregister the message handler.
----------------------------------	--

Output/Returns

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

3.6.2.12 ix_rm_packet_send()

This function sends a packet to the destination communication ID in a non-blocking mode. If the communication ID is in get-select mode and the internal package queue is full an error is returned specifying that the internal queue is full.

C Syntax

```
ix_error ix_rm_packet_send(  
    ix_communication_id arg_CommunicationId,  
    ix_buffer_handle arg_MessageBuffer,  
    ix_uint32 arg_UserData);
```

Input

arg_CommunicationId	The destination communication ID.
arg_MessageBuffer	The first buffer handle from the message that is sent to the specified communication ID.
arg_UserData	The custom user data that is passed to the packet handler registered with the specified communication ID. This argument can be used to pass extra information to the destination.

Output/Returns

Return Value	Returns IX_SUCCESS if successful or a valid ix_error for failure.
--------------	---

3.6.2.13 `ix_rm_packet_send_wait()`

This function sends a packet to the specified communication ID—the destination communication ID in a blocking mode. If the communication ID is in get-select mode and the internal packet queue is full then the send function waits for the specified timeout for the internal queue to become available before returning an error.

C Syntax

```
ix_error ix_rm_packet_send_wait(
    ix_communication_id arg_CommunicationId,
    ix_buffer_handle arg_PacketBuffer,
    ix_uint32 arg_UserData,
    ix_uint32 arg_Timeout);
```

Input

<code>arg_CommunicationId</code>	The destination communication ID.
<code>arg_PacketBuffer</code>	The first buffer handle from the packet that is sent to the specified communication ID.
<code>arg_UserData</code>	The custom user data that is passed to the packet handler registered with the specified communication ID. This argument can be used to pass extra information to the destination.
<code>arg_Timeout</code>	The timeout specifying how long to wait from the time the communication ID goes into buffered mode until the time the internal data queue becomes available—expressed in milliseconds.

Output/Returns

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

3.6.2.14 `ix_rm_message_send()`

This function sends a message to the destination communication ID in a non-blocking mode. If the communication ID is in get-select mode and the internal message queue is full then an error is returned specifying that the internal queue is full.

C Syntax

```
ix_error ix_rm_message_send(
    ix_communication_id arg_CommunicationId,
    ix_buffer_handle arg_MessageBuffer,
    ix_uint32 arg_UserData);
```

Input

<code>arg_CommunicationId</code>	The destination communication ID.
<code>arg_MessageBuffer</code>	The first buffer handle from the message that is sent to the specified communication ID.
<code>arg_UserData</code>	The custom user data that is passed to the message handler registered with the specified communication ID. This argument can be used to pass extra information to the destination.

Output/Returns

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

3.6.2.15 `ix_rm_message_send_wait()`

This function sends a message to the specified communication ID in a blocking mode. If the communication ID is in get-select mode and the internal message queue is full then the send function waits for the specified timeout—expressed in milliseconds—for the internal queue to become available before returning an error.

C Syntax

```
ix_error ix_rm_message_send_wait(
    ix_communication_id arg_CommunicationId,
    ix_buffer_handle arg_MessageBuffer,
    ix_uint32 arg_UserData,
    ix_uint32 arg_Timeout);
```


Input

<code>arg_CommunicationId</code>	The destination communication ID.
<code>arg_MessageBuffer</code>	The first buffer handle from the message that is sent to the specified communication ID.
<code>arg_UserData</code>	The custom user data that is passed to the message handler registered with the specified communication ID. This argument can be used to pass extra information to the destination.
<code>arg_Timeout</code>	The timeout specifying how long to wait from the time the communication ID goes into get-select mode until the time the internal data queue becomes available—expressed in milliseconds.

Output/Returns

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

3.6.2.16 `ix_rm_packet_peek()`

This function retrieves the number of packets stored in the internal queue for the specified communication ID. The number might not be the same after the function returns. If the communication ID is in callback receive mode, then an error is returned.

C Syntax

```
ix_error ix_rm_packet_peek(
    ix_communication_id arg_CommunicationId,
    ix_uint32* arg_pAvailablePackets);
```

Input

<code>arg_CommunicationId</code>	The local communication ID from which we want to get information.
----------------------------------	---

Output/Returns

<code>arg_pAvailablePackets</code>	A pointer to the number of available packets.
Return Value	Returns <code>IX_SUCCESS</code> if successful or a valid <code>ix_error</code> for failure.

3.6.2.17 `ix_rm_packet_get()`

This function retrieves the next packet and associated user data from a communication ID in a non-blocking, get-select mode. If the communication ID is in callback mode, the function returns an error. If the internal packet queue is empty then an error is returned immediately.

C Syntax

```
ix_error ix_rm_packet_get(
    ix_communication_id arg_CommunicationId,
    ix_buffer_handle* arg_pPacketBuffer,
    ix_uint32* arg_UserData);
```

Input

<code>arg_CommunicationId</code>	The local communication ID specifying the packet source.
<code>arg_pPacketBuffer</code>	A pointer to the packet buffer handle. The head of the internal queue is extracted and stored at this location.
<code>arg_pUserData</code>	A pointer specifying where to return the associated user data.

Output/Returns

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

3.6.2.18 `ix_rm_packet_get_wait()`

This function retrieves the next packet and associated user data from a get-select communication ID in blocking mode. If the communication ID is in callback receive mode then an error is returned. If the internal packet queue is empty then the function waits for the specified timeout—expressed in milliseconds—for the data to become available before returning an error.

This function can be called from different contexts for the same communication ID. When the data token becomes available only one of the instances returns and all the others remain blocked until enough data are available or they timeout. This behavior can be useful for the *pool of threads* processing design.

C Syntax

```
ix_error ix_rm_packet_get_wait(
    ix_communication_id arg_CommunicationId,
    ix_buffer_handle* arg_pPacketBuffer,
    ix_uint32* arg_UserData,
    ix_uint32 arg_Timeout);
```


Input

<code>arg_CommunicationId</code>	The local communication ID from which we want to get data.
<code>arg_pPacketBuffer</code>	A pointer to the packet buffer handle. The head of the internal queue is extracted and stored at this location.
<code>arg_pUserData</code>	A pointer specifying where to return the associated user data.
<code>arg_Timeout</code>	The timeout specifying how long to wait—when the communication ID is in get-select mode—for data to become available. This value is expressed in milliseconds.

Output/Returns

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

3.6.2.19 `ix_rm_message_peek()`

This function retrieves the number of messages stored in the internal queue for the specified communication ID. The number might not be the same after the function returns. If the communication ID is in callback receive mode an error is returned.

C Syntax

```
ix_error ix_rm_message_peek(
    ix_communication_id arg_CommunicationId,
    ix_uint32* arg_pAvailableMessages);
```

Input

<code>arg_CommunicationId</code>	The local communication ID from which to get data.
----------------------------------	--

Output/Returns

<code>arg_pAvailableMessages</code>	A pointer to the number of available messages.
Return Value	Returns <code>IX_SUCCESS</code> if successful or a valid <code>ix_error</code> token for failure.

3.6.2.20 ix_rm_message_get()

This function retrieves the next message and associated user data from a get-select communication ID in non-blocking mode. If the communication ID is not already in the get-select mode, the application must explicitly switch the communication ID to the get-select mode before calling this function.

If this function is called when the internal message queue is empty then an error is returned immediately. If this function is called when the communication ID is in callback receive mode then the this function is called when an error is returned.

C Syntax

```
ix_error ix_rm_message_get(
    ix_communication_id arg_CommunicationId,
    ix_buffer_handle* arg_pMessageBuffer,
    ix_uint32* arg_UserData);
```

Input

<code>arg_CommunicationId</code>	The local communication ID from which to get data.
<code>arg_pMessageBuffer</code>	A pointer to the message buffer handle. The head of the internal queue is extracted and stored at this location.
<code>arg_pUserData</code>	A pointer specifying where to return the associated user data.

Output/Returns

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

3.6.2.21 ix_rm_message_get_wait()

This function retrieves the next message and associated user data from a get-select communication ID in blocking mode. If this function is called when the internal message queue is empty then the function waits for the specified timeout—expressed in milliseconds—for the data to become available before returning an error. If this function is called when the communication ID is in callback mode, the function returns an error.

This function can be called from different contexts for the same communication ID. When a data token becomes available only one of the instances returns and all the others remain blocked until enough data are available or they timeout. This behavior can be useful for the *pool of threads* processing design.

C Syntax

```
ix_error ix_rm_message_get_wait(
    ix_communication_id arg_CommunicationId,
```



```
ix_buffer_handle* arg_pMessageBuffer,  
ix_uint32* arg_UserData,  
ix_uint32 arg_Timeout);
```

Input

<code>arg_CommunicationId</code>	The local communication ID from which to get data.
<code>arg_pMessageBuffer</code>	The address where the first buffer handle from the message from the head of the internal queue is extracted.
<code>arg_pUserData</code>	The address where the associated user data are stored.
<code>arg_Timeout</code>	The timeout specifying how long to wait—when the communication ID is in get-select mode—for data to become available. This value is expressed in milliseconds.

Output/Returns

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

3.6.2.22 ix_rm_comm_select()

This function waits on a local set of core component communication IDs for messages or packets to become available. The `arg_SelectActionSet` parameter is an array of 32-bit flags specifying what kind of data each of the specified communication IDs is waiting for. If one of the selected communication IDs has the type of data for which interest has been shown in the passed action set, the function returns immediately. If the caller wants to be notified on the arrival of packets or messages on a communication ID then the corresponding mask should be set with the appropriate flags.

The first element of the array corresponds to the first core communication ID—this function does not work with microblock communication IDs. The function waits for data for a maximum time specified by the `arg_Timeout` parameter. If after this time has elapsed no data are available the function returns an error. On return, the whole select action set is flushed and each mask is updated to specify if data are available or an error has occurred on that communication ID.

This function can be invoked from several contexts at the same time for the same set of communication IDs. All contexts awaken but the following non-blocking get functions succeed as long data are available on the specific communication ID.

C Syntax

```
ix_error ix_rm_comm_select(
    ix_comm_select_action_set arg_SelectActionSet,
    ix_uint32 arg_Timeout);
```

Input

<code>arg_Timeout</code>	The timeout specifying how long to wait for data to become available. This value is expressed in milliseconds.
--------------------------	--

Input/Output

<code>arg_SelectActionSet</code>	The array of 32-bit flags specifying what kind of data each of the specified communication IDs is waiting for. On return, the flags are cleared and specify only the data type that has been received on each local core component communication ID.
----------------------------------	--

Output/Returns

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

3.6.2.23 `ix_rm_ublock_packet_comm_init()`

This function initializes the packet communication from the Intel XScale® core to the microblock corresponding to the passed communication ID. The function accomplishes this by associating a hardware ring with a communication ID corresponding to the target microblock. The registered ring has to be passed to the microcode.

This function must be called before packet communication to the specific microblock is started. The passed hardware ring is associated with the communication ID and from this point on all packets sent on this ID are queued into the specified ring.

Packet communication to all microblocks can be done on the same hardware ring, in which case extra information about the destination has to be passed to the microcode along with the data. For each packet sent to the communication ID the buffer handle and user data passed into the function call is stored onto the associated ring.

If the microblock communication IDs have no ring associated with them and no data handler has been previously registered, then the buffers are dropped.

C Syntax

```
ix_error ix_rm_ublock_packet_comm_init(  
    ix_communication_id arg_CommunicationId,  
    ix_hw_ring_handle arg_hHwRing);
```

Input

<code>arg_CommunicationId</code>	The microblock communication ID of interest.
<code>arg_hHwRing</code>	The hardware ring through which packet communication from the core to microblock is performed.

Output/Returns

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

3.6.2.24 ix_rm_ublock_message_comm_init()

This function initializes the message communication from the Intel XScale® core to the microblock corresponding to the communication ID passed in as an argument. This function must be called before message communication to the specific microblock is started. The passed hardware ring is associated with the communication ID and, from this point on, all messages sent on this ID are queued into the specified ring. The registered ring has to be passed to the microcode.

Message communication to all microblocks can be done on the same hardware ring, in which case extra information about the destination must be passed along with the data to the microcode. For each message sent to the communication ID the passed buffer handle and user data are stored onto the associated ring.

If the microblock communication IDs have no ring associated with them and no data handler has been previously registered, then the buffers are dropped.

C Syntax

```
ix_error ix_rm_ublock_message_comm_init(  
    ix_communication_id arg_CommunicationId,  
    ix_hw_ring_handle arg_hHwRing);
```

Input

arg_CommunicationId	The microblock communication ID of interest.
arg_hHwRing	The hardware ring through which message communication between the Intel XScale® core and a microblock is performed.

Output/Returns

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

3.7 Remote Communication Extension API

As stated earlier the communication ID mechanism allows for local communication, as well as remote communication with other systems. If the destination is not intended for the local system then the messages and packets will be forwarded to the active remote communication service that routes the data through PCI or another communication pathway to the remote system.

A region of the communication ID identifies the remote system using a blade identifier. Each system that works in a group should have a unique blade ID. A value of zero for the blade ID field in the communication ID always specifies the local system—though it could be a dual ingress/egress system since a portion of the communication ID specifies the destination subsystem in the case of a dual ingress/egress system.

When the blade ID is zero—specifying the local system—then a subsystem type value of zero at the time of communication ID creation refers to the local subsystem, whereas a value of one refers to the peer subsystem.

When the blade ID specifies a remote system—that is, it has a value other than zero—then a value of zero for the subsystem type specifies the ingress subsystem whereas a value of one specifies the egress subsystem.

When data is sent to a communication ID that does not belong to the local subsystem—remote communication ID—then the data is forwarded to the active remote communication service. At the startup, the installed remote communication service is just dropping all packets and messages received. Once a remote communication service is installed, then all data is passed to the registered data handlers for the remote communication service. These handlers will receive the same information as any local data handlers—a buffer plus user data—plus the destination communication ID. From this point is the responsibility of the implementer of the service to handle the remote communication. For a single dual system, there is a predefined service that will facilitate the communication between ingress and egress through PCI bus. The API call `ix_rm_init_pci_remote_communication()` has been provided to easily register this remote communication service.

Table 3-13 summarizes the remote communication extension API.

Table 3-13. Resource Manager Remote Communication Extension API

Name	Description
<code>ix_remote_comm_service</code>	This structure defines a remote communication service.
<code>ix_remote_comm_data_handler</code>	This callback function prototype represents a generic remote data handler.
<code>ix_remote_comm_service_initializer</code>	Defines the generic remote communication service initializer.
<code>ix_remote_comm_service_finalizer</code>	Defines the generic remote communication service finalizer.
<code>ix_rm_remote_comm_service_register()</code>	Registers a remote communication service.
<code>ix_rm_remote_comm_service_unregister()</code>	Unregisters the active remote communication service.

Table 3-13. Resource Manager Remote Communication Extension API (Continued)

Name	Description
<code>ix_rm_init_pci_remote_communication()</code>	Installs the PCI remote communication service between the ingress and egress network processors for a single ingress/egress system.
<code>ix_rm_register_pci_communication_hw_free_list()</code>	Registers a hardware free list of choice to be used by the predefined dual single system PCI remote communication service.
<code>ix_rm_unregister_pci_communication_hw_free_list()</code>	Reverts to the default hardware free list to be used by the predefined dual single system PCI remote communication service.

3.7.1 Defined Types, Enumerations, and Data Structures

3.7.1.1 `ix_remote_comm_service`

This structure defines a remote communication service. The implementer of a remote communication service must provide one of these structures to the registration function, `ix_rm_remote_comm_service_register()`. The Resource Manager supports just one remote communication service at a time. However, the programmer can install one handler that can dispatch data to different destinations. The contexts must be valid for as long as this service is registered with the Resource Manager.

Members of this structure hold pointers to the service initializer and the service finalizer and a context shared by both, the service packet handler and its context, and the service message handler and its context.

All the functions describing the service should be properly defined. If these functions are not properly defined the behavior of the system is unpredictable.

C Syntax

```
typedef struct ix_s_remote_comm_service {
    ix_remote_comm_service_initializer  m_ServiceInitializer;
    ix_remote_comm_service_finalizer    m_ServiceFinalizer;
    void*                               m_pServiceContext;
    ix_remote_comm_data_handler          m_PacketHandler;
    void*                               m_pPacketHandlerContext;
    ix_remote_comm_data_handler          m_MessageHandler;
    void*                               m_pMessageHandlerContext;
} ix_remote_comm_service;
```


Data Members

<code>m_ServiceInitializer</code>	A pointer to the function called to initialize the service. This function must be provided by the calling application and provides all required initialization for the service.
<code>m_ServiceFinalizer</code>	A pointer to the function called to finalize the service. This function must be provided by the calling application and provides all required cleanup for the service.
<code>m_pServiceContext</code>	A pointer to the context passed to the service initialization and finalization functions.
<code>m_PacketHandler</code>	A pointer to the handler invoked for each packet sent to a remote target.
<code>m_pPacketHandlerContext</code>	A pointer to the context passed to the packet handler at each invocation.
<code>m_MessageHandler</code>	A pointer to the handler invoked for each message sent to a remote target.
<code>m_pMessageHandlerContext</code>	A pointer to the context passed to the message handler at each invocation.

3.7.2 Callback Function Prototypes

3.7.2.1 `ix_remote_comm_data_handler`

This callback function prototype represents a generic remote data handler.

When data is sent to a remote communication ID through `ix_rm_packet_send()`, `ix_rm_packet_send_wait()`, `ix_rm_message_send()`, and `ix_rm_message_send_wait()` functions, the registered remote handler is invoked. These functions have an `arg_UserData` parameter that is passed to the handler as the `arg_UserData` argument.

The implementer of a remote communication service must define two functions of this type—one to handle packet traffic and the other to handle message traffic. Any send function to a remote communication ID end up invoking the message or packet handler, as appropriate, registered with the active remote communication service.

If different processing is required for different destinations then the handler is responsible for the destination-sensitive behavior. Though the simplicity, a complex handling system can be created.

C Syntax

```
typedef ix_error (*ix_remote_comm_data_handler) (
    ix_buffer_handle arg_hBuffer,
    ix_uint32 arg_UserData,
    void* arg_pContext,
    ix_communication_id arg_CommId
```



```
);
```

Input

<code>arg_hBuffer</code>	Typically a handle to a packet or message buffer.
<code>arg_UserData</code>	An integer used to pass additional application data to the handler.
<code>arg_pContext</code>	The context pointer passed to Resource Manager at the time of remote communication set-up.
<code>arg_CommId</code>	A remote communication ID.

3.7.2.2 `ix_remote_comm_service_initializer`

Defines the generic remote communication service initializer. The implementer of a remote communication service must provide this function. This function must perform all required initialization for the remote communication service. At the time of the registration of the service—that is, when `ix_rm_remote_comm_service_register()` is invoked—this function is invoked with a new service context—the service context passed to Resource Manager at the time of the remote communication set-up. If this call succeeds, then this new remote communication service becomes active.

C Syntax

```
typedef ix_error (*ix_remote_comm_service_initializer)(void* arg_pContext);
```

Input

<code>arg_pContext</code>	A pointer to the context passed to the Resource Manager at the time the remote communication was set up.
---------------------------	--

3.7.2.3 `ix_remote_comm_service_finalizer`

Defines the generic remote communication service finalizer. The implementer of a remote communication service must provide this function. This function must perform all required cleanup for the remote communication service. At the time the service is unregistered—that is, when `ix_rm_remote_comm_service_unregister()` is invoked—this function is invoked with a service context—the service context passed to Resource Manager at the time of the remote communication set-up.

C Syntax

```
typedef ix_error (*ix_remote_comm_service_finalizer)(void* arg_pContext);
```


Input

`arg_pContext` A pointer to the context passed to the Resource Manager at the time the remote communication was set up.

3.7.3 API Functions

3.7.3.1 `ix_rm_remote_comm_service_register()`

This function registers a remote communication service if no other is already installed. If a service is already in place it must first be unregistered using `ix_rm_remote_comm_service_unregister()` and only then can the application register a new service. If the registration is possible then the initializer of the new service is invoked and, if that call is successful, the registration is completed.

At system initialization, a default service is registered that drops remote packets and messages. An error is returned if, at the time of this call, any other service is active other than the default service. The system must be brought to the initial state by unregistering the active service.

From the time this function returns, the new service handles all the remote traffic. The contexts passed for the service must be valid for as long as the service is active.

C Syntax

```
ix_error ix_rm_remote_comm_service_register (
    const ix_remote_comm_service* arg_pRemoteCommService);
```

Input

`arg_pRemoteCommService` The new remote comm service to be installed.

Output/Returns

Return Value Returns `IX_SUCCESS` if successful or a valid `ix_error` token to signal a failure.

3.7.3.2 `ix_rm_remote_comm_service_unregister()`

This function unregisters the active remote communication service. Upon return from this function the system is in the initial state. Before the service is unregistered, the finalizer of the service is invoked.

C Syntax

```
ix_error ix_rm_remote_comm_service_unregister(void);
```


Output/Returns

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

3.7.3.3 `ix_rm_init_pci_remote_communication()`

This function installs the PCI remote communication service between the ingress and egress network processors for a single ingress/egress system. Upon return from this call all communication to the peer subsystem is handled by the PCI remote communication service.

Note: This service works only for a single dual ingress/egress system.

C Syntax

```
ix_error ix_rm_init_pci_remote_communication(void);
```

Output/Returns

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

3.7.3.4 `ix_rm_register_pci_communication_hw_free_list()`

This function registers a hardware free list to be used by the receive side of the PCI remote communication driver instead of the default hardware free list. The PCI driver automatically creates an internal free list that is used for copying the buffers from the remote subsystem. This free list can be replaced by a free list of choice.

C Syntax

```
ix_error ix_rm_register_pci_communication_hw_free_list(  
    ix_buffer_free_list_handle arg_hFreeList);
```

Input

<code>arg_hFreeList</code>	The hardware free list to be used by the PCI remote communication driver.
----------------------------	---

Output/Returns

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

3.7.3.5 `ix_rm_unregister_pci_communication_hw_free_list()`

This function unregisters the previously registered user hardware free list for the PCI remote communication driver. From this point on, the default hardware free list (PCI driver internal list) is used to allocate the required buffers. The function returns the previously active user free list handle.

C Syntax

```
ix_error ix_rm_unregister_pci_communication_hw_free_list(  
    ix_buffer_free_list_handle* arg_pFreeListHandle);
```

Output/Returns

<code>arg_pFreeListHandle</code>	If successful, the previously registered user hardware free list handle is returned to the caller at the passed address.
Return Value	Returns <code>IX_SUCCESS</code> if successful or a valid <code>ix_error</code> token to signal a failure.

3.8 Memory Management API

The Resource Manager manages SRAM, DRAM, scratch and local memory for the IXP2400 and Intel® IXP2800 Network Processors. It exports an interface to allocate, access and free memory chunks. The Resource Manager owns SRAM and scratch memory completely and DRAM partially. The DRAM is partly owned by the operating system.

The memory managed by the Resource Manager is used to support system-wide data structures such as the buffer free pools, control blocks for building blocks, route table, trie table, and so on. The difference between this memory and memory allocated from the operating system is that this memory has no MMU protection and is always addressed at the same memory location by all processes. This has meaning only in Linux*. In VxWorks*, the memory model is a flat memory space shared by every task. Since this memory is shared with the microengines it is typically uncached.

All applications including the building block infrastructure must use this API to allocate memory in order to work in conjunction with the microengines. The operating system memory is not accessible from microcode.

Note: The Resource Manager memory management is designed to handle one-time memory allocation primarily to partition memory among the applications. In other words we are not attempting to re-implement `malloc()`. Applications that require handling large number of allocation and free operations dynamically need to obtain enough memory from the Resource Manager and manage it themselves.

The Resource Manager allocates memory such that DRAM is always returned aligned at an 8-byte boundary and any request is rounded off to an 8-byte boundary. For SRAM and scratch the alignment is always at a 4-byte boundary and all requests are rounded off to a 4-byte boundary.

Functions are provided to calculate the physical offset into the specific channel and physical address of the chunk allocated from the virtual address and vice versa. The microengines can only access memory using these offset values into a specific channel. The physical memory is contiguous—on a specific channel—for the IXP2400 and Intel® IXP2800 Network Processors.

Handling SRAM allocation requires supporting multiple channels. The memory management API takes the channel number as one of the inputs. For the Intel® IXP2400 Network Processor, the valid channel numbers are zero and one. For the Intel® IXP2800 Network Processor, the valid channel numbers are zero through three, inclusive.

For the Intel® IXP2800 Network Processor, the number of DRAM channels is three, while Intel® IXP2400 Network Processor supports only one DRAM channel.

The Resource Manager—at the time of initialization—can reserve space in SRAM, SCRATCH, SDRAM and local memory to support Microengine C. The linker (UCLD) requires a base address for these blocks of memory, but no provision exists to limit the size of these memory chunks. The Resource Manager gets information from the UCLO (loader) library about the memory required by the loaded microcode image and checks if that memory has been reserved at Resource Manager initialization. If any of the required areas have not been reserved, then an error is returned along with the memory areas that must be reserved for microcode usage. More details can be found in [Section 3.2.1.7, “ix_memory_reserved_area.”](#)

[Table 3-14](#) lists the structures and functions in the Memory Management API.

Table 3-14. Resource Manager Memory Management API

Name	Description
<code>ix_memory_type</code>	An enumerated type listing the types of memory supported by the Resource Manager.
<code>ix_memory_info</code>	Memory information data structure.
<code>ix_memory_alignment_type</code>	Alignment types for the aligned memory allocation and reservation calls.
<code>ix_rm_mem_alloc()</code>	Allocates memory—SRAM, DRAM, and scratch.
<code>ix_rm_mem_alloc_aligned()</code>	Allocates memory with alignment—SRAM, DRAM, and scratch.
<code>ix_rm_mem_reserve()</code>	Reserves memory.
<code>ix_rm_mem_reserve_aligned()</code>	Reserves memory with alignment.
<code>ix_rm_mem_free()</code>	Frees memory.
<code>ix_rm_mem_info()</code>	Retrieves memory information for the specified memory type and specified channel.
<code>ix_rm_mem_local_alloc()</code>	Allocates local memory.
<code>ix_rm_mem_local_reserve()</code>	Reserves local memory
<code>ix_rm_mem_local_free()</code>	Frees local memory.
<code>ix_rm_mem_local_info()</code>	Retrieves local memory information.
<code>ix_rm_get_phys_offset()</code>	Returns the physical offset of a memory block.
<code>ix_rm_get_virtual_address()</code>	Returns the virtual address of a memory block.
Read/Write Macros	Macros to read and write memory locations. See Section 3.8.2.13 .

3.8.1 Defined Types, Enumerations, and Data Structures

This section describes data structures used in Memory Management API. See also [Section 3.2.1.7](#), “`ix_memory_reserved_area`.”

3.8.1.1 `ix_memory_type`

This data type represents available memory types in the system under the control, in whole or in part, of the Resource Manager.

C Syntax

```
typedef enum ix_e_memory_type {
    IX_MEMORY_TYPE_FIRST = 0,
    IX_MEMORY_TYPE_DRAM = IX_MEMORY_TYPE_FIRST,
    IX_MEMORY_TYPE_SRAM,
    IX_MEMORY_TYPE_SCRATCH,
    IX_MEMORY_TYPE_LOCAL,
    IX_MEMORY_TYPE_LAST
} ix_memory_type;
```


3.8.1.2 ix_memory_info

This structure is used to retrieve memory information from the Resource Manager including the type of memory and the particular channel of memory that is managed by the Resource Manager.

C Syntax

```
typedef struct ix_s_memory_info {
    ix_uint32*    m_pStartAddress;
    ix_uint32     m_TotalSize;
    ix_uint32     m_FreeSize;
    ix_uint32     m_DefaultAlignmentShift;
    ix_uint32     m_ChannelPhysicalOffset;
    void*         m_pChannelPhysicalAddress;
} ix_memory_info;
```

Data Members

m_pStartAddress	A pointer to the start of the memory block.
m_TotalSize	The total memory size.
m_FreeSize	The size of the free memory area.
m_DefaultAlignmentShift	The default alignment expressed as a power of two.
m_ChannelPhysicalOffset	The physical offset of the start of the memory block inside the memory channel.
m_pChannelPhysicalAddress	A pointer to the physical address of the start of the current memory channel.

3.8.1.3 `ix_memory_alignment_type`

This enumerated type describes the possible requests for allocating or reserving memory with alignment constraints. `IX_MEMORY_ALIGNMENT_TYPE_VIRTUAL` allocates memory providing alignment for the virtual memory. The physical offset and/or physical address might not be aligned with the same alignment requirements. `IX_MEMORY_ALIGNMENT_TYPE_PHYSICAL` allocates memory providing alignment for the physical memory. The physical offset and/or virtual address might not be aligned with the same alignment requirements.

`IX_MEMORY_ALIGNMENT_TYPE_PHYSICAL_OFFSET` allocates memory providing alignment for the physical memory offset for the channel. The virtual address and/or physical address might not be aligned with the same alignment requirements.

For large alignment the physical address, physical offset and virtual address might not satisfy the same alignment at the same time.

C Syntax

```
typedef enum ix_e_memory_alignment_type {  
    IX_MEMORY_ALIGNMENT_TYPE_FIRST = 0,  
    IX_MEMORY_ALIGNMENT_TYPE_VIRTUAL = IX_MEMORY_ALIGNMENT_TYPE_FIRST,  
    IX_MEMORY_ALIGNMENT_TYPE_PHYSICAL,  
    IX_MEMORY_ALIGNMENT_TYPE_PHYSICAL_OFFSET,  
    IX_MEMORY_ALIGNMENT_TYPE_LAST  
} ix_memory_alignment_type;
```


3.8.2 API Functions

3.8.2.1 `ix_rm_mem_alloc()`

This function allocates common microengine and Intel XScale® core memory on behalf of the calling application. On return, if the allocation fails the pointer `arg_pMemoryAddr` is set to `NULL` and if the allocation succeeds `arg_pMemoryAddr` returns a virtual memory pointer to the allocated memory block address. The returned address is aligned according to the default alignment for the type of memory. If the size does not comply with memory alignment then the actual allocated size is rounded up to insure that alignment is satisfied for further allocations.

The Resource Manager allocates memory so that:

- DRAM alignment is always at an 8-byte boundary and any request is rounded off to an 8-byte boundary
- SRAM and scratch memory is always returned aligned at a 4-byte boundary and any request is rounded off to a 4-byte boundary

C Syntax

```
ix_error ix_rm_mem_alloc(
    ix_memory_type arg_MemType,
    ix_uint32 arg_MemChannel,
    ix_uint32 arg_Size,
    void** arg_pMemoryAddr);
```

Input

<code>arg_MemType</code>	The type of memory requested.
<code>arg_MemChannel</code>	The channel from which memory is requested.
<code>arg_Size</code>	The size in bytes of the requested memory block.

Output/Returns

Return Value	Returns <code>IX_SUCCESS</code> if successful and a valid <code>ix_error</code> value otherwise.
<code>arg_pMemoryAddr</code>	The pointer to the address of the newly allocated memory. The value of this pointer is <code>NULL</code> if the allocation fails and the address of the allocated memory block if successful.

3.8.2.2 `ix_rm_mem_alloc_aligned()`

The calling application uses this function to allocate memory common to microengines and the Intel XScale® core. On return, if the allocation fails the pointer `arg_pMemoryAddr` is set to `NULL` and if the allocation succeeds the pointer is set to the allocated memory block address.

The Resource Manager provides default alignment for different types of memory, but some applications might require a higher alignment than the default one. The alignment can be requested relative to the virtual address, the physical address or the physical offset. For example hardware rings require certain physical offset alignment for the controlled memory.

The returned address is aligned according to the `arg_AlignmentShift` parameter and `arg_AlignmentType` alignment type. If `arg_AlignmentShift` parameter is less than default alignment then the return memory is aligned according to the default alignment. If the alignment type is not `IX_MEMORY_ALIGNMENT_TYPE_VIRTUAL` then the resulting virtual address might not be aligned according to the `arg_AlignmentShift`—instead, the physical offset or physical address is aligned properly. When changes in alignment are required, the actual allocated size is rounded up to accommodate the required alignment.

C Syntax

```
ix_error ix_rm_mem_alloc_aligned(
    ix_memory_type arg_MemType,
    ix_uint32 arg_MemChannel,
    ix_uint32 arg_Size,
    ix_memory_alignment_type arg_AlignmentType,
    ix_uint32 arg_AlignmentShift,
    void** arg_pMemoryAddr);
```

Input

<code>arg_MemType</code>	Represents the type of memory requested.
<code>arg_MemChannel</code>	Represents the channel from which memory is requested.
<code>arg_Size</code>	Represents the size in bytes of the block requested.
<code>arg_AlignmentType</code>	Specifies the type of alignment required as explained in the Section 3.8.1.3, “ix_memory_alignment_type.”
<code>arg_AlignmentShift</code>	This value specifies the required memory block alignment as a power of two of the alignment. If the specified alignment is less than the default alignment for the corresponding memory then the default alignment is used.

Output/Returns

Return Value	Returns <code>IX_SUCCESS</code> if successful or a valid <code>ix_error</code> for failure.
<code>arg_pMemoryAddr</code>	A pointer to the allocated memory address. The pointer <code>arg_pMemoryAddr</code> is set to <code>NULL</code> if the allocation fails, or to the allocated memory block address if the allocation succeeds.

3.8.2.3 `ix_rm_mem_reserve()`

This function reserves a region of memory of the specified type at the specified offset. If memory of that type and at that offset has already been allocated an error is returned. This call is used to reserve SRAM, DRAM, or scratch memory for Microengine C. To reduce the chances of failure, call this function before any memory allocations are made.

The Resource Manager reserves memory such that:

- DRAM alignment is always at an 8-byte boundary and any request is rounded off to a 8-byte boundary
- SRAM and scratch memory is always returned aligned at a 4-byte boundary and any request is rounded off to a 4-byte boundary

C Syntax

```
ix_error ix_rm_mem_reserve(
    ix_memory_type arg_MemType,
    ix_uint32 arg_MemChannel,
    ix_uint32 arg_Size,
    ix_uint32* arg_pByteOffset);
```

Input

<code>arg_MemType</code>	The type of memory to reserve.
<code>arg_MemChannel</code>	The memory channel.
<code>arg_Size</code>	The size in bytes of the memory block to reserve.

Input/Output

<code>arg_pByteOffset</code>	The byte offset of the memory block requested for reservation. On return this value is adjusted to accommodate the memory type default alignment.
------------------------------	---

Output/Returns

Return Value Returns `IX_SUCCESS` if successful and a valid `ix_error` value otherwise. If any portion of the memory requested for reservation has been already allocated this call fails.

3.8.2.4 `ix_rm_mem_reserve_aligned()`

This function reserves an area of memory at the specified memory channel byte offset and of specified size. The function is provided as support for the Microengine C compiler and linker. This call should be made before any memory allocation has been made in order to reduce the chance of failure.

This function reserves memory with specified alignment for SRAM, DRAM and scratch memory.

If the byte offset is not be aligned according to alignment type and `arg_AlignmentShift`. If `arg_AlignmentShift` is less than the default alignment then the default alignment is used and more data than was requested might be reserved. If the alignment type is not `IX_MEMORY_ALIGNMENT_TYPE_VIRTUAL` then the resulting virtual address might not be aligned according to the `arg_AlignmentShift`—instead, the physical offset or physical address is properly aligned. In general the byte offset and size should comply with the memory alignment, otherwise more memory is reserved to insure the right alignment. On return `*arg_pByteOffset` might be modified to reflect alignment requirements.

C Syntax

```
ix_error ix_rm_mem_reserve_aligned(
    ix_memory_type arg_MemType,
    ix_uint32 arg_MemChannel,
    ix_uint32 arg_Size,
    ix_memory_alignment_type arg_AlignmentType,
    ix_uint32 arg_AlignmentShift,
    ix_uint32* arg_pByteOffset);
```

Input

<code>arg_MemType</code>	The type of memory to be reserved.
<code>arg_MemChannel</code>	The channel from which memory is to be reserved.
<code>arg_Size</code>	The size in bytes of the memory block to be reserved.
<code>arg_AlignmentType</code>	The type of alignment required as explained at ix_memory_alignment_type type description
<code>arg_AlignmentShift</code>	The required memory block alignment as the power of two of the alignment. If the specified alignment is less than the default alignment for the corresponding memory then the default alignment is used.

Input/Output

<code>arg_pByteOffset</code>	Represents the address of byte offset of the memory block requested for reservation. On return this value is adjusted to accommodate the memory type default alignment.
------------------------------	---

Output/Returns

Return Value	Returns <code>IX_SUCCESS</code> if successful or a valid <code>ix_error</code> for failure. If any portion of the memory requested for reservation has been already allocated then the call fails.
--------------	--

3.8.2.5 `ix_rm_mem_free()`

This function frees memory of a particular type previously allocated or reserved through a call to `ix_rm_mem_alloc()`. If the memory was not previously allocated by a call to `ix_rm_mem_alloc()` the results of this call are unpredictable. Passing an address that has not been allocated or that has already been freed results in an error. Passing `NULL` results in a no op. Attempting to accessing a memory location within the block after that memory block has been freed results in unpredictable behavior.

C Syntax

```
ix_error ix_rm_mem_free(  
    void* arg_pMemory);
```

Input

<code>arg_pMemory</code>	The address of the memory block to free.
--------------------------	--

Output/Returns

Return Value	Returns <code>IX_SUCCESS</code> if successful and a valid <code>ix_error</code> value otherwise. Passing an address that has not been allocated or that has already been freed results in an error. Passing <code>NULL</code> results in a no op.
--------------	---

3.8.2.6 `ix_rm_mem_info()`

This function retrieves memory information for the specified memory type and specified channel. The structure pointed to by `arg_pMemoryInfo` is returned with the requested memory information.

C Syntax

```
ix_error ix_rm_mem_info(  
    ix_memory_type arg_MemType,  
    ix_uint32 arg_MemChannel,  
    ix_memory_info* arg_pMemoryInfo);
```

Input

<code>arg_MemType</code>	The type of memory for which we want to get the info.
<code>arg_MemChannel</code>	The channel for which to get information.

Output/Returns

Return Value	Returns <code>IX_SUCCESS</code> if successful or a valid <code>ix_error</code> for failure.
<code>arg_pMemoryInfo</code>	On return points to the structure containing memory information

3.8.2.7 `ix_rm_mem_local_alloc()`

This function is used to allocate local memory for a specific microengine. The memory allocated cannot be used on the Intel XScale® core. This call returns an offset into the local memory of the microengine. This offset is patched into the microcode by the Intel XScale® core.

The Resource Manager allocates local memory such that the offset is always returned aligned at a 4-byte boundary and any request is rounded off to a 4-byte boundary.

The pointer, `arg_pByteOffset`, is set to `(ix_uint32) (-1)` if the allocation fails or to the allocated memory block offset if the allocation succeeds. The local memory is not accessible to Intel XScale® core applications. The applications retrieve the offset and patch it into the microcode for the specific microengine. The returned offset is 4-byte aligned. If the size does not comply with memory alignment then the actual allocated size is rounded up to a 4-byte boundary.

C Syntax

```
ix_error ix_rm_mem_local_alloc(
    ix_uint32 arg_MeNumber,
    ix_uint32 arg_Size,
    ix_uint32* arg_pByteOffset);
```

Input

<code>arg_MeNumber</code>	The microengine number for which local memory should be allocated. Allowed values for the microengine number are 0x00 through 0x03 and 0x10 through 0x13 for the Intel® IXP2400 Network Processor and 0x00 through 0x07 and 0x10 through 0x17 for the Intel® IXP2800 Network Processor. The validity of the microengine number is checked only in debug mode.
<code>arg_Size</code>	The size in bytes of the requested memory block.

Output/Returns

Return Value	Returns <code>IX_SUCCESS</code> if successful and a valid <code>ix_error</code> value otherwise.
<code>arg_pByteOffset</code>	The pointer used to return the allocated memory offset in bytes. <code>arg_pByteOffset</code> is set to <code>(ix_uint32) (-1)</code> if the allocation fails or to the allocated memory block offset if the allocation succeeds.

3.8.2.8 `ix_rm_mem_local_reserve()`

This function reserves a region of local memory at the specified offset. If that memory has already been allocated then an error is returned. This call is used to reserve local memory for Microengine C. To reduce the chances of failure, call this function before any memory allocations are made.

The Resource Manager reserves memory such that local memory is always returned aligned at a 4-byte boundary and any request is rounded off to a 4-byte boundary.

C Syntax

```
ix_error ix_rm_mem_local_reserve(
    ix_uint32 arg_MeNumber,
    ix_uint32 arg_Size,
    ix_uint32* arg_pByteOffset);
```

Input

<code>arg_MeNumber</code>	The microengine number for which local memory is reserved. Allowed values for the microengine number are 0x00 through 0x03 and 0x10 through 0x13 for the Intel® IXP2400 Network Processor and 0x00 through 0x07 and 0x10 through 0x17 for the Intel® IXP2800 Network Processor. The validity of the microengine number is checked only in debug mode.
<code>arg_Size</code>	The size in bytes of the requested memory block to be reserved.

Input/Output

<code>arg_pByteOffset</code>	On input, a pointer to the byte offset of the memory block to be reserved. On return this value is adjusted to reflect the default alignment for the memory block.
------------------------------	--

Output/Returns

Return Value	Returns <code>IX_SUCCESS</code> if successful or a valid <code>ix_error</code> for failure. If any portion of the memory requested for reservation has been already allocated then the call fails.
--------------	--

3.8.2.9 `ix_rm_mem_local_free()`

This function frees memory previously allocated or reserved. Passing an address that has not been allocated or that has already been freed results in an error. Passing `NULL` results in a no op. Accessing the memory location after the block has been freed results in unpredictable behavior.

C Syntax

```
ix_error ix_rm_mem_local_free(
    ix_uint32 arg_MeNumber,
    ix_uint32 arg_MemoryOffset);
```

Input

<code>arg_vMeNumber</code>	The microengine number for which local memory should be freed. Allowed values for the microengine number are 0x00 through 0x03 and 0x10 through 0x13 for the Intel® IXP2400 Network Processor and 0x00 through 0x07 and 0x10 through 0x17 for the Intel® IXP2800 Network Processor. The validity of the microengine number is checked only in debug mode.
<code>arg_MemoryOffset</code>	The offset of the memory block to be freed.

Output/Returns

Return Value	Returns <code>IX_SUCCESS</code> if successful or a valid <code>ix_error</code> for failure. Passing an address that has not been allocated or that has already been freed results in an error. Passing <code>NULL</code> results in a no op.
--------------	--

3.8.2.10 `ix_rm_mem_local_info()`

This function retrieves memory information for the specified memory type and specified channel. On return the structure pointed to by `arg_pMemoryInfo` contains the requested memory information. This function retrieves local memory information for each microengine.

C Syntax

```
ix_error ix_rm_mem_local_info(
    ix_memory_type arg_MeNumber,
    ix_memory_info* arg_pMemoryInfo);
```

Input

<code>arg_MeNumber</code>	Represents the microengine number for which to get memory information. Allowed values for the microengine number are 0x00 through 0x03 and 0x10 through 0x13 for the Intel® IXP2400 Network Processor and 0x00 through 0x07 and 0x10 through 0x17 for the Intel® IXP2800 Network Processor. The validity of the microengine number is checked only in debug mode.
---------------------------	---

Output/Returns

Return Value	Returns <code>IX_SUCCESS</code> if successful or a valid <code>ix_error</code> for failure.
<code>arg_pMemoryInfo</code>	On return points to the memory information data structure.

3.8.2.11 `ix_rm_get_phys_offset()`

Returns the physical offset in bytes for memory allocated by the Resource Manager.

This routine takes in as input a virtual memory pointer used to access the memory on the Intel XScale® core. It returns the physical offset into the channel in bytes, the physical address, the memory type, and the channel for this memory area.

If any of the *Out* parameters are `NULL` then the corresponding piece of data is not returned.

C Syntax

```
ix_error ix_rm_get_phys_offset(
    const void* arg_pMemory,
    ix_memory_type* arg_pMemType,
    ix_uint32* arg_pMemChannel,
    ix_uint32* arg_pOffset,
    void** arg_pPhysicalAddress);
```


Input

`arg_pMemory` The pointer to the memory block to query.

Output/Returns

Return Value Returns `IX_SUCCESS` if successful and a valid `ix_error` value otherwise.

`arg_pMemType` A pointer to the type of memory at the location in question.

`arg_pMemChannel` A pointer to the memory channel for the location in question.

`arg_pOffset` A pointer to the offset for the memory location in question.

`arg_pPhysicalAddress` The pointer to the physical address.

3.8.2.12 `ix_rm_get_virtual_address()`

Returns the virtual pointer into memory given a physical offset into the memory buffer, specified in bytes.

The virtual pointer returned can be used to access this memory on the Intel XScale® core.

C Syntax

```
ix_error ix_rm_get_virtual_address(
    ix_memory_type arg_MemType,
    ix_uint32 arg_MemChannel,
    ix_uint32 arg_Offset,
    ix_uint8** arg_pMemoryAddr);
```

Input

`arg_MemType` The type for the physical memory location.

`arg_MemChannel` The channel for the physical memory location.

`arg_Offset` The physical offset in bytes for the physical memory location.

Output/Returns

Return Value Returns `IX_SUCCESS` if successful and a valid `ix_error` value otherwise.

`arg_pMemoryAddr` A virtual pointer to the given physical address.

3.8.2.13 Read/Write Macros

To provide portability between code running on hardware and code running under a foreign model, the following macros are provided. Use of these macros makes it easier to write code that runs both under the foreign model and real hardware. In general for access to memory that is used by both the Intel XScale® core and the microengines, these macros should be used instead of direct dereference of the addresses. These macros support code running on microengines in big-endian mode and on the Intel XScale® core in little-endian mode without any change to the code. Using these macros facilitates writing code that runs both under the foreign model and real hardware.

Table 3-15 lists the macros provided.

Table 3-15. Resource Manager Memory Management Macros

Macro Name	Description
<code>IX_RM_MEM_UINT8_READ</code>	Returns an <code>ix_uint8</code> representing the value at the specified location.
<code>IX_RM_MEM_UINT16_READ</code>	Returns an <code>ix_uint16</code> representing the value at the specified location.
<code>IX_RM_MEM_UINT32_READ</code>	Returns an <code>ix_uint32</code> representing the value at the specified location.
<code>IX_RM_MEM_UINT64_READ</code>	Returns an <code>ix_uint64</code> representing the value at the specified location.
<code>IX_RM_MEM_UINT8_WRITE</code>	Writes an <code>ix_uint8</code> value to the specified location.
<code>IX_RM_MEM_UINT16_WRITE</code>	Writes an <code>ix_uint16</code> value to the specified location.
<code>IX_RM_MEM_UINT32_WRITE</code>	Writes an <code>ix_uint32</code> value to the specified location.
<code>IX_RM_MEM_UINT64_WRITE</code>	Writes an <code>ix_uint64</code> value to the specified location.

3.8.2.13.1 `IX_RM_MEM_UINT8_READ`

A macro used to read an 8-bit value from memory.

C Syntax

```
#define IX_RM_MEM_UINT8_READ(arg_pMemoryAddr)
```

Input

`arg_pMemoryAddr` The pointer to the address of the memory location to be read. The value at this location should be of the type `ix_uint8*`.

Output/Returns

Return Value An `ix_uint8` representing the value at the specified location.

3.8.2.13.2 `IX_RM_MEM_UINT16_READ`

A macro used to read a 16-bit value from memory.

C Syntax

```
#define IX_RM_MEM_UINT16_READ(arg_pMemoryAddr)
```

Input

<code>arg_pMemoryAddr</code>	A pointer to the address of the memory location to be read. The value at this location should be of the type <code>ix_uint16*</code> and aligned on a 16-bit boundary.
------------------------------	--

Output/Returns

Return Value	An <code>ix_uint16</code> representing the value at the specified location.
--------------	---

3.8.2.13.3 `IX_RM_MEM_UINT32_READ`

A macro used to read a 32-bit value from memory.

C Syntax

```
#define IX_RM_MEM_UINT32_READ(arg_pMemoryAddr)
```

Input

<code>arg_pMemoryAddr</code>	A pointer to the address of the memory location to be read. The value at this location should be of the type <code>ix_uint32*</code> and aligned on a 32-bit boundary.
------------------------------	--

Output/Returns

Return Value	An <code>ix_uint32</code> representing the value at the specified location.
--------------	---

3.8.2.13.4 `IX_RM_MEM_UINT64_READ`

A macro used to read a 64-bit value from memory.

C Syntax

```
#define IX_RM_MEM_UINT64_READ(arg_pMemoryAddr)
```

Input

<code>arg_pMemoryAddr</code>	The pointer to the address of the memory location to be read. The value at this location should be of the type <code>ix_uint64*</code> and aligned on a 64-bit boundary.
------------------------------	--

Output/Returns

Return Value	An <code>ix_uint64</code> representing the value at the specified location.
--------------	---

3.8.2.13.5 `IX_RM_MEM_UINT8_WRITE`

Writes an 8-bit value to memory.

C Syntax

```
#define IX_RM_MEM_UINT8_WRITE(arg_pMemoryAddr, arg_Value)
```

Input

<code>arg_pMemoryAddr</code>	The address of the memory location to be written. The value at this location should be of the type <code>ix_uint8*</code> . The address need not be aligned.
<code>arg_vValue</code>	The value to be written to the specified location. The value should be of the type <code>ix_uint8</code> .

Output/Returns

Return Value	An <code>ix_uint8</code> representing the value written at the specified location.
--------------	--

3.8.2.13.6 `IX_RM_MEM_UINT16_WRITE`

Writes a 16-bit value to memory.

C Syntax

```
#define IX_RM_MEM_UINT16_WRITE(arg_pMemoryAddr, arg_Value)
```

Input

<code>arg_pMemoryAddr</code>	The address of the memory location to be written. The value at this location should be of the type <code>ix_uint16*</code> . The address should be 16-bit aligned.
<code>arg_vValue</code>	The value to be written to the specified location. The value should be of the type <code>ix_uint16</code> .

Output/Returns

Return Value	An <code>ix_uint16</code> representing the value written at the specified location.
--------------	---

3.8.2.13.7 `IX_RM_MEM_UINT32_WRITE`

Writes a 32-bit value to memory.

C Syntax

```
#define IX_RM_MEM_UINT32_WRITE(arg_pMemoryAddr, arg_Value)
```

Input

<code>arg_pMemoryAddr</code>	The address of the memory location to be written. The value at this location should be of the type <code>ix_uint32*</code> . The address should be 32-bit aligned.
<code>arg_vValue</code>	The value to be written to the specified location. The value should be of the type <code>ix_uint32</code> .

Output/Returns

Return Value	An <code>ix_uint32</code> representing the value written at the specified location.
--------------	---

3.8.2.13.8 `IX_RM_MEM_UINT64_WRITE`

Writes a 64-bit value to memory.

C Syntax

```
#define IX_RM_MEM_UINT64_WRITE(arg_pMemoryAddr, arg_Value)
```

Input

<code>arg_pMemoryAddr</code>	The address of the memory location to be written. The value at this location should be of the type <code>ix_uint64*</code> . The address should be 64-bit aligned.
<code>arg_vValue</code>	The value to be written to the specified location. The value should be of the type <code>ix_uint64</code> .

Output/Returns

Return Value	An <code>ix_uint64</code> representing the value written at the specified location.
--------------	---

3.9 System Repository API

The system repository is designed as a collection of tree constructs that store system properties. The structure of these tree constructs is similar to a file system structure, and the navigation is also similar.

Properties represent name-value entities that can be set and accessed throughout the system in a consistent manner. Configuration properties are defined by property handles that link together the name and value pair. There is a limit on the number of properties that can be created in the system. Each property has a set of attributes associated with it. There could be properties that can act just as nodes in the hierarchy and that have no value associated with them. Once a property has been created the calling application can register with the property to receive notifications if the property changes. Every application in the system can create properties at a certain node level, and retrieve and modify them.

For now we have identified just one property tree—the core software property tree—that stores all the properties of the applications residing on the core side. The configuration property handle corresponding to the root of this tree is `IX_CP_CORE_PROPERTY_ROOT`.

The system repository API is shown in [Table 3-16](#).

Table 3-16. Resource Manager System Repository API

Name	Description
<code>ix_configuration_property_handle</code>	Generic handle type for configuration properties.
<code>ix_cp_property_info</code>	Configuration property information structure.
<code>ix_rm_cp_property_create()</code>	Creates a new configuration property at a certain node level.
<code>ix_rm_cp_property_delete()</code>	Deletes a configuration property.
<code>ix_rm_cp_property_open()</code>	Retrieves a configuration property based on a base node and a name.
<code>ix_rm_cp_property_close()</code>	Invalidates a configuration property handle.
<code>ix_rm_cp_property_attach()</code>	A communication ID is registered with the property to receive change notifications.
<code>ix_rm_cp_property_detach()</code>	A communication ID is unregistered from the property notification list.
<code>ix_rm_cp_property_set_value()</code>	A value is associated with a configuration property or the previous value is replaced with the new one.
<code>ix_rm_cp_property_get_value()</code>	Retrieves a property value.
<code>ix_rm_cp_property_set_value_uint32()</code>	A 32-bit unsigned value is associated with a configuration property or the previous value is replaced with the new one.
<code>ix_rm_cp_property_get_value_uint32()</code>	Retrieves a 32-bit unsigned property value.
<code>ix_rm_cp_property_delete_value()</code>	Deletes the value associated with a property.
<code>ix_rm_cp_property_get_info()</code>	Gets information pertaining to a configuration property.
<code>ix_rm_cp_property_get_subproperty()</code>	Navigates a subtree of a configuration property.

3.9.1 Defined Types, Enumerations, and Data Structures

3.9.1.1 `ix_configuration_property_handle`

This type represents a generic configuration property handle. A handle is used to access any configuration property in the system. Two handles are reserved as the roots of the two configuration property trees in the system: `IX_CP_CORE_PROPERTY_ROOT`.

C Syntax

```
typedef ix_handle ix_configuration_property_handle;
```

`IX_NULL_CONFIGURATION_PROPERTY_HANDLE`

Defines a NULL configuration property handle.

C Syntax

```
#define IX_NULL_CONFIGURATION_PROPERTY_HANDLE  
((ix_configuration_property_handle)0)
```

`IX_CP_MAX_PROPERTY_NAME_LENGTH`

This symbol defines the maximum length for a property name.

C Syntax

```
#define IX_CP_MAX_PROPERTY_NAME_LENGTH 32UL
```

3.9.1.2 `ix_cp_property_info`

This structure represents property attributes information—and is used in functions in this API to retrieve property information. Each property has a link to its parent and links to the previous and next properties in a sibling list. It contains a subproperty list. The name is the name relative to the parent property.

C Syntax

```
typedef struct ix_s_cp_property_info {  
    ix_uint32                m_Options;  
    ix_uint32                m_DataSize;  
    ix_uint32                m_DataType;  
    ix_configuration_property_handle m_hParentProperty;  
    ix_configuration_property_handle m_hPropertyNext;  
    ix_configuration_property_handle m_hPropertyPrev;  
    ix_configuration_property_handle m_hChildrenPropertiesList;  
    char                    m_aName[IX_CP_MAX_PROPERTY_NAME_LENGTH];  
} ix_cp_property_info;
```


3.9.2 API Functions

3.9.2.1 `ix_rm_cp_property_create()`

This function creates a new configuration property relative to a parent property. At the parent level the property name should be unique. When a new property is created it has an internal reference count of one.

C Syntax

```
ix_error ix_rm_cp_property_create(
    ix_configuration_property_handle arg_hParentProperty,
    const char* arg_pPropertyName,
    ix_uint32 arg_PropertyOptions,
    ix_configuration_property_handle* arg_pProperty);
```

Input

<code>arg_hParentProperty</code>	Represents the property that is to contain the newly created property. If this value is <code>IX_CP_CORE_PROPERTY_ROOT</code> the new property is created at the top level of the registry.
<code>arg_pPropertyName</code>	The name of the new property to be created at this level. The name must be unique at the parent property level or an error is returned.
<code>arg_PropertyOptions</code>	The options to be applied to the newly created property.

Output/Returns

Return Value	Returns <code>IX_SUCCESS</code> if successful or a valid <code>ix_error</code> token for failure. If the parent property is invalid then an error is returned. If a property with the same name exists at the specified location, an error is returned.
<code>arg_pProperty</code>	Represents the location where the handle of the newly created property is stored. In the case of failure this value is set to <code>IX_NULL_CONFIGURATION_PROPERTY_HANDLE</code> upon return.

3.9.2.2 ix_rm_cp_property_delete()

This function deletes the specified property from the registry. If the property has a notification list, all registered communication IDs are notified about the deletion. After deletion the property is no longer available in the system. The property is deleted regardless of the internal reference count value for the property.

C Syntax

```
ix_error ix_rm_cp_property_delete(  
    ix_configuration_property_handle arg_hProperty);
```

Input

arg_hProperty	The property to delete from the registry.
---------------	---

Output/Returns

Return Value	Returns IX_SUCCESS if successful or a valid ix_error token for failure.
--------------	---

3.9.2.3 `ix_rm_cp_property_open()`

This function retrieves the handle of the property named `arg_pPropertyName` at the level of `arg_hParentProperty`. After a successful call to this function the returned handle is a valid property handle that can be used for further work.

Names can be specified as a path as in:

`level1_name/level2_name/name`

The / separator is used between levels in the path. In the example above, the property is three levels down in the subtree relative to the parent property.

A call to `ix_rm_cp_property_open()` results in an increment of the internal reference count for the property.

C Syntax

```
ix_error ix_rm_cp_property_open(
    ix_configuration_property_handle arg_hParentProperty,
    const char* arg_pPropertyName,
    ix_configuration_property_handle* arg_pProperty);
```

Input

<code>arg_hParentProperty</code>	The name of the property specifying the level to search for the property named <code>arg_pPropertyName</code> .
<code>arg_pPropertyName</code>	The name of the property to search for. The name can be a full path name in which case the search in the tree structure takes place relative to the <code>arg_hParentProperty</code> level.

Output/Returns

Return Value	Returns <code>IX_SUCCESS</code> if successful or a valid <code>ix_error</code> token for failure.
<code>arg_pProperty</code>	Represents the location where the required property handle is stored. If the property does not exist at the specified level this value is set to <code>IX_NULL_CONFIGURATION_PROPERTY_HANDLE</code> upon return.

3.9.2.4 `ix_rm_cp_property_close()`

This function invalidates the property handle. From this point on the use of this handle in any operation returns an error—this handle can not be used to access a property. A call to `ix_rm_cp_property_close()` decrements the internal reference count for the property. Once the reference count reaches zero calls to `ix_rm_cp_property_close()` have the same effect as calls to `ix_rm_cp_property_delete()`—the property is deleted. That is why it is important to match each call to `ix_rm_cp_property_open()` with one *and only one* call to `ix_rm_cp_property_close()`.

C Syntax

```
ix_error ix_rm_cp_property_close(  
    ix_configuration_property_handle arg_hProperty);
```

Input

<code>arg_hProperty</code>	The handle of the property to delete.
----------------------------	---------------------------------------

Output/Returns

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

3.9.2.5 ix_rm_cp_property_attach()

This function registers a communication ID with a property—specified by the handle, `arg_hProperty`—in order to receive message notifications when the property changes. When this property changes, a message is sent to the registered communication ID with the handle of the property that has changed. When the message is processed is dependant on the communication ID receive mode. A message is sent before the property is deleted.

C Syntax

```
ix_error ix_rm_cp_property_attach(  
    ix_configuration_property_handle arg_hProperty,  
    ix_communication_id arg_hNotificationCommId);
```

Input

<code>arg_hProperty</code>	The handle of the property whose changes we are interested in.
<code>arg_hNotificationCommId</code>	The communication ID requesting notification events.

Output/Returns

Return Value	Returns <code>IX_SUCCESS</code> if successful or a valid <code>ix_error</code> token for failure. If the communication ID is already registered to receive notification messages then an error is returned.
--------------	---

3.9.2.6 `ix_rm_cp_property_detach()`

This function unregisters the communication ID from the notification list of a specified configuration property. From this point on, notification messages are no longer sent to the detached communication ID from the specified configuration property.

C Syntax

```
ix_error ix_rm_cp_property_detach(  
    ix_configuration_property_handle arg_hProperty,  
    ix_communication_id arg_hNotificationCommId);
```

Input

<code>arg_hProperty</code>	The handle of the property of interest.
<code>arg_hNotificationCommId</code>	The communication ID that we want to unregister from the property notification list.

Output/Returns

Return Value	Returns <code>IX_SUCCESS</code> if successful or a valid <code>ix_error</code> token for failure. If the given communication ID is not registered for notification with the specified property, then an error is returned.
--------------	--

3.9.2.7 `ix_rm_cp_property_set_value()`

This function associates a value with a property or replaces an existing value.

When a property is created no value is associated with that property. It acts just as a node in the System Repository structure.

There are two types of value that can be associated with a property: a general opaque value that should have meaning just for entities interested in that property, and a unsigned 32-bit value that can be used by every entity as long it understands its meaning. Once a type of value is in place, trying to retrieve it as the other type results in an error.

C Syntax

```
ix_error ix_rm_cp_property_set_value(  
    ix_configuration_property_handle arg_hProperty,  
    ix_uint32 arg_DataSize,  
    const void* arg_pData);
```

Input

<code>arg_hProperty</code>	The handle of the property whose value we want to set.
<code>arg_DataSize</code>	The size of the data value—in bytes.
<code>arg_pData</code>	A pointer to the property value to set.

Output/Returns

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

3.9.2.8 `ix_rm_cp_property_get_value()`

This function retrieves the generic opaque value associated with this property. If the property has no value associated with it or if the property is an unsigned 32-bit property an error is returned.

C Syntax

```
ix_error ix_rm_cp_property_get_value(  
    ix_configuration_property_handle arg_hProperty,  
    ix_uint32 arg_DataSize,  
    void* arg_pData);
```

Input

<code>arg_hProperty</code>	The handle specifying the property whose value is to be returned.
<code>arg_DataSize</code>	The size of the data to be copied from the value to the location specified by <code>arg_pData</code> .

Output/Returns

Return Value	Returns <code>IX_SUCCESS</code> if successful or a valid <code>ix_error</code> token for failure.
<code>arg_pData</code>	A pointer to the opaque property value.

3.9.2.9 ix_rm_cp_property_set_value_uint32()

This function associates an unsigned 32-bit value with a property or replaces the existing value.

Note: This function is the only one used for the microcode system repository tree.

C Syntax

```
ix_error ix_rm_cp_property_set_value_uint32(  
    ix_configuration_property_handle arg_hProperty,  
    ix_uint32 arg_Value);
```

Input

arg_hProperty	A handle specifying a property.
arg_Value	The new value for the specified property.

Output/Returns

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

3.9.2.10 `ix_rm_cp_property_get_value_uint32()`

This function retrieves the unsigned 32-bit value associated with a property. If no value exists or the value is not the appropriate type, then an error is returned.

C Syntax

```
ix_error ix_rm_cp_property_get_value_uint32(  
    ix_configuration_property_handle arg_hProperty,  
    ix_uint32* arg_pValue);
```

Input

<code>arg_hProperty</code>	A handle of the property whose value is to be retrieved.
----------------------------	--

Output/Returns

Return Value	Returns <code>IX_SUCCESS</code> if successful or a valid <code>ix_error</code> token for failure.
<code>arg_pValue</code>	A pointer to the location for the returned <code>uint32</code> property value.

3.9.2.11 `ix_rm_cp_property_delete_value()`

This function deletes the value of the specified property. From this point on, this property has no value associated with it—trying to retrieve the value for this property results in an error.

C Syntax

```
ix_error ix_rm_cp_property_delete_value(
    ix_configuration_property_handle arg_hProperty);
```

Input

<code>arg_hProperty</code>	The handle of the property whose value is to be deleted.
----------------------------	--

Output/Returns

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

3.9.2.12 `ix_rm_cp_property_get_info()`

This function returns the information associated with the property. This function can be used for navigation of the property subtree or for enumerating all subproperties of a specified property.

C Syntax

```
ix_error ix_rm_cp_property_get_info(
    ix_configuration_property_handle arg_hProperty,
    ix_cp_property_info* arg_pInfo);
```

Input

<code>arg_hProperty</code>	A handle to the property whose attributes are to be retrieved.
----------------------------	--

Output/Returns

Return Value	Returns <code>IX_SUCCESS</code> if successful or a valid <code>ix_error</code> token for failure.
<code>arg_pInfo</code>	A pointer to the property information.

3.9.2.13 ix_rm_cp_property_get_subproperty()

This function returns the handle of the property with the `arg_SubpropertyIndex` index. The subproperties indexes start from zero. This function can be used for property subtree navigation or for enumerating all subproperties of a certain property.

C Syntax

```
ix_error ix_rm_cp_property_get_subproperty(
    ix_configuration_property_handle arg_hProperty,
    ix_uint32 arg_SubpropertyIndex,
    ix_configuration_property_handle* arg_pSubproperty);
```

Input

<code>arg_hProperty</code>	A handle specifying the property whose subproperty is to be returned.
<code>arg_SubpropertyIndex</code>	The index of the subproperty to retrieve.

Output/Returns

Return Value	Returns <code>IX_SUCCESS</code> if successful or a valid <code>ix_error</code> token for failure.
<code>arg_pSubproperty</code>	A pointer to the retrieved subproperty handle.

3.10 64-Bit Counters API

The 64-bit counter API is provided as statistics support for the Intel XScale® core applications. *RFC2863* states that 64-bit counters must be supported for interfaces that operate at data rates greater than 650Mbps.

The design of these counters is such that for each 64-bit counter there is a corresponding 32-bit counter residing in SRAM or SCRATCH that can be updated atomically by the microengines. Each counter has an associated overflow time for the internal 32-bit counter. The API has a background thread that monitors all the overflow times for the internal 32-bit counters and do an atomic read/set to 0 operation (atomic swap) for the internal 32-bit counters, and then update the 64-bit counterparts accordingly. The memory necessary for the internal counters is allocated by the callers as they need to know what values need to be patched into the microcode.

Table 3-17 shows the relevant data structures and functions.

Table 3-17. Resource Manager 64-Bit Counter API

Name	Description
<code>ix_counter_64bit_handle()</code>	Generic type for a 64-bit counter handle.
<code>ix_rm_counter_64bit_new()</code>	Allocates an array of 64-bit counters.
<code>ix_rm_counter_64bit_delete()</code>	Deletes a 64-bit counter.
<code>ix_rm_counter_64bit_get_internal_overflow_time()</code>	Retrieves the overflow time for the internal 32-bit counter of the specified 64-bit counter.
<code>ix_rm_counter_64bit_set_internal_overflow_time()</code>	Sets the overflow time for the internal 32-bit counter of the specified 64-bit counter.
<code>ix_rm_counter_64bit_get_value()</code>	Returns the 64-bit core value of the counter.
<code>ix_rm_counter_64bit_set_value()</code>	Sets the 64-bit core value of the counter.

3.10.1 Defined Types, Enumerations, and Data Structures

3.10.1.1 `ix_counter_64bit_handle()`

This is a generic type for a 64-bit counter handle.

C Syntax

```
typedef ix_handle ix_counter_64bit_handle;
```

NULL Counter Handle

Defines a null 64-bit counter handle.

C Syntax

```
#define IX_NULL_COUNTER_64BIT_HANDLE ((ix_counter_64bit_handle)0)
```

3.10.2 API Functions

3.10.2.1 `ix_rm_counter_64bit_new()`

This function allocates an array of new 64-bit counters returning an array of counter handles with the handles of the newly created counters. The calling application specifies the number of counters to create.

C Syntax

```
ix_error ix_rm_counter_64bit_new(
    ix_uint32 arg_CounterNumber,
    ix_uint32* arg_InternalCounterMemoryBlock,
    ix_uint32 arg_InternalOverflowTime,
    ix_counter_64bit_handle* arg_pCounterHandle);
```


Input

<code>arg_CounterNumber</code>	The number of counters to create.
<code>arg_InternalCounterMemoryBlock</code>	<p>The virtual address of the memory block to use for the internal 32-bit counters corresponding to each 64-bit counter. The block of memory should be an array of 32-bit integers with at least <code>arg_CounterNumber</code> elements. The first element of this array is used as the internal 32-bit counter of the first 64-bit counter created, and so on. The calling application can simply patch into the microcode the offset of the start of this block.</p> <p>This memory block should be common memory allocated by the calling application through a call to the <code>ix_rm_mem_alloc()</code> function.</p>
<code>arg_InternalOverflowTime</code>	<p>The overflow time for the internal 32-bit microengine counter, expressed in milliseconds.</p> <p>The internal 32-bit timer is guaranteed to be read and reset more often than this specified overflow time—in this way the internal timer is guaranteed not to overflow.</p>

Output/Returns

Return Value	Returns <code>IX_SUCCESS</code> if successful or a valid <code>ix_error</code> token for failure.
<code>arg_pCounterHandle</code>	A pointer to the first element of a 64-bit counter handle array used to return the handles of the newly created counters. The calling application must allocate this handle array before calling this function.

3.10.2.2 `ix_rm_counter_64bit_delete()`

This function deletes the specified 64-bit counter. The memory corresponding to the internal 32-bit counter is not freed. From this point on the handle is invalid and should not be used. All resources related to the counter are released.

C Syntax

```
ix_error ix_rm_counter_64bit_delete(
    ix_counter_64bit_handle arg_hCounter);
```

Input

<code>arg_hCounter</code>	The handle of the counter to be deleted.
---------------------------	--

Output/Returns

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

3.10.2.3 `ix_rm_counter_64bit_get_internal_overflow_time()`

This function returns the overflow time for the internal 32-bit counter specified by a counter handle. The overflow time is expressed in milliseconds.

C Syntax

```
ix_error ix_rm_counter_64bit_get_internal_overflow_time(
    ix_counter_64bit_handle arg_hCounter,
    ix_uint32* arg_pInternalOverflowTime);
```

Input

<code>arg_hCounter</code>	A handle to the counter of interest.
<code>arg_pInternalOverflowTime</code>	A pointer used to return the current internal counter overflow time.

Output/Returns

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

3.10.2.4 ix_rm_counter_64bit_set_internal_overflow_time()

This function sets a new overflow time for the specified internal 32-bit microengine counter. The overflow time is expressed in milliseconds.

Note: This operation might result in a read of all currently registered counters.

C Syntax

```
ix_error ix_rm_counter_64bit_set_internal_overflow_time(
    ix_counter_64bit_handle arg_hCounter,
    ix_uint32 arg_InternalOverflowTime);
```

Input

arg_hCounter	A handle to the counter of interest.
arg_InternalOverflowTime	The new overflow time for the internal 32-bit microengine counter.

Output/Returns

Return Value	Returns IX_SUCCESS if successful or a valid ix_error token for failure.
--------------	---

3.10.2.5 ix_rm_counter_64bit_get_value()

This function returns the 64-bit core value of the counter. Internally the function reads the microengine counter and resets it to zero then increments the core counter with the microengine counter value.

C Syntax

```
ix_error ix_rm_counter_64bit_get_value(
    ix_counter_64bit_handle arg_hCounter,
    ix_uint64* arg_pCounterValue);
```

Input

arg_hCounter	A handle to the counter of interest.
--------------	--------------------------------------

Output/Returns

Return Value	Returns <code>IX_SUCCESS</code> if successful or a valid <code>ix_error</code> token for failure.
<code>arg_pCounterValue</code>	A pointer to use to return the value of the counter.

3.10.2.6 `ix_rm_counter_64bit_set_value()`

This function sets the value of the core counter to the specified value.

C Syntax

```
ix_error ix_rm_counter_64bit_set_value(  
    ix_counter_64bit_handle arg_hCounter,  
    ix_uint64 arg_CounterValue);
```

Input

<code>arg_hCounter</code>	A handle to the counter of interest.
<code>arg_CounterValue</code>	The new value for the core counter.

Output/Returns

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

3.11 Services API

The Services API provides a set of functions that allows core applications to take advantage of certain hardware features. The API provides a set of atomic operations and fast memory operations. [Table 3-18](#) lists data structures and functions included in this API.

Table 3-18. Resource Manager Services API

Functions	Description
<code>ix_rm_atomic_sram_swap()</code>	Performs an atomic swap between an SRAM memory location and one arbitrary memory location.
<code>ix_rm_atomic_sram_add()</code>	Performs an atomic add to an SRAM memory location.
<code>ix_rm_atomic_sram_test_and_add()</code>	Performs an atomic add to an SRAM memory location and returns the value stored at the memory location before add operation.
<code>ix_rm_atomic_sram_subtract()</code>	Performs an atomic subtract to an SRAM memory location.
<code>ix_rm_atomic_sram_test_and_subtract()</code>	Performs an atomic subtract to an SRAM memory location and returns the value stored at the memory location before the subtract operation.
<code>ix_rm_atomic_sram_bit_set()</code>	Performs an atomic bit set operation to an SRAM memory location.
<code>ix_rm_atomic_sram_bit_test_and_set()</code>	Performs an atomic bit set operation to an SRAM memory location and returns the value stored at the memory location before the bit set operation.
<code>ix_rm_atomic_sram_bit_clear()</code>	Performs an atomic bit clear operation to an SRAM memory location.
<code>ix_rm_atomic_sram_bit_test_and_clear()</code>	Performs an atomic bit clear operation to an SRAM memory location and returns the value stored at the memory location before the bit clear operation.
<code>ix_rm_atomic_scratch_swap()</code>	Performs an atomic swap between a SCRATCH memory location and one arbitrary memory location.
<code>ix_rm_atomic_scratch_add()</code>	Performs an atomic add to a SCRATCH memory location.
<code>ix_rm_atomic_scratch_test_and_add()</code>	Performs an atomic add to a SCRATCH memory location and returns the value stored at the memory location before the add operation.
<code>ix_rm_atomic_scratch_subtract()</code>	Performs an atomic subtract to a SCRATCH memory location.
<code>ix_rm_atomic_scratch_test_and_subtract()</code>	Performs an atomic subtract to a SCRATCH memory location and returns the value stored at the memory location before the subtract operation.
<code>ix_rm_atomic_scratch_bit_set()</code>	Performs an atomic bit set operation to a SCRATCH memory location.
<code>ix_rm_atomic_scratch_bit_test_and_set()</code>	Performs an atomic bit set operation to a SCRATCH memory location and returns the value stored at the memory location before the bit set operation.
<code>ix_rm_atomic_scratch_bit_clear()</code>	Performs an atomic bit clear operation to a SCRATCH memory location.
<code>ix_rm_atomic_scratch_bit_test_and_clear()</code>	Performs an atomic bit clear operation to a SCRATCH memory location and returns the value stored at the memory location before the bit clear operation.

Table 3-18. Resource Manager Services API (Continued)

<code>ix_rm_managed_to_os_memory_copy()</code>	Performs a fast memory copy from a managed memory location to an OS memory location.
<code>ix_rm_os_to_managed_memory_copy()</code>	Performs a fast memory copy from an OS memory location to a managed memory location.
<code>ix_rm_managed_to_managed_memory_copy()</code>	Performs a fast memory copy from a managed memory location to another managed memory location.

3.11.1 API Functions

3.11.1.1 `ix_rm_atomic_sram_swap()`

This function will perform an SRAM memory atomic swap. The value referred by `arg_pValue1` will be swapped with the value referred by `arg_pValue2`. On return `*arg_pValue1` will be changed to the previous `arg_pValue2` and vice-versa. The operation is atomic relative to the second argument of the swap operation, as long as `*arg_pValue2` points to an SRAM memory location.

C Syntax

```
void ix_rm_atomic_sram_swap(
    ix_uint32* arg_pValue1,
    ix_uint32* arg_pValue2
);
```

Input/Output

<code>arg_pValue1</code>	The address of the first operand of the swap operation in SRAM memory.
<code>arg_pValue2</code>	The address of the second operand of the swap operation. Must be stored in SRAM memory for the operation to be atomic.

3.11.1.2 `ix_rm_atomic_sram_add()`

This function increments the `arg_pSramValue` address contents by `arg_IncrementValue`. The `arg_pSramValue` address should be an SRAM address, as it will not be checked. The value at `arg_pSramValue` saturates at 0. If you add a negative number that is larger in absolute value than the value at `arg_pSramValue`, then the result at the `arg_pSramValue` location will be 0.

C Syntax

```
void ix_rm_atomic_sram_add(
    ix_int32 arg_IncrementValue,
    ix_uint32* arg_pSramValue
);
```


Input

`arg_IncrementValue` Value to be added to the memory location addressed by `arg_pSramValue`.

Input/Output

`arg_pSramValue` On input, location of the value to be incremented by `arg_IncrementValue`.
Upon return, the incremented value.

3.11.1.3 ix_rm_atomic_sram_test_and_add()

This function increments the `arg_pSramValue` address contents by `arg_IncrementValue`. The function also returns the value at `arg_pSramValue` location before the increment operation completes. The `arg_pSramValue` address should be an SRAM address, as it will not be checked. The value at `arg_pSramValue` saturates at 0. If you add a negative number that is larger in absolute value than the value at `arg_pSramValue`, then the result at the `arg_pSramValue` location will be 0.

C Syntax

```
ix_uint32 ix_rm_atomic_sram_test_and_add(
    ix_int32 arg_IncrementValue,
    ix_uint32* arg_pSramValue
);
```

Input

`arg_IncrementValue` Value to be added to the memory location addressed by `arg_pSramValue`.

Input/Output

`arg_pSramValue` On input, location of the value to be incremented by `arg_IncrementValue`.
Upon return, the incremented value.

Return

Return Value Returns the `arg_pSramValue` value before the atomic add has been performed.

3.11.1.4 ix_rm_atomic_sram_subtract()

This function decrements the `arg_pSramValue` address contents by `arg_DecrementValue`. The `arg_pSramValue` address should be an SRAM address, as it will not be checked. The value at `arg_pSramValue` saturates at 0. If you subtract a positive number that is larger in absolute value than the value at `arg_pSramValue`, then the result at the `arg_pSramValue` location will be 0.

C Syntax

```
void ix_rm_atomic_sram_subtract(
    ix_int32 arg_DecrementValue,
    ix_uint32* arg_pSramValue
);
```

Input

<code>arg_DecrementValue</code>	Value to be subtracted to the memory location addressed by <code>arg_pSramValue</code> .
---------------------------------	--

Input/Output

<code>arg_pSramValue</code>	On input, value at this location will be decremented by <code>arg_DecrementValue</code> . Upon return, the decremented value.
-----------------------------	--

3.11.1.5 ix_rm_atomic_sram_test_and_subtract()

This function decrements the `arg_pSramValue` address contents by `arg_DecrementValue`. The function also returns the value at `arg_pSramValue` location before the decrement operation completes. The `arg_pSramValue` address should be an SRAM address, as it will not be checked. The value at `arg_pSramValue` saturates at 0. If you subtract a positive number that is larger in absolute value than the value at `arg_pSramValue`, then the result at the `arg_pSramValue` location will be 0.

C Syntax

```
ix_uint32 ix_rm_atomic_sram_test_and_subtract(
    ix_int32 arg_DecrementValue,
    ix_uint32* arg_pSramValue
);
```

Input

<code>arg_DecrementValue</code>	Value to be subtracted to the memory location addressed by <code>arg_pSramValue</code> .
---------------------------------	--

Input/Output

`arg_pSramValue` On input, value at this location will be decremented by `arg_DecrementValue`.
Upon return, the decremented value.

Return

Return Value Returns the `arg_pSramValue` value before the atomic subtract was performed.

3.11.1.6 ix_rm_atomic_sram_bit_set()

This function atomically sets the bits of the `arg_pSramValue` address contents corresponding to the bits set to 1 in the `arg_SetValue` argument. In order for the operation to complete successfully, the `arg_pSramValue` address must be an SRAM address, as it will not be checked.

C Syntax

```
void ix_rm_atomic_sram_bit_set(
    ix_uint32 arg_SetValue,
    ix_uint32* arg_pSramValue
);
```

Input

`arg_SetValue` Value specifying which bits of the memory location addressed by `arg_pSramValue` should be set.

Input/Output

`arg_pSramValue` On input, specifies the value whose bits will be set.
Upon return, value at this location will have the bits specified by `arg_SetValue` argument set.

3.11.1.7 ix_rm_atomic_sram_bit_test_and_set()

This function atomically sets the bits of the `arg_pSramValue` address contents corresponding to the bits set to 1 in the `arg_SetValue` argument. Upon return, the function returns the value at `arg_pSramValue` location before the bit set operation. In order for the operation to complete successfully, the `arg_pSramValue` address must be an SRAM address, as it will not be checked.

C Syntax

```
ix_uint32 ix_rm_atomic_sram_bit_test_and_set(
    ix_uint32 arg_SetValue,
    ix_uint32* arg_pSramValue
);
```

Input

arg_SetValue Value specifying which bits of the memory location addressed by **arg_pSramValue** should be set.

Input/Output

arg_pSramValue On input, specifies the value whose bits will be set.
Upon return, value at this location will have the bits specified by **arg_SetValue** argument set.

Return

Return Value Returns the **arg_pSramValue** value before the bit set operation was performed.

3.11.1.8 **ix_rm_atomic_sram_bit_clear()**

This function atomically clears the bits of the **arg_pSramValue** address contents corresponding to the bits set to 1 in the **arg_ClearValue** argument. In order for the operation to complete successfully, the **arg_pSramValue** address must be an SRAM address, as it will not be checked.

C Syntax

```
void ix_rm_atomic_sram_bit_clear(
    ix_uint32 arg_ClearValue,
    ix_uint32* arg_pSramValue
);
```

Input

arg_ClearValue Value specifying which bits of the memory location addressed by **arg_pSramValue** should be cleared.

Input/Output

`arg_pSramValue` On input, specifies the value whose bits will be cleared.
 Upon return, value at this location will have the bits specified by `arg_SetValue` argument cleared.

3.11.1.9 ix_rm_atomic_sram_bit_test_and_clear()

This function atomically clears the bits of the `arg_pSramValue` address contents corresponding to the bits set to 1 in the `arg_ClearValue` argument. Upon return, the function returns the value at `arg_pSramValue` location before the bit clear operation. In order for the operation to complete successfully, the `arg_pSramValue` address must be an SRAM address, as it will not be checked.

C Syntax

```
ix_uint32 ix_rm_atomic_sram_bit_test_and_clear(
    ix_uint32 arg_ClearValue,
    ix_uint32* arg_pSramValue
);
```

Input

`arg_ClearValue` Value specifying which bits of the memory location addressed by `arg_pSramValue` should be cleared.

Input/Output

`arg_pSramValue` On input, specifies the value whose bits will be cleared.
 Upon return, value at this location will have the bits specified by `arg_ClearValue` argument cleared.

Return

Return Value Returns the `arg_pSramValue` value before the bit clear operation was performed.

3.11.1.10 ix_rm_atomic_scratch_swap()

This function performs a scratch memory atomic swap. The value referred by `arg_pValue1` is swapped with the value referred by `arg_pValue2`. On return `*arg_pValue1` will be changed to previous `arg_pValue2` and vice-versa. The operation is atomic only relative to the second argument of the swap operation (`*arg_pValue2`) and that only if that points to a SCRATCH memory location.

C Syntax

```
void ix_rm_atomic_scratch_swap(
    ix_uint32* arg_pValue1,
    ix_uint32* arg_pValue2
);
```

Input/Output

arg_pValue1	The address of the first operand of the swap operation—stored in scratch memory.
arg_pValue2	The address of the second operand of the swap operation—stored in scratch memory.

3.11.1.11 ix_rm_atomic_scratch_add()

This function increments the arg_pScratchValue address contents by arg_IncrementValue. The arg_pScratchValue address should be a SCRATCH address, as it will not be checked.

C Syntax

```
void ix_rm_atomic_scratch_add(
    ix_uint32 arg_IncrementValue,
    ix_uint32* arg_pScratchValue
);
```

Input

arg_IncrementValue	Value to be added to the memory location addressed by arg_pScratchValue.
--------------------	--

Input/Output

arg_pScratchValue	On input, location of the value to be incremented by arg_IncrementValue. Upon return, the incremented value.
-------------------	---

3.11.1.12 ix_rm_atomic_scratch_test_and_add()

This function increments the arg_pScratchValue address contents by arg_IncrementValue. The function also returns the value at arg_pScratchValue location before the increment operation completes. The arg_pScratchValue address should be an scratch address, as it will not be checked.

C Syntax

```

ix_uint32 ix_rm_atomic_scratch_test_and_add(
                                ix_uint32 arg_IncrementValue,
                                ix_uint32* arg_pScratchValue
                                );

```

Input

arg_IncrementValue Value to be added to the memory location addressed by **arg_pScratchValue**.

Input/Output

arg_pScratchValue On input, value at this location will be incremented by **arg_IncrementValue**.
Upon return, the incremented value.

Return

Return Value Returns the **arg_pScratchValue** value before the atomic add has been performed.

3.11.1.13 ix_rm_atomic_scratch_subtract()

This function decrements the **arg_pScratchValue** address contents by **arg_DecrementValue**. The **arg_pScratchValue** address should be a scratch address, as it will not be checked. The value at **arg_pScratchValue** saturates at 0. If you subtract a number that is larger in absolute value than the value at **arg_pScratchValue**, then the result at the **arg_pScratchValue** location will be 0.

C Syntax

```

void ix_rm_atomic_scratch_subtract(
                                ix_uint32 arg_DecrementValue,
                                ix_uint32* arg_pScratchValue
                                );

```

Input

arg_DecrementValue Value to be subtracted to the memory location addressed by **arg_pScratchValue**.

Input/Output

`arg_pScratchValue` On input, value at this location will be decremented by `arg_DecrementValue`.
Upon return, the decremented value.

3.11.1.14 ix_rm_atomic_scratch_test_and_subtract()

This function decrements the `arg_pScratchValue` address contents by `arg_DecrementValue`. The function also returns the value at `arg_pScratchValue` location before the decrement operation completes. The `arg_pScratchValue` address should be a scratch address, as it will not be checked. The value at `arg_pScratchValue` saturates at 0. If you subtract a positive number that is larger in absolute value than the value at `arg_pScratchValue`, then the result at the `arg_pSramValue` location will be 0.

C Syntax

```
ix_uint32 ix_rm_atomic_scratch_test_and_subtract(
                                                    ix_uint32 arg_DecrementValue,
                                                    ix_uint32* arg_pScratchValue
                                                    );
```

Input

`arg_DecrementValue` Value to be subtracted to the memory location addressed by `arg_pScratchValue`.

Input/Output

`arg_pScratchValue` On input, value at this location will be decremented by `arg_DecrementValue`.
Upon return, the decremented value.

Return

Return Value Returns the `arg_pScratchValue` value before the atomic subtract was performed.

3.11.1.15 ix_rm_atomic_scratch_bit_set()

This function atomically sets the bits of the `arg_pScratchValue` address contents corresponding to the bits set to 1 in the `arg_SetValue` argument. In order for the operation to complete successfully, the `arg_pScratchValue` address must be a scratch address, as it will not be checked.

C Syntax

```
void ix_rm_atomic_scratch_bit_set(
    ix_uint32 arg_SetValue,
    ix_uint32* arg_pScratchValue
);
```

Input

`arg_SetValue` Value specifying which bits of the memory location addressed by `arg_pScratchValue` should be set.

Input/Output

`arg_pScratchValue` On input, specifies the value whose bits will be set.
Upon return, value at this location will have the bits specified by `arg_SetValue` argument set.

3.11.1.16 ix_rm_atomic_scratch_bit_test_and_set()

This function atomically sets the bits of the `arg_pScratchValue` address contents corresponding to the bits set to 1 in the `arg_SetValue` argument. Upon return, the function returns the value at `arg_pScratchValue` location before the bit set operation. In order for the operation to complete successfully, the `arg_pScratchValue` address must be a scratch address, as it will not be checked.

C Syntax

```
ix_uint32 ix_rm_atomic_scratch_bit_test_and_set(
    ix_uint32 arg_SetValue,
    ix_uint32* arg_pScratchValue
);
```

Input

`arg_SetValue` Value specifying which bits of the memory location addressed by `arg_pScratchValue` should be set.

Input/Output

`arg_pScratchValue` On input, specifies the value whose bits will be set.
Upon return, value at this location will have the bits specified by `arg_SetValue` argument set.

Return

Return Value Returns the `arg_pScratchValue` value before the bit set operation was performed.

3.11.1.17 `ix_rm_atomic_scratch_bit_clear()`

This function atomically clears the bits of the `arg_pScratchValue` address contents corresponding to the bits set to 1 in the `arg_ClearValue` argument. In order for the operation to complete successfully, the `arg_pScratchValue` address must be a scratch address, as it will not be checked.

C Syntax

```
void ix_rm_atomic_scratch_bit_clear(
    ix_uint32 arg_ClearValue,
    ix_uint32* arg_pScratchValue
);
```

Input

`arg_ClearValue` Value specifying which bits of the memory location addressed by `arg_pScratchValue` should be cleared.

Input/Output

`arg_pScratchValue` On input, specifies the value whose bits will be cleared. Upon return, value at this location will have the bits specified by `arg_SetValue` argument cleared.

3.11.1.18 `ix_rm_atomic_scratch_bit_test_and_clear()`

This function atomically clears the bits of the `arg_pScratchValue` address contents corresponding to the bits set to 1 in the `arg_ClearValue` argument. Upon return, the function returns the value at `arg_pScratchValue` location before the bit clear operation. In order for the operation to complete successfully, the `arg_pScratchValue` address must be a scratch address, as it will not be checked.

C Syntax

```
ix_uint32 ix_rm_atomic_scratch_bit_test_and_clear(
    ix_uint32 arg_ClearValue,
    ix_uint32* arg_pScratchValue
);
```


Input

`arg_ClearValue` Value specifying which bits of the memory location addressed by `arg_pScratchValue` should be cleared.

Input/Output

`arg_pScratchValue` On input, specifies the value whose bits will be cleared. Upon return, value at this location will have the bits specified by `arg_ClearValue` argument cleared.

Return

Return Value Returns the `arg_pScratchValue` value before the bit clear operation was performed.

3.11.1.19 ix_rm_managed_to_os_memory_copy()

This function copies a managed memory area into operating system memory. This function uses only the CPU fast instructions if data are 4-byte aligned. If data are not 4-byte aligned, a regular memory copy is performed.

Note: The semantics of this call are patterned after the standard C function, `memcpy()`.

C Syntax

```
void* ix_rm_managed_to_os_memory_copy(
    void* arg_pDest,
    const void* arg_pSrc,
    ix_uint32 arg_Count);
```

Inputs

`arg_pSrc` A pointer to the managed memory location which is the source of the data to copy.

`arg_Count` The amount of data to copy.

Output/Return

<code>arg_pDest</code>	A pointer to the destination operating system buffer the data are copied to.
Return Value	Returns a pointer to the destination operating system buffer. This pointer is identical to <code>arg_pDest</code> .

3.11.1.20 `ix_rm_os_to_managed_memory_copy()`

This function copies an operating system memory area into managed memory. This function uses only the CPU fast instructions if data are 4-byte aligned. If data are not 4-byte aligned, a regular memory copy is performed.

Note: The semantics of this call are patterned after the standard C function, `memcpy()`.

C Syntax

```
void* ix_rm_os_to_managed_memory_copy(
    void* arg_pDest,
    const void* arg_pSrc,
    ix_uint32 arg_Count);
```

Inputs

<code>arg_pSrc</code>	A pointer to the operating system buffer which is the source of the data to copy.
<code>arg_Count</code>	The amount of data to copy.

Output/Return

<code>arg_pDest</code>	A pointer to the destination operating system memory location the data are copied to.
Return Value	Returns a pointer to the destination buffer. This pointer is identical to <code>arg_pDest</code> .

3.11.1.21 `ix_rm_managed_to_managed_memory_copy()`

This function copies one managed memory block into another managed memory block. This function uses only the CPU fast instructions if data are 4-byte aligned. If data are not 4-byte aligned, a regular memory copy is performed.

Note: The semantics of this call are patterned after the standard C function, `memcpy()`.

```
void* ix_rm_managed_to_managed_memory_copy(
```



```
void* arg_pDest,
const void* arg_pSrc,
ix_uint32 arg_Count);
```

Inputs

<code>arg_pSrc</code>	A pointer to the managed memory location the data are copied from.
<code>arg_Count</code>	The amount of data to be copied.

Output/Return

<code>arg_pDest</code>	A pointer to the destination managed memory location the data are copied to.
Return Value	Returns a pointer to the destination buffer. This pointer is identical to <code>arg_pDest</code> .

3.12 Hash API

The Hash API provides hash operations for Intel XScale® core applications. In the current implementation, the programmer can choose to perform the hash operations with hardware support from the chip's hash unit or in software. By default the resource manager uses hardware support from the hash unit. In order to change this behavior, the `_IX_RM_IMPL_SOFTWARE_HASH_` symbol must be defined on the command line at the time the Resource Manager is compiled.

The Hash API provides 48-bit, 64-bit, and 128-bit hash operations based on application-provided 48-bit, 64-bit, and 128-bit multipliers. The Hash API should be used to achieve a uniform distribution of 48-bit, 64-bit, or 128-bit numbers. If the distribution is not satisfactory, then the hash multipliers can be changed.

The hash algorithm is explained in great detail in the *Intel® IXP2400/IXP2800 Network Processor Programmer's Reference Manual* on the IXA SDK Tools CD.

Table 3-19 summarizes the hash API.

Table 3-19. Resource Manager Hash API

Name	Description
<code>ix_hash_48</code>	This type represents a 48-bit hash data type.
<code>ix_hash_64</code>	This type represents a 64-bit hash data type.
<code>ix_hash_128</code>	This type defines the 128-bit hash data type.
<code>ix_hash_multiplier_48</code>	This type defines the 48-bit multiplier data type.
<code>ix_hash_multiplier_64</code>	This type defines the 64-bit multiplier data type.
<code>ix_hash_multiplier_128</code>	This type defines the 128-bit multiplier data type.
<code>ix_rm_hash_48_hash()</code>	This function performs a 48-bit hash operation.

Table 3-19. Resource Manager Hash API (Continued)

Name	Description
<code>ix_rm_hash_48_multiplier_set()</code>	This function sets a new multiplier value for the 48-bit hash operations.
<code>ix_rm_hash_48_multiplier_get()</code>	The function retrieves the current multiplier value for 48-bit hash operations.
<code>ix_rm_hash_64_hash()</code>	This function performs a 64-bit hash operation.
<code>ix_rm_hash_64_multiplier_set()</code>	This function sets a new multiplier value for the 64-bit hash operations.
<code>ix_rm_hash_64_multiplier_get()</code>	The function retrieves the current multiplier value for 64-bit hash operations.
<code>ix_rm_hash_128_hash()</code>	This function performs a 128-bit hash operation.
<code>ix_rm_hash_128_multiplier_set()</code>	This function sets a new multiplier value for the 128-bit hash operations.
<code>ix_rm_hash_128_multiplier_get()</code>	The function retrieves the current multiplier value for 128-bit hash operations.

3.12.1 Defined Types, Enumerations, and Data Structures

3.12.1.1 `ix_hash_48`

This type represents a 48-bit hash data type. A reference to a variable of this type is passed to the hash operation containing the value to be hashed. On return the variable contains the hashed value corresponding to the value input. The member `m_LW0` represents longword zero of the 48-bit value of interest. The member `m_LW1` represents longword one. For `m_LW1`, only the least significant 16 bits are considered for the hash operation.

C Syntax

```
typedef struct ix_s_hash_48 {
    ix_uint32 m_LW0;
    ix_uint32 m_LW1;
} ix_hash_48;
```

Members

<code>m_LW0</code>	The first longword—that is, the first 32 bits—of the 48-bit value of interest.
<code>m_LW1</code>	The second longword—representing the upper 16 bits—of the 48-bit value of interest.

NOTE: Only the first 16 bits of this longword are considered.

3.12.1.2 `ix_hash_64`

This type represents a 64-bit hash data type. A reference to a variable of this type is passed to the hash operation containing the value to be hashed. On return the variable contains the hashed value corresponding to the input value. The member `m_LW0` represents longword zero of the 64-bit value of interest. The member `m_LW1` represents longword one.

C Syntax

```
typedef struct ix_s_hash_64 {
    ix_uint32 m_LW0;
    ix_uint32 m_LW1;
} ix_hash_64;
```

Members

m_LW0	The first longword—that is, the first 32 bits—of the 64-bit value of interest.
m_LW1	The second longword—that is, the upper 32 bits—of the 64-bit value of interest.

3.12.1.3 ix_hash_128

This type defines the 128-bit hash data type. A reference to a variable of this type is passed to the hash operation containing the value to be hashed. On return the variable contains the hashed value corresponding to the input value. The member m_LW0 represents longword zero of the 128-bit value of interest, member m_LW1 represents longword one, member m_LW2 represents longword two, and member m_LW3 represents longword three.

C Syntax

```
typedef struct ix_s_hash_128 {
    ix_uint32 m_LW0;
    ix_uint32 m_LW1;
    ix_uint32 m_LW2;
    ix_uint32 m_LW3;
} ix_hash_128;
```

Members

m_LW0	The first longword of the 128-bit hash value.
m_LW1	The second longword of the 128-bit hash value.
m_LW2	The third longword of the 128-bit hash value.
m_LW3	The fourth longword of the 128-bit hash value.

3.12.1.4 ix_hash_multiplier_48

This type defines the 48-bit multiplier data type. The multiplier is used to perform a 48-bit hash operation, and different values of the multiplier yield different hashed values and different number distributions.

C Syntax

```
typedef ix_hash_48 ix_hash_multiplier_48;
```

3.12.1.5 **ix_hash_multiplier_64**

This type defines the 64-bit multiplier data type. The multiplier is used to perform a 64-bit hash operation, and different values of the multiplier will yield different hashed values and different number distributions.

C Syntax

```
typedef ix_hash_64 ix_hash_multiplier_64;
```

3.12.1.6 **ix_hash_multiplier_128**

This type defines the 128-bit multiplier data type. The multiplier is used to perform a 128-bit hash operation, and different values of the multiplier will yield different hashed values and different number distributions.

C Syntax

```
typedef ix_hash_128 ix_hash_multiplier_128;
```

3.12.2 **API Functions**

3.12.2.1 **ix_rm_hash_48_hash()**

This function performs a 48-bit hash operation. The argument is a reference to the value to be hashed on input, and a reference to the hashed value on output.

Before performing this operation the 48-bit hash multiplier must be properly set. At power up the multiplier is initialized to zero.

C Syntax

```
ix_error ix_rm_hash_48_hash( ix_hash_48* arg_pHash48 );
```

Input/Output

arg_pHash48	On input this argument is a reference to the data to be hashed and on output it is a reference to the hashed value.
--------------------	---

Output/Returns

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

3.12.2.2 `ix_rm_hash_48_multiplier_set()`

This function sets a new multiplier value for the 48-bit hash operations.

C Syntax

```
ix_error ix_rm_hash_48_multiplier_set(
    const ix_hash_multiplier_48* arg_pHashMultiplier48 );
```

Input

`arg_pHashMultiplier48` The new 48-bit hash multiplier value.

Output/Returns

Return Value Returns `IX_SUCCESS` if successful or a valid `ix_error` token for failure.

3.12.2.3 `ix_rm_hash_48_multiplier_get()`

The function retrieves the current multiplier value for 48-bit hash operations.

C Syntax

```
ix_error ix_rm_hash_48_multiplier_get(
    ix_hash_multiplier_48* arg_pHashMultiplier48 );
```

Output/Returns

`arg_pHashMultiplier48` The location where, when the call successfully returns, the 48-bit hash multiplier will be stored.

Return Value Returns `IX_SUCCESS` if successful or a valid `ix_error` token for failure.

3.12.2.4 `ix_rm_hash_64_hash()`

This function performs a 64-bit hash operation. The argument is a reference to the value to be hashed on input, and a reference to the hashed value on output.

Before performing this operation the 64-bit hash multiplier must be properly set. At power up the multiplier is initialized to zero.

C Syntax

```
ix_error ix_rm_hash_64_hash( ix_hash_64* arg_pHash64 );
```


Input/Output

<code>arg_pHash64</code>	On input this argument is a reference to the data to be hashed and on output it is a reference to the hashed value.
--------------------------	---

Output/Returns

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

3.12.2.5 `ix_rm_hash_64_multiplier_set()`

This function sets a new multiplier value for the 64-bit hash operations.

C Syntax

```
ix_error ix_rm_hash_64_multiplier_set(
    const ix_hash_multiplier_64* arg_pHashMultiplier64 );
```

Input

<code>arg_pHashMultiplier64</code>	The new 64-bit hash multiplier value.
------------------------------------	---------------------------------------

Output/Returns

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

3.12.2.6 `ix_rm_hash_64_multiplier_get()`

The function retrieves the current multiplier value for 64-bit hash operations.

C Syntax

```
ix_error ix_rm_hash_64_multiplier_get(
    ix_hash_multiplier_64* arg_pHashMultiplier64 );
```


Output/Returns

<code>arg_pHashMultiplier64</code>	The location where, when the call successfully returns, the 64-bit hash multiplier will be stored.
Return Value	Returns <code>IX_SUCCESS</code> if successful or a valid <code>ix_error</code> token for failure.

3.12.2.7 `ix_rm_hash_128_hash()`

This function performs a 128-bit hash operation. The argument is a reference to the value to be hashed on input, and a reference to the hashed value on output.

Before performing this operation the 128-bit hash multiplier must be properly set. At power up the multiplier is initialized to zero.

C Syntax

```
ix_error ix_rm_hash_128_hash( ix_hash_128* arg_pHash128 );
```

Input/Output

<code>arg_pHash128</code>	On input this argument is a reference to the data to be hashed and on output it is a reference to the hashed value.
---------------------------	---

Output/Returns

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

3.12.2.8 `ix_rm_hash_128_multiplier_set()`

This function sets a new multiplier value for the 128-bit hash operations.

C Syntax

```
ix_error ix_rm_hash_128_multiplier_set(
    const ix_hash_multiplier_128* arg_pHashMultiplier128 );
```

Input

<code>arg_pHashMultiplier128</code>	The new 128-bit hash multiplier value.
-------------------------------------	--

Output/Returns

Return Value Returns `IX_SUCCESS` if successful or a valid `ix_error` token for failure.

3.12.2.9 ix_rm_hash_128_multiplier_get()

The function retrieves the current multiplier value for 128-bit hash operations.

C Syntax

```
ix_error ix_rm_hash_128_multiplier_get(
    ix_hash_multiplier_128* arg_pHashMultiplier128 );
```

Output/Returns

`arg_pHashMultiplier128` The location where, when the call successfully returns, the 128-bit hash multiplier will be stored.

Return Value Returns `IX_SUCCESS` if successful or a valid `ix_error` token for failure.

3.13 Microengine Services API

The Microengine Services API provides several sets of APIs, which are used to coordinate operations between the Intel XScale® core and the microengines. The following operations are supported:

- Locking and unlocking mechanisms
- Reading and writing of microengine transfer registers
- Sending a notification signal to a microengine

Note: In some places in this document, microengine is abbreviated as ME.

Locks between the microengines and the Intel XScale® core are used to synchronize data access operations. These locks are based on the SCRATCH memory atomic operations. On the Intel XScale® core side, the API suffices to do lock/unlock operations. For a particular lock object, the Intel XScale® core-based application must retrieve the SCRATCH byte offset of the underlying SCRATCH memory location used for atomic set and test and clear and the bit position of the lock into that memory location, and patch these values into the microcode (or pass it through any other means).

On the microengine side, in order for microengines to acquire the lock, they must perform a bit test and clear for the passed SCRATCH offset on the bit position passed from the Intel XScale® core. The returned value is masked for the Intel XScale® core passed bit position of the lock. If the

resulting value from the operation is nonzero, then there was a successful locking, otherwise it means that the lock is held by somebody else. In order to release a held lock, the microengines must perform a bit set operation for the SCRATCH offset and bit position of the lock.

On the Intel XScale® core side, the API guarantees if the microengine acquired the lock, then the lock will not be released by the Intel XScale® core. On the microengine side, it is the responsibility of the programmers to insure this behavior (the same microengine context that acquired the lock will be the one that will release it). In this release, 64 locks for microengine - Intel XScale® core operations are supported by the system, but that value can be changed as required.

The microengine services API also allows the Intel XScale® core applications to read and write microengine transfer registers and to signal microengines. The signaling is achieved by writing the SAME_ME_REGISTER for the desired microengine. There is a 3 cycles latency from the write CSR command until the CSR is actually written, and 8 cycles until the microengine is signaled after the CSR write. That should be considered when issuing many signals to the same microengine. During the call, the code checks if the microengine is in the 4 contexts mode or 8 contexts mode. In the first case, the valid context numbers are 0, 2, 4, and 6. In the second case, they are in the range 0-7.

Table 3-20 lists data structures and functions included in this API.

Table 3-20. Resource Manager Microengine Services API

Functions and Data Structures	Description
<code>ix_me_xscale_lock_handle</code>	A microengine-to-Intel XScale® core lock handle
<code>ix_me_xscale_lock_status</code>	Type describing all possible states of a microengine-to-Intel XScale® core lock.
<code>ix_me_xscale_lock_owner</code>	Type describing all possible owners of a microengine-to-Intel XScale® core lock.
<code>ix_me_xscale_lock_info</code>	Structure providing information about a microengine-to-Intel XScale® core lock.
<code>ix_me_transfer_register_type</code>	Enumerated type describing all types of ME transfer registers.
<code>ix_rm_me_xscale_lock_new()</code>	Creates a new microengine-to-Intel XScale® core lock object.
<code>ix_rm_me_xscale_lock_delete()</code>	Deletes the specified microengine-to-Intel XScale® core lock object.
<code>ix_rm_me_xscale_lock_acquire()</code>	Acquires the specified microengine-to-Intel XScale® core lock.
<code>ix_rm_me_xscale_lock_release()</code>	Releases the specified microengine-to-Intel XScale® core lock.
<code>ix_rm_me_xscale_lock_get_info()</code>	Returns useful information about a microengine-to-Intel XScale® core lock.
<code>ix_rm_me_transfer_register_read()</code>	Reads the value of a ME transfer register.
<code>ix_rm_me_transfer_register_write()</code>	Writes the value of a ME transfer register.
<code>ix_rm_me_signal()</code>	Sends a specified signal to a ME.

3.13.1 Defined Types, Enumerations, and Data Structures

3.13.1.1 `ix_me_xscale_lock_handle`

This type defines a generic microengine-to-Intel XScale® core lock handle. On the Intel XScale® core side, the locks are identified by one of these handles.

C Syntax

```
typedef ix_handle ix_me_xscale_lock_handle;
```


3.13.1.2 ix_me_xscale_lock_status

This enumerated type defines all possible states for microengine-to-Intel XScale® core locks.

C Syntax

```
typedef enum ix_e_me_xscale_lock_status
{
    IX_ME_XSCALE_LOCK_STATUS_FIRST = 0,
    IX_ME_XSCALE_LOCK_STATUS_AVAILABLE = IX_ME_XSCALE_LOCK_STATUS_FIRST,
    IX_ME_XSCALE_LOCK_STATUS_UNAVAILABLE,
    IX_ME_XSCALE_LOCK_STATUS_LAST
} ix_me_xscale_lock_status;
```

3.13.1.3 ix_me_xscale_lock_owner

This enumerated type describes all possible owners of a microengine-to-Intel XScale® core lock.

C Syntax

```
typedef enum ix_e_me_xscale_lock_owner
{
    IX_ME_XSCALE_LOCK_OWNER_FIRST = 0,
    IX_ME_XSCALE_LOCK_OWNER_NONE = IX_ME_XSCALE_LOCK_OWNER_FIRST,
    IX_ME_XSCALE_LOCK_OWNER_XSCALE,
    IX_ME_XSCALE_LOCK_OWNER_ME,
    IX_ME_XSCALE_LOCK_OWNER_LAST
} ix_me_xscale_lock_owner;
```

3.13.1.4 ix_me_xscale_lock_info

This structure describes the information associated with a microengine-to-Intel XScale® core lock.

C Syntax

```
typedef struct ix_s_me_xscale_lock_info
{
    ix_uint32 m_ScratchOffset;
    ix_uint32 m_BitPosition;
    ix_me_xscale_lock_status m_Status;
    ix_me_xscale_lock_owner m_Owner;
} ix_me_xscale_lock_info;
```


Data Members

<code>m_ScratchOffset</code>	Byte offset of the underlying SCRATCH memory location of the lock.
<code>m_BitPosition</code>	Bit position of the lock into the underlying SCRATCH memory location.
<code>ix_me_xscale_lock_status m_Status</code>	Status of the lock.
<code>ix_me_xscale_lock_owner m_Owner</code>	Owner of the lock.

3.13.1.5 `ix_me_transfer_register_type`

This enumerated type describes all microengine transfer register types. The Intel XScale® core can read the SRAM write transfer registers of a microengine and can write the SRAM or DRAM read transfer registers of a microengine.

C Syntax

```
typedef enum ix_e_me_transfer_register_type
{
    IX_ME_TRANSFER_REGISTER_TYPE_FIRST = 0,
    IX_ME_TRANSFER_REGISTER_TYPE_SRAM = IX_ME_TRANSFER_REGISTER_TYPE_FIRST,
    IX_ME_TRANSFER_REGISTER_TYPE_DRAM,
    IX_ME_TRANSFER_REGISTER_TYPE_LAST
} ix_me_transfer_register_type;
```

3.13.2 API Functions

3.13.2.1 `ix_rm_me_xscale_lock_new()`

This function creates a new microengine-to-Intel XScale® core lock object and returns a handle to the newly created lock. The returned handle must be used for all subsequent operations on the lock. The maximum number of microengine-to-Intel XScale® core locks is set to 64 but is configurable at compile-time.

C Syntax

```
ix_error ix_rm_me_xscale_lock_new(
    ix_me_xscale_lock_handle* arg_pMeXScaleLockHandle
);
```


Output/Return

`arg_pMeXScaleLockHandle` Location where a handle to the newly created microengine-to-Intel XScale® core lock will be stored upon return.

Return Value IX_SUCCESS if successful or a valid `ix_error` token for failure.

3.13.2.2 `ix_rm_me_xscale_lock_delete()`

This function deletes the specified microengine-to-Intel XScale® core lock object and releases all resources associated with it. Once deleted, a lock cannot be accessed by the Intel XScale® core, although the microengines can still access the lock. This approach is not recommended.

C Syntax

```
ix_error ix_rm_me_xscale_lock_delete(
    ix_me_xscale_lock_handle arg_hMeXScaleLock
);
```

Input

`arg_hMeXScaleLock` Handle of the microengine-to-Intel XScale® core lock to be deleted.

Return

Return Value IX_SUCCESS if successful or a valid `ix_error` token for failure.

3.13.2.3 `ix_rm_me_xscale_lock_acquire()`

This function retrieves the specified microengine-to-Intel XScale® core lock on behalf of the calling Intel XScale® core application. If the lock is already taken, then an error is returned.

C Syntax

```
ix_error ix_rm_me_xscale_lock_acquire(
    ix_me_xscale_lock_handle arg_hMeXScaleLock
);
```

Input

`arg_hMeXScaleLock` Handle of the lock to acquire.

Return

Return Value IX_SUCCESS if successful or a valid ix_error token for failure.

3.13.2.4 ix_rm_me_xscale_lock_release()

This function releases the specified microengine-to-Intel XScale® core lock. If the lock has been acquired by the microengines, an error is returned.

If the lock is in the IX_ME_XSCALE_LOCK_STATUS_AVAILABLE state, then the action will result in no-op. A microengine-to-Intel XScale® core lock can be released by another thread of control than the one that acquired it.

C Syntax

```
ix_error ix_rm_me_xscale_lock_release(
    ix_me_xscale_lock_handle arg_hMeXScaleLock
);
```

Input

arg_hMeXScaleLock Handle of the lock to release.

Return

Return Value IX_SUCCESS if successful or a valid ix_error token for failure.

3.13.2.5 ix_rm_me_xscale_lock_get_info()

This function returns useful information associated with the passed microengine-to-Intel XScale® core lock. This function should be called to retrieve the SCRATCH memory offset and bit position associated with the lock before passing this information to microengines.

C Syntax

```
ix_error ix_rm_me_xscale_lock_get_info(
    ix_me_xscale_lock_handle arg_hMeXScaleLock,
    ix_me_xscale_lock_info* arg_pMeXScaleLockInfo
);
```

Input

arg_hMeXScaleLock Lock handle for the information.

Output/Return

`arg_pMeXScaleLockInfo` Address of structure where the lock information will be stored upon return.

Return Value IX_SUCCESS if successful or a valid `ix_error` token for failure.

3.13.2.6 `ix_rm_me_transfer_register_read()`

This function reads the value of a microengine transfer register for the specified microengine, context, and register type. Only the `IX_ME_TRANSFER_REGISTER_TYPE_SRAM` registers can be read.

During the call, the code checks if the microengine is in the 4 context mode or 8 context mode. In the first case, the valid context numbers are 0, 2, 4, 6; for the second case, they are in the range 0-7. For 4 context mode, the value of the index is in the range 0-31; for 8 context mode, the index value range is 0-15.

C Syntax

```
ix_error ix_rm_me_transfer_register_read(
    ix_uint32 arg_MENumber,
    ix_uint32 arg_ContextNumber,
    ix_me_transfer_register_type arg_RegisterType,
    ix_uint32 arg_RegisterIndex,
    ix_uint32* arg_pRegisterValue
);
```

Inputs

<code>arg_MENumber</code>	Microengine whose transfer register is to be read.
<code>arg_ContextNumber</code>	Microengine context number whose transfer register is to be read.
<code>arg_RegisterType</code>	Type of the transfer register to read. For read operations, only the <code>IX_ME_TRANSFER_REGISTER_TYPE_SRAM</code> type is supported.
<code>arg_RegisterIndex</code>	Index of the ME transfer register to be read. The value of the index is in the range 0-31 if the microengine is in 4 contexts mode. The index value is in the range 0-15 for 8 contexts mode.

Output/Return

`arg_pRegisterValue` Address where the value of the microengine transfer register will be stored upon return.

Return Value IX_SUCCESS if successful or a valid `ix_error` token for failure.

3.13.2.7 `ix_rm_me_transfer_register_write()`

This function writes the value of a microengine transfer register for the specified microengine, context and register type. During the call, the code checks if the microengine is in the 4 context mode or 8 context mode. In the first case, the valid context numbers are 0, 2, 4, 6; in the second case, they are in the range 0-7. For 4 context mode, the value of the index is in the range 0-31; for 8 context mode, the index value range is 0-15.

C Syntax

```
ix_error ix_rm_me_transfer_register_write(
    ix_uint32 arg_MENumber,
    ix_uint32 arg_ContextNumber,
    ix_me_transfer_register_type arg_RegisterType,
    ix_uint32 arg_RegisterIndex,
    ix_uint32 arg_RegisterValue
);
```

Inputs

<code>arg_MENumber</code>	Microengine whose transfer register is to be written.
<code>arg_ContextNumber</code>	Microengine context number whose transfer register is to be written.
<code>arg_RegisterType</code>	Type of the transfer register to write.
<code>arg_RegisterIndex</code>	Index of the ME transfer register to be written. The value of the index is in the range 0-31 if the microengine is in 4 context mode. The index value is in the range 0-15 for 8 context mode.

Return

Return Value `IX_SUCCESS` if successful or a valid `ix_error` token for failure.

3.13.2.8 `ix_rm_me_signal()`

This function sends a specified signal to the specified microengine and context. The signaling is achieved by writing the `SAME_ME_REGISTER` for the desired microengine and context. There is a 3 cycle latency from the write CSR command until the CSR is actually written, and 8 cycles until the microengine will signaled after the CSR write. That should be considered when issuing many signals to the same microengine.

During the call, the code checks if the microengine is in the 4 context mode or 8 context mode. In the first case, the valid context numbers are 0, 2, 4, and 6; in the second case, they are in the range 0-7.

C Syntax

```
ix_error ix_rm_me_signal(
```



```

        ix_uint32 arg_MENumber,
        ix_uint32 arg_ContextNumber,
        ix_uint32 arg_SignalNumber
    );

```

Inputs

arg_MENumber	Microengine to be signaled.
arg_ContextNumber	Context number of the microengine to be signaled.
arg_SignalNumber	Signal number to be sent to the specified microengine context. Valid values are in the range 1-15.

Return

Return Value IX_SUCCESS if successful or a valid ix_error token for failure.

3.14 Debug Support API

The Debug Support API provides a series of functions that provide debugging features to the programmers. In order for debug features to be turned on, the resource manager library must be compiled with the `_IX_RM_DEBUG_` preprocessor symbol defined. For the debug builds, this particular symbol is automatically defined. This symbol can be turned off if debug functions are not needed.

Some debugging functionality specific to buffers is compiled in only if the `_IX_RM_BUFFER_DEBUG_` preprocessor symbol is defined. This symbol is automatically defined when the `_IX_RM_DEBUG_` symbol is defined, but it can be turned off independently of `_IX_RM_DEBUG_`. By including the debugging support, the performance of the system is affected because extra processing is done, especially when `_IX_RM_BUFFER_DEBUG_` symbol is defined in the area of buffer processing.

For the VxWorks operating system, the debugging functions can be called exactly as they are described from the shell. In the case of Linux, the resource manager creates an entry in */proc* file system called *rm_debug*.

To see how to invoke the described functions, type *more /proc/rm_debug* at the command shell, and information on how to issue the commands is printed. The way a certain function should be called is as follows: *echo "function_name [param1] [param2]" > /proc/rm_debug* where *[param1] [param2]* means that some of the functions need no parameter, some need one parameter, and some need two parameters.

Note: The debugging functions can be called only after the resource manager has been initialized, so just inserting the resource manager module in the kernel won't be enough. One other module should

call `ix_rm_init()` before the functions can be called. This statement is valid for both VxWorks and Linux operating systems.

Table 3-21 lists functions included in this API.

Table 3-21. Resource Manager Debug Support API

Functions	Description
<code>ix_rm_mem_status_print()</code>	Prints information about a memory manager associated with a certain memory type and channel.
<code>ix_rm_scratch_ring_print_info()</code>	Prints information about a SCRATCH ring.
<code>ix_rm_scratch_ring_print_data()</code>	Prints the data belonging to a SCRATCH ring.
<code>ix_rm_sram_ring_print_info()</code>	Prints information about an SRAM ring.
<code>ix_rm_sram_ring_print_data()</code>	Prints the data belonging to an SRAM ring.
<code>ix_rm_free_list_print_available_buffers()</code>	Prints the number of available buffers in a certain free list.
<code>ix_rm_free_list_print_buffers_info()</code>	Prints the allocation information for the buffers belonging to a certain hardware free list.
<code>ix_rm_free_list_print_info()</code>	Prints information related to certain free list.
<code>ix_rm_buffer_print_meta()</code>	Prints the buffer meta information for the passed handle.
<code>ix_rm_buffer_print_data()</code>	Prints the buffer data for the passed handle.
<code>ix_rm_buffer_print_debug_info()</code>	Prints the buffer debug information for the passed handle.

3.14.1 API Functions

3.14.1.1 `ix_rm_mem_status_print()`

This function prints a snapshot of a memory manager associated with the passed memory type and channel. All allocated and free memory cells will be listed.

C Syntax

```
ix_error ix_rm_mem_status_print(
    ix_memory_type arg_MemType,
    ix_uint32 arg_MemChannel
);
```

Inputs

<code>arg_MemType</code>	Memory type whose status is to be displayed. Values include: <ul style="list-style-type: none"> 0 = DRAM 1 = SRAM 2 = SCRATCH
<code>arg_MemChannel</code>	Channel whose status information is to be printed.

Return

Return Value IX_SUCCESS if successful or a valid ix_error token for failure.

3.14.1.2 ix_rm_scratch_ring_print_info()

This function prints information about the requested SCRATCH ring number. The SCRATCH rings numbers are in the range 0-15.

C Syntax

```
void ix_rm_scratch_ring_print_info(
    ix_uint32 arg_RingNumber
);
```

Input

arg_RingNumber Scratch ring number whose info is to be displayed.

3.14.1.3 ix_rm_scratch_ring_print_data()

This function prints data belonging to the requested SCRATCH ring number. The SCRATCH rings numbers are in the range 0-15.

C Syntax

```
void ix_rm_scratch_ring_print_data(
    ix_uint32 arg_RingNumber
);
```

Input

arg_RingNumber Scratch ring number whose info is to be displayed.

3.14.1.4 ix_rm_sram_ring_print_info()

This function prints information about the requested SRAM ring number on the specified channel. The SRAM ring numbers are in the range 0-63, and they correspond to QArray descriptor entries in the SRAM controller.

An SRAM ring is initialized only after some microcode copies the QArray descriptors from SRAM into the SRAM controller. After this operation is completed, even if the core creates a new SRAM ring, the ring will not be valid as the information has not been copied into the controller. Some of the QArray descriptor entries can be used for queues or journals; in that case, an appropriate error message will be printed.

C Syntax

```
void ix_rm_sram_ring_print_info(
    ix_uint32 arg_ChannelNumber,
    ix_uint32 arg_RingNumber
);
```

Inputs

<code>arg_ChannelNumber</code>	Channel number of the SRAM ring whose info is to be displayed.
<code>arg_RingNumber</code>	SRAM ring identifier whose info is to be displayed.

3.14.1.5 ix_rm_sram_ring_print_data()

This function prints the data belonging to the requested SRAM ring number on the specified channel. The SRAM rings numbers are in the range 0-63, and they correspond to QArray descriptors entries in the SRAM controller.

An SRAM ring is initialized only after some microcode copies the QArray descriptors from SRAM into the SRAM controller. After this operation is completed, even if the core creates a new SRAM ring, the ring will not be valid as the information has not been copied into the controller. Some of the QArray descriptor entries can be used for queues or journals; in that case, an appropriate error message will be printed.

C Syntax

```
void ix_rm_sram_ring_print_data(
    ix_uint32 arg_ChannelNumber,
    ix_uint32 arg_RingNumber
);
```

Inputs

<code>arg_ChannelNumber</code>	Channel number of the SRAM ring whose data is to be displayed.
<code>arg_RingNumber</code>	SRAM ring identifier whose data is to be displayed.

3.14.1.6 ix_rm_free_list_print_available_buffers()

This function prints the number of available buffers in a certain free list. On each channel, the hardware free list numbers are in the range 0 through 15. For software free lists, the numbers are in the range 0 through [255 - (SRAM_Channels * 16)].

If hardware free lists are allowed on all channels and there are a maximum of 16 hardware free lists on each channel, use the following equation to retrieve information about a hardware free list on channel *n*: $\text{arg_FreeListNumber} = n * 16 + \text{the_free_list_number}$ on that channel.

C Syntax

```
void ix_rm_free_list_print_available_buffers(
    ix_uint32 arg_FreeListType,
    ix_uint32 arg_FreeListNumber
);
```

Inputs

<code>arg_FreeListType</code>	Indicates the type of free list whose info is to be displayed. Values are: <ul style="list-style-type: none"> • 0 = hardware • 1 = software
<code>arg_FreeListNumber</code>	Free list identifier whose info is to be displayed.

3.14.1.7 ix_rm_free_list_print_buffers_info()

This function prints allocation information for the buffers owned by the specified hardware free list.

If hardware free lists are allowed on all channels and there are a maximum of 16 hardware free lists on each channel, use the following equation to retrieve information about a hardware free list on channel *n*: `arg_FreeListNumber = n*16 + the_free_list_number` on that channel.

C Syntax

```
void ix_rm_free_list_print_buffers_info(
    ix_uint32 arg_FreeListNumber
);
```

Input

<code>arg_FreeListNumber</code>	Free list identifier whose buffer allocation info is to be displayed.
---------------------------------	---

3.14.1.8 ix_rm_free_list_print_info()

This function prints detailed free list information about the requested free list. On each channel, the hardware free list numbers are in the range 0 through 15. For software free lists, the numbers are in the range 0 through [255 - (SRAM_Channels * 16)].

If hardware free lists are allowed on all channels and there are a maximum of 16 hardware free lists on each channel, use the following equation to retrieve information about a hardware free list on channel *n*: `arg_FreeListNumber = n*16 + the_free_list_number` on that channel.

C Syntax

```
void ix_rm_free_list_print_info(
    ix_uint32 arg_FreeListType,
```



```

        ix_uint32 arg_FreeListNumber
    );

```

Inputs

`arg_FreeListType` Indicates the type of free list whose info is to be displayed. Values are:

- 0 = hardware
- 1 = software

`arg_FreeListNumber` Free list identifier whose info is to be displayed.

3.14.1.9 `ix_rm_buffer_print_meta()`

This function prints buffer metadata for the passed buffer handle.

C Syntax

```

void ix_rm_buffer_print_meta(
    ix_buffer_handle arg_hBuffer
);

```

Input

`arg_hBuffer` Buffer handle whose metadata is to be displayed.

3.14.1.10 `ix_rm_buffer_print_data()`

This function prints buffer data for the passed buffer handle.

C Syntax

```

void ix_rm_buffer_print_data(
    ix_buffer_handle arg_hBuffer
);

```

Input

`arg_hBuffer` Buffer handle whose data is to be displayed.

3.14.1.11 `ix_rm_buffer_print_debug_info()`

This function prints buffer debug information for the passed buffer.

C Syntax

```
void ix_rm_buffer_print_debug_info(  
    ix_buffer_handle arg_hBuffer  
);
```

Input

arg_hBuffer Buffer handle whose debug information is to be displayed.

Core Component Infrastructure

4

This chapter describes the core component infrastructure interface providing framework support for:

- Running core components in their own execution engines—where each execution engine encapsulates a calling application thread of control
- Prioritizing message and packet data paths

Table 4-1 summarizes the core-component infrastructure API.

Table 4-1. cci API

Name	Description
<code>ix_cci_cc_add_event_handler()</code>	Adds an event handler using the handle of the component.
<code>ix_event_func()</code>	The function prototype specifying the signature for an event-handler function provided by the calling application.
<code>ix_cci_cc_add_event_handler_ex()</code>	Similar to <code>ix_cci_cc_add_event_handler()</code> but instead of specifying the core component or engine context this function allows the calling application to specify the event context.
<code>ix_cci_change_event()</code>	Changes the period of a periodic event.
<code>ix_cci_cc_add_message_handler()</code>	Adds a message handler to a core component and associates it with an input ID.
<code>ix_msg_handler()</code>	The function prototype for message-handler callback functions.
<code>ix_cci_cc_add_packet_handler()</code>	Adds a packet handler to a core component and associates it with an input ID.
<code>ix_pkt_handler()</code>	The function prototype for packet-handler callback functions.
<code>ix_cci_cc_create()</code>	Creates a core component and returns a component handle.
<code>ix_cc_init()</code>	The function prototype for a core component initialization function.
<code>ix_cc_fini()</code>	The function prototype for a core component termination function.
<code>ix_cci_cc_destroy()</code>	Destroys a core component specified by a handle to the component.
<code>ix_cci_cc_remove_event_handler()</code>	Removes an event created using <code>ix_cci_cc_add_event_handler()</code> or <code>ix_cci_cc_add_event_handler_ex()</code> .
<code>ix_cci_cc_remove_message_handler()</code>	Deletes a message handler.
<code>ix_cci_cc_remove_packet_handler()</code>	Deletes a packet handler.
<code>ix_cci_exe_add_policy()</code>	Adds a policy or policy tree to an execution engine.
<code>ix_cci_exe_get_info()</code>	Returns the execution-engine handle, engine number, and a context pointer associated with the execution engine from which the operation was invoked.
<code>ix_cci_exe_run()</code>	Runs the execution engine in a spawned task, thread, or process.
<code>ix_exe_init()</code>	The function prototype for the application-defined initialization function used when an execution engine is started.
<code>ix_exe_fini()</code>	The function prototype for the application-defined termination function used when an execution engine is shut down.

Table 4-1. cci API (Continued)

Name	Description
<code>ix_cci_exe_set_default()</code>	Sets an execution engine as the default engine whose information is returned by <code>ix_cci_exe_get_info()</code> when that function is called from a non-engine thread.
<code>ix_cci_exe_shutdown()</code>	Shuts down the execution engine identified by a handle, terminating its task, thread, or process.
<code>ix_cci_init()</code>	Initializes the framework for use by the core-component infrastructure.
<code>ix_cci_fini()</code>	Terminates the framework.
<code>ix_cci_policy_add_branch()</code>	Adds a branch—another policy or policy tree—to a scheduling policy. This operation supports construction of a hierarchical scheduling policy.
<code>ix_cci_policy_add_leaf()</code>	Adds a leaf node or input ID to a scheduling policy or execution engine.
<code>ix_cci_policy_create()</code>	Allocates a scheduling policy specifying the policy type—type is one of <i>round robin</i> , <i>weighted round robin</i> , or <i>priority</i> .
<code>ix_cci_policy_destroy()</code>	Frees the scheduling policy. Deallocates all resources associated with a tree.
<code>ix_cci_register_fatal_error_handler()</code>	Allows a control application to register a fatal-error handler.
<code>ix_ferror_func()</code>	This is the function prototype for the fatal-error handler callback provided by the calling application.
<code>ix_cci_send_message()</code>	Sends a message to a specific input of a core component.
<code>ix_cci_send_packet()</code>	Sends a packet to a specific input of a core component or to a microblock identified by microblock ID.

4.1 API Functions

4.1.1 `ix_cci_cc_add_event_handler()`

Adds an event handler to a core component. This function should be called within application core-component code.

Note: The execution engine only checks for events between the handling of messages and packets. Therefore, if a handler takes a long time to process a packet or message, the event triggers considerably later or less often than requested in the `arg_ms` parameter. The event function is called in the thread context of the execution engine. Event functions take strict priority over message and packet handlers.

C Syntax

```
ix_error ix_cci_cc_add_event_handler(
    ix_cc_handle arg_hComponent,
    ix_uint32 arg_ms,
    ix_event_func* arg_EventFunc,
    ix_event_type arg_EventType,
    ix_uint32 arg_Priority,
    ix_event_handle* arg_phEvent);
```


Input

<code>arg_hComponent</code>	A handle specifying the core component or execution engine to which to add an event handler.
<code>arg_ms</code>	The minimum number of milliseconds from the time the operation is invoked to the time at which the event will be triggered. Also the inter-event interval when <code>arg_EventType</code> is <code>IX_EVENT_TYPE_PERIODIC</code> .
<code>arg_EventFunc</code>	An event handler with the function signature specifying Section 4.1.2 , “ <code>ix_event_func()</code> .”
<code>arg_EventType</code>	Specifies the type of event. This value is one of: <ul style="list-style-type: none"> <code>IX_EVENT_TYPE_PERIODIC</code>—for periodic, recurring events <code>IX_EVENT_TYPE_ONESHOT</code>—for one-shot events
<code>arg_Priority</code>	The priority value the event scheduler uses to sequence events if two or more events are scheduled to occur at the same time.

Output/Returns

Return Value	Returns <code>IX_SUCCESS</code> if successful or an <code>ix_error</code> token encapsulating one of the following error codes: <ul style="list-style-type: none"> <code>IX_CCI_ERR_OS_MEM_ALLOC</code>—not enough memory for internal event structure <code>IX_CCI_ERR_CC_ADD_HANDLER_INVALID_HANDLE</code>—invalid handle passed in first argument <code>IX_CCI_ERR_EVENT_HANDLER_GET_TIME</code>—error getting system time <code>IX_CCI_ASSERTION</code>—invalid function parameter—debug compilations only
<code>arg_phEvent</code>	A pointer to an event handle for the newly added event handler—used for removing event handlers.

4.1.2 `ix_event_func()`

The function prototype specifying the signature for an event-handler function provided by the calling application.

C Syntax

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

Input

<code>arg_pContext</code>	The core-component context created by the calling-application implementation of the <code>ix_cc_init()</code> function passed to the core component creation function, <code>ix_cci_cc_create()</code> .
---------------------------	--

4.1.3 `ix_cci_cc_add_event_handler_ex()`

Similar to `ix_cci_cc_add_event_handler()` but instead of specifying the core component or engine context this function allows the calling application to specify the event context.

Note: The execution engine only checks for events between the handling of messages and packets. Therefore, if a handler takes a long time to process a packet or message, the event will be triggered considerably later (or less often) than requested in the `arg_ms` parameter. The event function is called in the thread context of the execution engine. Event functions take strict priority over message and packet handlers.

C Syntax

```
ix_error ix_cci_cc_add_event_handler(
    ix_cc_handle arg_hComponent,
    ix_uint32 arg_ms,
    ix_event_func* arg_EventFunc,
    ix_event_type arg_EventType,
    ix_uint32 arg_Priority,
    void* arg_EventContext,
    ix_event_handle* arg_phEvent );
```

Input

<code>arg_hComponent</code>	A handle to the core component.
<code>arg_ms</code>	The minimum number of milliseconds from the time the operation is invoked to the time at which the event will be triggered. Also the inter-event interval when <code>arg_EventType</code> is <code>IX_EVENT_TYPE_PERIODIC</code> .
<code>arg_EventFunc</code>	An event handler with function signature specified in Section 4.1.2 , “ <code>ix_event_func()</code> .”
<code>arg_EventType</code>	Specifies the type of event. This value is one of: <ul style="list-style-type: none"> <code>IX_EVENT_TYPE_PERIODIC</code>—for periodic, recurring events <code>IX_EVENT_TYPE_ONESHOT</code>—for one-shot events
<code>arg_Priority</code>	The priority value the event scheduler uses to sequence events if two or more events are scheduled to occur at the same time.
<code>arg_EventContext</code>	Context to be passed to the event handler specified by <code>arg_EventFunc</code> when the event occurs.

Output/Returns

<code>arg_phEvent</code>	A pointer to the event handle, used for removing event handlers.
Return Value	<p>Returns <code>IX_SUCCESS</code> if successful or an <code>ix_error</code> token encapsulating one of the following error codes:</p> <ul style="list-style-type: none"> <code>IX_CCI_ERR_OS_MEM_ALLOC</code>—there was not enough memory for the internal event structure <code>IX_CCI_ERR_CC_ADD_HANDLER_INVALID_HANDLE</code>—an invalid handle was passed in as the first argument <code>IX_CCI_ERR_EVENT_HANDLER_GET_TIME</code>—there was an error getting the system time <code>IX_CCI_ASSERTION</code>—an invalid function parameter was specified—applies to debug compilations only

4.1.4 `ix_cci_change_event()`

Changes the period of a periodic event.

C Syntax

```
ix_error ix_cci_change_event(
    ix_event_handle arg_hEvent, ix_uint32 arg_ms );
```

Input

<code>arg_hEvent</code>	A handle to the periodic event whose period is to be changed.
<code>arg_ms</code>	The new event period for the event specified by <code>arg_hEvent</code> .

Output/Returns

Return Value	<p>Returns <code>IX_SUCCESS</code> if successful or an <code>ix_error</code> token encapsulating the following error code:</p> <ul style="list-style-type: none"> <code>IX_CCI_ERROR_ASSERTION</code>—an invalid handle was passed in as the first argument or the event is not periodic—applies to debug compilations only
--------------	--

4.1.5 ix_cci_cc_add_message_handler()

Adds a message handler to a core component and associates it with an input ID. 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.

Note: This function should be called in the `ix_cci_init()` function for the core component or the `ix_exe_init()` function for the core component's execution engine.

C Syntax

```
ix_error ix_cci_cc_add_message_handler(
    ix_cc_handle arg_hComponent,
    ix_uint32 arg_InputID,
    ix_msg_handler* arg_Handler,
    ix_input_type arg_SourceType);
```

Input

<code>arg_hComponent</code>	A handle to the execution engine or a core component—used to gain access to the execution engine's default policy. If the execution engine is using its default policy tree, the message handler is automatically added to this policy tree. Otherwise, if a custom policy tree has been connected to the execution engine—using <code>ix_cci_exe_add_policy()</code> —it is the responsibility of the calling application to add the handler to the appropriate policy using the function <code>ix_cci_policy_add_leaf()</code> .
<code>arg_InputID</code>	Input ID to be associated with the handler.
<code>arg_Handler</code>	A pointer to a message handler provided by the calling application. The callback function prototype is specified in Section 4.1.5.1 , " <code>ix_msg_handler()</code> ."
<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>—messages are received from a single source <code>IX_INPUT_TYPE_MULTI_SRC</code>—messages are received from multiple sources <p>NOTE: The framework uses this type value to determine whether or not to employ locking when writing data into the queue corresponding to the input ID associated with this handler.</p>

Output

Return Value

Returns `IX_SUCCESS` if successful or an `ix_error` token encapsulating one of the following error codes:

- `IX_CCI_ERR_OS_MEM_ALLOC`—could not allocate enough memory for internal handler structure
- `IX_CCI_TOKEN_PROCESSOR_CONTAINER_FULL`—attempted to add the handler to a full policy that cannot grow
- `IX_CCI_ERR_TKP_CONTAINER_MEM_ALLOC`—a memory allocation error occurred when attempting to add the handler to a full policy
- `IX_CCI_ERR_MSG_SET_MODE`—an error was reported by the Resource Manager when setting the handler's mode of operation
- `IX_CCI_ASSERTION`—invalid function parameter—debug compilations only

4.1.5.1 `ix_msg_handler()`

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

C Syntax

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

Input

<code>arg_hDataToken</code>	Calling application specific data.
<code>arg_UserData</code>	Calling application specific data.
<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> .

Output/Returns

Return Value	Returns <code>IX_SUCCESS</code> or a valid <code>ix_error</code> .
--------------	--

4.1.6 ix_cci_cc_add_packet_handler()

Adds a packet handler to a core component and associates it with an input ID. 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 is already associated with a handler returns an error.

Note: This function should be called in the `ix_cci_init()` for the core component or the `ix_exe_init()` function of the core component's execution engine.

C Syntax

```
ix_error ix_cci_cc_add_packet_handler(
    ix_cc_handle arg_hComponent,
    ix_uint32 arg_InputID,
    ix_pkt_handler arg_Handler,
    ix_input_type arg_SourceType );
```

Input

<code>arg_hComponent</code>	A handle to the execution engine or a core component—used to gain access to the execution engine's default policy. If the execution engine is using its default policy tree, the packet handler is automatically added to this policy tree. Otherwise, if a custom policy tree has been connected to the execution engine using <code>ix_cci_exe_add_policy()</code> it is the responsibility of the calling application to add the handler to the appropriate policy using the function <code>ix_cci_policy_add_leaf()</code> .
<code>arg_Handler</code>	Pointer to the packet handler with the signature specified in Section 4.1.6.1, “ix_pkt_handler()” .
<code>arg_InputID</code>	The input ID to be associated with the handler.
<code>arg_SourceType</code>	Indicates whether the input receives messages from one source or multiple sources. Valid values are: <code>IX_INPUT_TYPE_SINGLE_SRC</code> and <code>IX_INPUT_TYPE_MULTI_SRC</code> . NOTE: The framework uses this type value to determine whether or not to employ locking when writing data into the queue corresponding to the input ID associated with this handler.

Output

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

4.1.6.1 ix_pkt_handler()

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

C Syntax

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

Input

<code>arg_hDataToken</code>	A handler to calling-application defined data.
<code>arg_ExceptionCode</code>	A calling-application defined exception code.
<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> .

Output/Returns

Return Value	Returns <code>IX_SUCCESS</code> or a valid <code>ix_error</code> .
--------------	--

4.1.7 `ix_cci_cc_create()`

Creates a core component. This function should be called from the application-defined `ix_exe_init()` function in the core component's execution engine.

C Syntax

```
ix_error ix_cci_cc_create(
    ix_exe_handle arg_EngHandle,
    ix_cc_init arg_InitFunc,
    ix_cc_fini arg_FiniFunc,
    void* arg_pContextIn,
    ix_cc_handle* arg_phComponent);
```

Input

<code>arg_EngHandle</code>	A handle to the execution engine where the core component code is to be run.
<code>arg_InitFunc</code>	<p>A pointer to the core component initialization function. Application code may use this function to create resources shared between the core and any microblocks with which the component interacts. It may also patch symbols to those microblocks.</p> <p>Implementation of this function is an application responsibility. The function signature is described in Section 4.1.7.1, “<code>ix_cc_init()</code>.”</p> <p>The <code>ppContext</code> parameter points to an input/output context. The context pointer is stored in the component and passed to the function specified by <code>arg_FiniFunc</code> when the component is destroyed. The context is also passed to any events, message handlers, or packet handlers added to the core component.</p>
<code>arg_FiniFunc</code>	<p>A pointer to the component termination function.</p> <p>The function signature is shown in Section 4.1.7.2, “<code>ix_cc_fini()</code>.”</p>
<code>arg_pContextIn</code>	A pointer to a context that is passed into <code>arg_InitFunc</code> .

Output

Return Value	<p>Returns <code>IX_SUCCESS</code> if successful or an <code>ix_error</code> token encapsulating one of the following error codes:</p> <ul style="list-style-type: none"> <code>IX_CCI_ERR_OS_MEM_ALLOC</code>—there was not enough memory to complete the operation <code>IX_CCI_ASSERTION</code>—an invalid function parameter was passed in—applies to debug compilations only Any errors returned by the <code>ix_cc_init()</code> callback function
<code>arg_phComponent</code>	A pointer to the location for return of the core-component handle.

4.1.7.1 `ix_cc_init()`

This is the function prototype for a component initialization function.

C Syntax

```
ix_error (*ix_cc_init)( void** ppContext );
```

Input/Output

<code>ppContext</code>	A pointer to any context convenient for use by the core component. As an input, it may be some global context; as an output it may be a pointer to resources allocated within the initialization function.
------------------------	--

Output/Returns

Return Value	Returns <code>IX_SUCCESS</code> if successful or an <code>ix_error</code> token.
--------------	--

4.1.7.2 `ix_cc_fini()`

This is the function prototype for a component termination function.

C Syntax

```
ix_error (*ix_cc_fini)( void* pContext );
```

Input

<code>pContext</code>	The context output by core-component initialization function—see Section 4.1.7.1, “<code>ix_cc_init()</code>.”
-----------------------	--

Output/Returns

Return Value	Returns <code>IX_SUCCESS</code> if successful or an <code>ix_error</code> token.
--------------	--

4.1.8 `ix_cci_cc_destroy()`

Destroys the component. The function performs internal cleanup and invokes the application-defined `ix_cc_fini()` function associated with the component. This function should be called from the application-defined `ix_exe_fini()` function for the core component's execution engine.

C Syntax

```
ix_error ix_cci_cc_destroy( ix_cc_handle arg_hComponent);
```

Input

`arg_hComponent` A handle to the core component originally initialized using `ix_cci_cc_create()`.

Output

`Return Value` Returns `IX_SUCCESS` if successful or an `ix_error` token encapsulating one of the following error codes:

- Any errors returned by the `ix_cc_fini()` callback function
- `IX_CCI_ASSERTION`—invalid function parameter—debug compilations only

4.1.9 `ix_cci_cc_remove_event_handler()`

Removes an event handler from a core component. This function should be called within application core-component code and is only required for removing periodic events and unexpired one-shot events.

Note: It is not necessary to remove a one-shot event handler after it has expired. Although this function checks that the input parameters are valid, attempting to remove an expired event does not return an error.

C Syntax

```
ix_error ix_cci_cc_remove_event_handler(ix_event_handle arg_hEvent);
```

Input

`arg_hEvent` A handle to the event to be removed.

Output

Return Value Returns `IX_SUCCESS` if successful or an `ix_error` token encapsulating one of the following error codes:

- `IX_CCI_ASSERTION`—an invalid function parameter was specified—applies to debug compilations only
- `IX_CCI_ERR_EVENT_NOT_FOUND`—no trace of event found, even on expired-event list—debug compilations only

4.1.10 `ix_cci_cc_remove_message_handler()`

Removes a message handler previously added using `ix_cci_cc_add_message_handler()`. This function may be called from the application-defined termination function of the core component's execution engine, or from the core component's own termination function.

C Syntax

```
ix_error ix_cci_cc_remove_message_handler(ix_uint32 arg_InputID);
```

Input

<code>arg_InputID</code>	The input ID associated with the message handler to be removed. After calling this function, any messages arriving on this input ID are dropped.
--------------------------	--

Output

Return Value	Returns <code>IX_SUCCESS</code> if successful or an <code>ix_error</code> token encapsulating one of the following error codes: <ul style="list-style-type: none">• <code>IX_CCI_ERR_DISABLE_MSG_HANDLER</code>—error disabling message handler in Resource Manager• <code>IX_CCI_ASSERTION</code>—invalid function parameter—debug compilations only
--------------	--

4.1.11 `ix_cci_cc_remove_packet_handler()`

Removes a packet handler previously added using `ix_cci_cc_add_packet_handler()`. This function may be called from the application-defined termination function of the core component's execution engine, or from the core component's own termination function.

C Syntax

```
ix_error ix_cci_cc_remove_packet_handler(ix_uint32 arg_InputID );
```

Input

<code>arg_InputID</code>	The input ID where packet handler is to be removed. After calling this function, any packets arriving on this input ID are dropped.
--------------------------	---

Output/Returns

Return Value	<p>Returns <code>IX_SUCCESS</code> if successful or an <code>ix_error</code> token encapsulating one of the following error codes:</p> <ul style="list-style-type: none"> • <code>IX_CCI_ERR_DISABLE_MSG_HANDLER</code>—error disabling message handler in Resource Manager • <code>IX_CCI_ASSERTION</code>—invalid function parameter—debug compilations only
--------------	--

4.1.12 `ix_cci_exe_add_policy()`

Adds a branch node—a scheduling policy—to an execution engine. This function should be called from the application-defined `ix_exe_init()` function for the execution engine in order to replace the execution engine's default policy tree with a different scheduling policy or policy tree.

Note: This function, in conjunction with `ix_cci_policy_add_branch()` and `ix_cci_policy_add_leaf()`, should only be called to replace the default policy tree with a different policy or policy tree. All message and packet handlers must be added to the execution engine's core components before calling this function. Failure to do so results in an error being returned from the add-handler function. Care should be taken that all IDs are added to the new policy tree using `ix_cci_policy_add_leaf()`—otherwise the handlers do not run when packets or messages arrive on those IDs.

C Syntax

```
ix_error ix_cci_exe_add_policy(
    ix_exe_handle arg_hParent,
    ix_policy_handle arg_hChild,
    ix_uint32 arg_NumCredits);
```


Input

<code>arg_hParent</code>	The execution engine to which to add the policy.
<code>arg_hChild</code>	A handle to the policy to add.
<code>arg_NumCredits</code>	The number of times a handler is called before another handler is selected by the policy tree—this assumes the handler’s queue has enough entries and that no new messages or packets are received with a higher priority.

Output

Return Value	Returns <code>IX_SUCCESS</code> if successful or an <code>ix_error</code> token encapsulating the following error code: <ul style="list-style-type: none"> <code>IX_CCI_ASSERTION</code>—invalid function parameter—debug compilations only
--------------	--

4.1.13 `ix_cci_exe_get_info()`

Returns the execution-engine handle, engine number, and a context pointer associated with the execution engine in which the caller is running. If this function is called from a non-engine thread, it returns information about the default engine. The default engine is set by the function `ix_cci_exe_set_default()`—if this function has not been called, the default engine is engine zero.

C Syntax

```
ix_error ix_cci_get_info(
    ix_exe_handle* arg_pExeHandle,
    ix_uint32* arg_pEngineNumber,
    void* arg_ppContext);
```

Output/Returns

<code>arg_pExeHandle</code>	A handle to the engine running this thread.
<code>arg_pEngineNumber</code>	A unique engine number. The first engine created is engine number zero, the next is engine number one, and so on.
<code>arg_ppContext</code>	A pointer to the engine context initialized by the application-defined <code>ix_cc_init()</code> function passed into <code>ix_cci_exe_run()</code> when the engine is created.
Return Value	Returns <code>IX_SUCCESS</code> if successful or an <code>ix_error</code> token encapsulating the following error code: <ul style="list-style-type: none"> <code>IX_CCI_ERR_ENG_THREAD_ID</code>—error getting current thread ID or default engine no longer exists

4.1.14 ix_cci_exe_run()

Creates and runs an execution engine—that is, a thread of control. This function should be called from application code after `ix_cci_init()` has been called to initialize the core framework.

Note: This function calls the application-defined `ix_exe_init()` function associated with this engine and stores the output context internally so that it can be passed to the application-defined `ix_exe_fini()` function for this engine when `ix_cci_exe_shutdown()` is called.

C Syntax

```
ix_error ix_cci_exe_run(
    const char* arg_FileName,
    ix_exe_init_func arg_InitFunc,
    ix_exe_fini_func arg_FiniFunc,
    const char* arg_Name,
    ix_exe_handle* arg_pHandle);
```

Input

<code>arg_FileName</code>	If the operating environment is process-based and is required to spawn a new process for each execution engine, this parameter specifies the filename where the execution engine's process entry point is defined. Otherwise, the parameter should be set to <code>NULL</code> .
<code>arg_InitFunc</code>	Specifies a application-defined initialization function that is called by <code>ix_cci_exe_run()</code> . This function is responsible for creating the execution-control tree and setting up the message and packet handlers. If the operating environment is process-based and is required to spawn a new process for each execution engine, this parameter should be set to <code>NULL</code> , and the initialization should be performed by the process' entry function. The function signature is specified in Section 4.1.14.1 , " <code>ix_exe_init()</code> ." The memory for the context passed as an argument into <code>ix_exe_init()</code> must be dynamically allocated by the initialization function. The initialization function must return this context pointer using the same argument so that this pointer can be stored by the execution engine. This context pointer is passed to the execution engine's <code>ix_exe_fini()</code> function at shutdown. The <code>ix_exe_fini()</code> function uses this context to gain access to any core components and policies that the execution engine must delete.
<code>arg_FiniFunc</code>	Specifies a application-defined cleanup function that is called in <code>ix_cci_exe_run()</code> as a result of <code>ix_cci_exe_shutdown()</code> being called in a control thread. After the cleanup function has returned the <code>ix_cci_exe_run()</code> function terminates. If the operating environment is process-based and is required to spawn a new process for each execution engine, this parameter should be set to <code>NULL</code> , and the engine's <code>ix_exe_fini()</code> pointer should be initialized by the process' entry function. The function signature is specified in Section 4.1.14.2 , " <code>ix_exe_fini()</code> ."

Input (Continued)

`arg_Name` Name to be associated with the current execution engine.

Output

`Return Value` Returns `IX_SUCCESS` if successful or an `ix_error` token encapsulating one of the following error codes:

- `IX_CCI_ERR_ENG_INIT_SEMA_PROP_READ`—could not read registry properties for the global synchronization semaphore
- `IX_CCI_ERR_ENG_PROP_CLOSE`—there was an error closing the property for global semaphore
- `IX_CCI_ERR_ENG_INIT_ENGINE_PROP_READ`—could not read properties for shared-memory integers used by all execution engines
- `IX_CCI_ERR_ENG_PROP_CLOSE`—there was an error closing the property for shared-memory integers
- `IX_CCI_ERR_ENG_THREAD_SPAWN`—there was an error spawning the engine thread
- `IX_CCI_ERR_OS_MEM_ALLOC`—there was a memory allocation error
- `IX_CCI_ERR_TKP_CONTAINER_MEM_ALLOC`—there was a memory allocation error
- `IX_CCI_ERR_ENG_THREAD_ID`—could not read the current thread ID
- `IX_CCI_ERR_SHARED_MEM_ALLOC`—there was a shared-memory allocation error
- `IX_CCI_ERR_ENG_INIT_SEMA_INIT`—could not initialize this engine's semaphore
- `IX_CCI_NO_SELECTED_CHILD`—no event/message/packet handler was added—either directly or via one of its core components—to the engine
- `IX_CCI_ASSERTION`—invalid function parameter was passed in—applies to debug compilations only
- Errors returned from `ix_cci_policy_create()`, `ix_cci_policy_add_branch()`, and `ix_cci_exe_add_policy()` while trying to create the default policy tree
- Errors returned from the callback function whose prototype is `ix_cc_init()`

`arg_pHandle` A pointer to the new execution engine's handle.

4.1.14.1 `ix_exe_init()`

The function prototype for the application-defined initialization function used when an execution engine is started.

C Syntax

```
ix_error (*ix_exe_init)(  
    ix_exe_handle ExeHandle,  
    void** ppContext);
```

Input

ExeHandle	A handle assigned to this execution engine. If this handle is saved in the execution engine's context, then when the engine is running, information about the engine can be retrieved using the function <code>ix_cci_exe_get_info()</code> .
ppContext	The memory for the context pointed to by this argument must be dynamically allocated by the initialization function. The initialization function must return this context pointer using <code>ppContext</code> so that this pointer can be stored by the execution engine.

Output/Returns

Return Value	Returns <code>IX_SUCCESS</code> if successful or an <code>ix_error</code> token.
--------------	--

4.1.14.2 `ix_exe_fini()`

The function prototype for the application-defined finalization function used when an execution engine is shut down.

C Syntax

```
ix_error (*ix_exe_fini)(
    ix_exe_handle ExeHandle,
    void* pContext );
```

Input

ExeHandle	Handle assigned to this execution engine. If this handle is saved in the execution engine's context, then when the engine is running, information about the engine can be retrieved using the function <code>ix_cci_exe_get_info()</code> .
pContext	A pointer to the context created by and passed back from the matching <code>ix_exe_init()</code> function. This context provides access to any core components and policies that the execution engine must delete.

Output/Returns

Return Value	Returns <code>IX_SUCCESS</code> if successful or an <code>ix_error</code> token.
--------------	--

4.1.15 `ix_cci_exe_set_default()`

Sets an engine as the default engine whose information is returned by `ix_cci_exe_get_info()` when that function is called from a non-engine thread. If this function is never called, the default engine is the first engine created after `ix_cci_init()`.

C Syntax

```
ix_error ix_cci_exe_set_default(ix_exe_handle arg_Handle);
```

Input

<code>arg_Handle</code>	A handle to the execution engine to set as the default.
-------------------------	---

Output/Returns

Return Value	Returns <code>IX_SUCCESS</code> if successful or an <code>ix_error</code> token encapsulating the following error code: <ul style="list-style-type: none"><code>IX_CCI_ASSERTION</code>—invalid function parameter—debug compilations only
--------------	--

4.1.16 `ix_cci_exe_shutdown()`

Sets a flag in the execution engine, telling it to shut down. Each execution engine contains a semaphore that the engine unlocks when it has finished processing and has executed the execution engine application-defined function specified by the `ix_cc_fini()` function prototype. This shutdown function blocks until the execution engine indicates shutdown is complete. All applications should call this function to shut down the application.

C Syntax

```
ix_error ix_cci_exe_shutdown( ix_exe_handle arg_Handle);
```

Input

<code>arg_Handle</code>	Handle to execution engine originally initialized by <code>ix_cci_exe_run()</code> .
-------------------------	--

Output

Return Value	Returns <code>IX_SUCCESS</code> if successful or an <code>ix_error</code> token encapsulating one of the following error codes: <ul style="list-style-type: none">• <code>IX_CCI_ASSERTION</code>—invalid function parameter—debug compilations only• Fatal errors returned by event, message, or packet handlers• Errors returned by callback functions with an <code>ix_cc_fini()</code> function prototype
--------------	---

4.1.17 `ix_cci_init()`

Initializes the core component infrastructure. This function should be called from application code before calling any other functions in the core-component infrastructure.

Note: The function `ix_rm_init()` must be called before calling this function.

C Syntax

```
ix_error ix_cci_init( void );
```

Output

Return Value	Returns <code>IX_SUCCESS</code> if successful or an <code>ix_error</code> token encapsulating one of the following error codes: <ul style="list-style-type: none">• <code>IX_CCI_ERR_INIT_ERR_LOG</code>—unable to initialize error-message log• <code>IX_CCI_ERR_SHARED_MEM_ALLOC</code>—unable to allocate shared memory needed by the Core Component Infrastructure• <code>IX_CCI_ERR_GLOB_SEMA_INIT</code>—unable to initialize the global Core Component Infrastructure semaphore• <code>IX_CCI_ERR_PROPERTY_CREATE</code>—error creating Resource Manager properties• <code>IX_CCI_ERR_PROPERTY_SET</code>—error setting a property
--------------	---

4.1.18 `ix_cci_fini()`

Terminates the core component infrastructure. This function should be called in application code after all execution engines have been shut down and destroyed.

Note: The function `ix_cci_fini()` should be called before calling `ix_rm_term()`.

C Syntax

```
ix_error ix_cci_fini( void );
```

Output

Return Value	Returns <code>IX_SUCCESS</code> if successful or a an <code>ix_error</code> token returned from the Resource Manager or Operating System Services Layer when trying to free a resource.
--------------	---

4.1.19 `ix_cci_policy_add_branch()`

Adds a branch node—a scheduling policy—to another scheduling policy. Up to 32 nodes, that is, branches and leaves, can be added to an individual policy. This function should be called from the application-defined `ix_exe_init()` function of an execution engine on an existing policy created using `ix_cci_policy_create()`.

C Syntax

```
ix_error ix_cci_policy_add_branch(
    ix_policy_handle arg_hPolicy,
    ix_policy_handle arg_hChild,
    ix_uint32 arg_WeightOrPriority);
```

Input

<code>arg_hPolicy</code>	A handle to the scheduling policy to which the branch is added.
<code>arg_hChild</code>	A handle to the branch—or policy—to add.
<code>arg_WeightOrPriority</code>	Weight if the parent is a weighted round robin policy or priority if the parent is a strict priority policy. This parameter is ignored if the parent is a round robin policy.

Output

Return Value	<p>Returns <code>IX_SUCCESS</code> if successful or an <code>ix_error</code> token encapsulating one of the following error codes:</p> <ul style="list-style-type: none"> <code>IX_CCI_TOKEN_PROCESSOR_CONTAINER_FULL</code>—the parent policy is full and was initialized to be fixed in size <code>IX_CCI_ERR_TKP_CONTAINER_MEM_ALLOC</code>—there was a memory allocation error <code>IX_CCI_ASSERTION</code>—an invalid function parameter was passed in—applies to debug compilations only
--------------	--

4.1.20 `ix_cci_policy_add_leaf()`

Adds a leaf node—packet or message input node—to a scheduling policy. Up to 32 nodes—branches and leaves—can be added to an individual policy. This function should be called from the application-defined `ix_exe_init()` function of an execution engine on an existing policy, created using `ix_cci_policy_create()`.

C Syntax

```
ix_error ix_cci_policy_add_leaf(
    ix_policy_handle arg_hPolicy,
    ix_uint32 arg_InputId,
    ix_uint32 arg_IsPacketId,
    ix_uint32 arg_WeightOrPriority);
```

Input

<code>arg_hPolicy</code>	A handle to the scheduling policy to which to add a leaf.
<code>arg_InputId</code>	The packet or message input ID.
<code>arg_IsPacketId</code>	Indicates if the leaf is a packet input node. Used as a boolean: <ul style="list-style-type: none"> 1—for packet-input ID 0—for not a packet-input ID—that is, the node is a message-input ID
<code>arg_WeightOrPriority</code>	A weight if parent is a weighted round robin policy or a priority if parent is a strict priority policy. This parameter is ignored if the parent is a round robin policy.

Output

Return Value	Returns <code>IX_SUCCESS</code> if successful or an <code>ix_error</code> token encapsulating one of the following error codes: <ul style="list-style-type: none"> <code>IX_CCI_TOKEN_PROCESSOR_CONTAINER_FULL</code>—the parent policy is full and was initialized to be fixed in size <code>IX_CCI_ERR_TKP_CONTAINER_MEM_ALLOC</code>—there was a memory allocation error <code>IX_CCI_ASSERTION</code>—an invalid function parameter was passed in—applies to debug compilations only
--------------	---

4.1.21 `ix_cci_policy_create()`

Creates a scheduling policy. This function should be called in the application-defined `ix_exe_init()` function of an execution engine.

C Syntax

```
ix_error ix_cci_policy_create(
    ix_policy_type argPolicyType,
    ix_policy_handle* arg_pHandle );
```

Input

`argPolicyType`

The policy type—valid values defined by the infrastructure include:

- `IX_POLICY_TYPE_RR`
- `IX_POLICY_TYPE_WRR`
- `IX_POLICY_TYPE_SP`

Output

Return Value

Returns `IX_SUCCESS` if successful or an `ix_error` token encapsulating one of the following error codes:

- `IX_CCI_ERR_POLICY_CREATE`—there was a memory allocation error
- `IX_CCI_ASSERTION`—an invalid function parameter was passed in—applies to debug compilations only

`arg_pHandle`

A pointer to a handle to the new policy.

4.1.22 ix_cci_policy_destroy()

Destroys a scheduling policy. This function should be called from the application-defined `ix_exe_fini()` function of an execution engine.

Note: This function does not destroy any subtree policies. The application code is responsible for destroying all policies that have been created. The core component infrastructure ensures that notifications are dropped gracefully from a tree that is in the process of being dismantled.

C Syntax

```
ix_error ix_cci_policy_destroy( ix_policy_handle arg_Handle );
```

Input

<code>arg_Handle</code>	A handle to the policy to destroy—the policy is invalid after this function returns.
-------------------------	--

Output

Return Value	Returns <code>IX_SUCCESS</code> if successful or an <code>ix_error</code> token encapsulating the following error code: <ul style="list-style-type: none"><code>IX_CCI_ASSERTION</code>—an invalid function parameter was passed in—applies to debug compilations only
--------------	--

4.1.23 `ix_cci_register_fatal_error_handler()`

Allows a control application to register a fatal-error handler. The Core Component Infrastructure calls this handler if an event, message, or packet handler returns a fatal error. This allows the control application to disable any microcode associated with this engine and shut the engine or whole Core Component Infrastructure system down. This function is optional. If used, it must be invoked after `ix_cci_init()`.

C Syntax

```
ix_error ix_cci_register_fatal_error_handler(
    ix_ferror_func arg_Handler,
    void* arg_pContext);
```

Input

<code>arg_Handler</code>	<p>A fatal error handler that is invoked whenever a fatal error occurs in the Core Component Infrastructure.</p> <p>The function signature is:</p> <pre>void (* ix_ferror_func)(ix_error arg_FatalException, ix_exe_handle arg_hEngine, void* arg_pContext);</pre> <p>The <code>arg_FatalException</code> argument reports the fatal error detected by the Core Component Infrastructure. The <code>arg_hEngine</code> argument specifies the execution engine where the error occurred. The <code>arg_pContext</code> argument is the context registered by <code>ix_cci_register_fatal_error_handler</code>.</p>
<code>arg_pContext</code>	The context to return when the handler is invoked.

Output/Returns

Return Value	<p>Returns <code>IX_SUCCESS</code> if successful or an <code>ix_error</code> token encapsulating one of the following error codes:</p> <ul style="list-style-type: none"> <code>IX_CCI_ERR_ENG_INIT_ENGINE_PROP_READ</code>—error reading shared-memory property <code>IX_CCI_ASSERTION</code>—invalid function parameter—debug compilations only
--------------	---

4.1.23.1 `ix_ferror_func()`

This is the function prototype for the fatal-error handler callback provided by the calling application.

C Syntax

```
void (* ix_ferror_func)(  
    ix_error arg_FatalException,  
    ix_exe_handle arg_hEngine,  
    void* arg_pContext);
```

Input

<code>arg_FatalException</code>	Reports the fatal error detected by the Core Component Infrastructure.
<code>arg_hEngine</code>	Specifies the execution engine where the error occurred.
<code>arg_pContext</code>	The context registered by <code>ix_cci_register_fatal_error_handler()</code> .

Output/Returns

Return Value	Returns <code>IX_SUCCESS</code> if successful or an <code>ix_error</code> token.
--------------	--

4.1.24 ix_cci_send_message()

Sends a message from a core component's packet or message handler to another core component. The application may call this function at any time after the core framework has been initialized, but it is typically called from within a core component's packet or message handler to send a message to another core component or to itself. The infrastructure also calls this function, or an internal equivalent, when a microblock sends a message to a core component. If this function is called to send a message to an ID that does not have a handler attached, then there is a call to a default message handler which simply drops the message and frees the message's shared-memory resources.

Note: This function encapsulates `ix_rm_message_send()` with an exception code of zero. Messages can be sent to core components and microblocks using either API.

C Syntax

```
ix_error ix_cci_send_message(  
    ix_uint32 arg_OutputId,  
    ix_buffer_handle arg_Handle,  
    ix_uint32 arg_UserData);
```

Input

<code>arg_OutputId</code>	The output ID identifying the message destination.
<code>arg_Handle</code>	A handle to the message to be sent.
<code>arg_UserData</code>	Application-defined content that can be used by the receiving message handler for any purpose—for example, identifying the message type.

Output

Return Value	Returns <code>IX_SUCCESS</code> if successful or a valid <code>ix_error</code> token retrieved from <code>ix_rm_message_send()</code> .
--------------	---

4.1.25 `ix_cci_send_packet()`

Sends a packet from a core component's packet/message handler to another core component or microblock. The calling application may invoke this function at any time after the core framework has been initialized. Typically this operation is called from within a core component's packet handler to forward a packet to another core component or even to itself. The infrastructure also calls this function—or an internal equivalent—when a microblock sends a packet to a core component. If this function is called to send a packet to an ID that does not have a handler attached, then there is a call to a default packet handler which simply drops the packet and frees the packet's shared-memory resources.

Note: This function encapsulates `ix_rm_packet_send()` with an exception code of zero. Packets can be sent to core components and microblocks using either API.

C Syntax

```
ix_error ix_cci_send_packet(  
    ix_uint32 arg_OutputId,  
    ix_buffer_handle arg_Handle );
```

Input

<code>arg_OutputId</code>	The output ID—the packet destination.
<code>arg_Handle</code>	A handle to the packet to be sent.

Output

Return Value	Returns <code>IX_SUCCESS</code> if successful or a valid <code>ix_error</code> type retrieved from <code>ix_rm_packet_send()</code> .
--------------	---

4.2 Symbolic Constants—Tuning Behavior and Memory Footprint

The symbolic constants defined in this section allow the system designer to tune the behavior and memory footprint of the Core Component Infrastructure.

IX_CCI_EVENT_CHECK_INTERVAL

The default frequency at which execution engines check for events and shutdown flag. The check interval is in units of milliseconds. If an event is set that is of shorter duration than this interval, the engine semaphore's timeout uses the event timeout instead. Therefore, events can be handled even if this value is set to wait forever. However, if the value is set to wait forever, it might not be possible to shut down the execution engines.

Note: Do not set this value to zero.

C Syntax and Default

```
#define IX_CCI_EVENT_CHECK_INTERVAL 1000
```

IX_CCI_MEMORY_CHANNEL

The channel number for Core Component Infrastructure memory.

C Syntax and Default

```
#define IX_CCI_MEMORY_CHANNEL IX_DRAM_CHANNELS_NUMBER - 1
```

IX_CCI_SHARED_MEMORY_TYPE

The type of memory used for Core Component Infrastructure shared memory.

C Syntax and Default

```
#define IX_CCI_SHARED_MEMORY_TYPE IX_MEMORY_TYPE_DRAM
```

IX_CCI_MAX_INPUT_IDS

Defines the maximum number of input IDs.

C Syntax and Default

```
#define IX_CCI_MAX_INPUT_IDS IX_COMM_LOCAL_ID_NUMBER - \
IX_COMM_LAST_UBLOCK_ID - 1
```


[IX_CCI_MAX_ENGINES](#)

The maximum number of engines on an ingress or egress board. This value does not need to be a power of two. It should not be greater than 64 in the current implementation.

C Syntax and Default

```
#define IX_CCI_MAX_ENGINES 64
```

[IX_CCI_MAX_POLICIES](#)

The maximum number of policies on an ingress or egress board. This value does not need to be a power of two. It should not be greater than 512 in the current implementation.

C Syntax and Default

```
#define IX_CCI_MAX_POLICIES 256
```

[IX_CCI_MAX_CCS](#)

The maximum number of core components in an engine. This value does not need to be a power of two. It should not be greater than 256 in the current implementation.

C Syntax and Default

```
#define IX_CCI_MAX_CCS 64
```

[IX_CCI_MAX_EVENTS](#)

The maximum number of events in an engine. This value does not need to be a power of two. It should not be greater than 2,000,000 in the current implementation.

C Syntax and Default

```
#define IX_CCI_MAX_EVENTS 256
```

[IX_CCI_NUM_DEFAULT_ENGINES](#)

The number of default engines associated with non-engine threads. The code presently only allows for one default engine.

C Syntax and Default

```
#define IX_CCI_NUM_DEFAULT_ENGINES 1
```


IX_CCI_EXE_RUN_GRANULARITY

The number of handlers the executable engine runs before passing control to the next execution engine.

C Syntax and Default

```
#if !defined(IX_CCI_EXE_RUN_GRANULARITY)
#define IX_CCI_EXE_RUN_GRANULARITY      4
#endif
```

Policy Container Capacity Constants

The following constants determines the initial capacity of the policy container. Policy containers can be grown in size, so if most execution engines have few handlers associated with them, this number can be set to a low value to save memory usage.

C Syntax and Default

```
#define IX_CCI_INIT_WRR_POLICY_SIZE      1
#define IX_CCI_INIT_RR_POLICY_SIZE      1
#define IX_CCI_INIT_SP_POLICY_SIZE      2
#define IX_CCI_WRR_POLICY_GROW_INCR     1
#define IX_CCI_RR_POLICY_GROW_INCR      1
#define IX_CCI_SP_POLICY_GROW_INCR      1
```


TCAM Lookup Libraries

5

This chapter describes a common API used for managing and searching tables on the Intel XScale® core and on the microengines for Intel® IXP2400 and IXP2800 Network Processors.

The lookup library provides a way of managing different search and lookup tables that can be used for many different networking applications. The goal of the search table is to hide the details of both the data structures and the underlying hardware implementation from the application designer. This abstraction allows the application to remain unchanged, while different data structures are added or hardware assisted search devices are used, such as TCAM (Ternary Content Addressable Memory)..

Table 5-1. TCAM Lookup Library

Name	Description
<code>ix_lkup</code>	Handle that is returned when the application first initializes and gets a handle to the lookup library.
<code>ix_lkup_table</code>	Handle that is returned when a new table is created by calling <code>IX_LKUP_CREATE_TABLE()</code> on a valid <code>ix_lkup</code> .
<code>ix_lkup_table_type</code>	Defines the different table types and the associated search methods for that table.
<code>ix_lkup_tcam_params</code>	Data structure that is passed when initializing the TCAM version of the library by calling <code>ix_lkup_tcam_init()</code> .
<code>ix_lkup_table_conf</code>	Data structure used to pass the configuration parameters when a new table is created.
<code>ix_lkup_cookie</code>	Opaque cookie that is passed between some of the API calls, primarily calls that are used for enumerating the contents of a table.
<code>ix_lkup_sw_init()</code>	Initializes the software lookup management library and gets a handle to the library for subsequent operations.
<code>ix_lkup_tcam_init()</code>	Initializes the TCAM lookup management library and returns a handle to the library for subsequent operations.
<code>IX_LKUP_CREATE_TABLE()</code>	Creates a new instance of a search table.
<code>IX_LKUP_DESTROY_TABLE()</code>	Destroys a table that was created earlier with <code>IX_LKUP_CREATE_TABLE</code> .
<code>IX_LKUP_FINI()</code>	Destroys a previously obtained <code>ix_lkup</code> handle.
<code>IX_LKUP_ADD_ENTRY()</code>	Adds an entry to a table.
<code>IX_LKUP_REMOVE_ENTRY()</code>	Removes an entry from a table.
<code>IX_LKUP_UPDATE_ENTRY()</code>	Updates the data associated with an element already in the table.
<code>IX_LKUP_SEARCH_TABLE()</code>	Searches a specific table and returns the associated data if a match is found.
<code>IX_LKUP_FIND_ENTRY()</code>	Searches a table for an entry that matches the exact key and mask, weight combination.
<code>IX_LKUP_READ_FIRST_ENTRY()</code>	Retrieves the first item stored in the table.
<code>IX_LKUP_READ_NEXT_ENTRY()</code>	Retrieves successive items stored in the table.
<code>IX_LKUP_RESET_TABLE()</code>	Clears all the items in the table and resets it to its initial state.
<code>IX_LKUP_SET_PROPERTY()</code>	Allows the caller to set special attributes of the table.
<code>IX_LKUP_GET_PROPERTY()</code>	Allows the caller to get special attributes of the table.

Table 5-1. TCAM Lookup Library (Continued)

Name	Description
<code>IX_LKUP_GET_TABLE_INFO()</code>	Returns table identifier and data information.
<code>ix_tcam_lkup_build_handle()</code>	Builds the handle that must be passed to the search functions.
<code>ix_tcam_lkup_start()</code>	Starts a search using the <code>in_key</code> to launch the search request.
<code>ix_tcam_lkup_complete()</code>	Completes a search that was started and returns the results.
<code>ix_tcam_lkup_get_data()</code>	Returns the data associated with a successful search.
<code>ix_sw_lkup_lpm_build_handle()</code>	Builds the handle that must be passed to the search functions for longest prefix match searching.
<code>ix_sw_lkup_lpm_search()</code>	Searches a longest prefix match table and returns the results.
<code>ix_sw_lkup_exact_build_handle()</code>	Builds the handle that must be passed to the search functions for exact match searching.
<code>ix_sw_lkup_exact_search()</code>	Searches an exact match table and returns the results.
<code>ix_sw_lkup_range_build_handle()</code>	Builds the handle that must be passed to the search functions for range match searching.
<code>ix_sw_lkup_range_search()</code>	Searches a range match table and returns the results.
<code>ix_s_lkup</code>	Data structure that all implementations need to fill out and return when the library is initialized.
<code>ix_s_lkup_table</code>	Data structure that all implementations need to fill out and return when a table is created.

5.1 Defined Types, Enumerations, and Data Structures

5.1.1 Constants

The following constants are used for the data structures listed below:

```
#define IX_LKUP_MEM_SRAM 1
#define IX_LKUP_MEM_DRAM 2
#define IX_LKUP_MEM_TCAM 3
```

5.1.2 `ix_lkup`

`ix_lkup` is a handle that is returned when the application first initializes and gets a handle to the lookup library. To users of this library, the `ix_lkup` handle is opaque. To the implementer of the library, the details of the data structure are detailed in [Section 5.5.1, “`ix_s_lkup`” on page 309](#).

The declaration for the handle is as follows:

```
typedef struct ix_s_lkup * ix_lkup;
```

The users of the API access a set of functions provided by this handle through a set of macro calls. The calls that are supported on this handle are:

`IX_LKUP_CREATE_TABLE()`

`IX_LKUP_DESTROY_TABLE()`

IX_LKUP_FINI()

The behavior of these calls and their arguments are documented in section [Section 5.2.2, “Table Macros”](#).

5.1.3 ix_lkup_table

ix_lkup_table is a handle that is returned when a new table is created by calling IX_LKUP_CREATE_TABLE () on a valid ix_lkup handle. As with the ix_lkup handle, the contents of this structure are opaque to the application. To the implementer of the library, the details of the data structure are detailed in [Section 5.5.1, “ix_s_lkup” on page 309](#).

The declaration for the handle is as follows:

```
typedef struct ix_s_lkup_table * ix_lkup_table;
```

The user of the API access a set of functions provided by this handle through a set of macro calls. The calls that are supported on this handle are:

IX_LKUP_ADD_ENTRY()

IX_LKUP_UPDATE_ENTRY()

IX_LKUP_REMOVE_ENTRY()

IX_LKUP_SEARCH_TABLE()

IX_LKUP_FIND_ENTRY()

IX_LKUP_READ_FIRST_ENTRY()

IX_LKUP_READ_NEXT_ENTRY()

IX_LKUP_RESET_TABLE ()

IX_LKUP_SET_PROPERTY ()

IX_LKUP_GET_PROPERTY()

IX_LKUP_GET_TABLE_INFO ()

The behavior of these calls and their arguments are documented in [Section 5.5.2, “ix_s_lkup_table” on page 310](#).

5.1.4 ix_lkup_table_type

This enumeration defines the different table types and the associated search methods for that table.

```
typedef enum {
    IX_LKUP_TABLE_LPM,      /* longest prefix match table */
    IX_LKUP_TABLE_EXACT,    /* exact match on the key data */
    IX_LKUP_TABLE_RANGE     /* allows range searches, etc */
    IX_LKUP_TABLE_LAST     /* last entry */
} ix_lkup_table_type;
```


5.1.5 ix_lkup_tcam_params

This is a data structure that is passed when initializing the TCAM version of the library by calling `ix_lkup_tcam_init()`. This contains state that is specific to the TCAM implementation.

```
typedef struct ix_lkup_s_tcam_params {
    int    qdrChannel;
    int    qdrDeviceSelect;
} ix_lkup_tcam_params;
```

Data Members

<code>qdrChannel</code>	Indicates which of the Quad Data Rate (QDR) SRAM channels the TCAM is attached to.
<code>qdrDeviceSelect</code>	Indicates which device select is used to select the TCAM device on the QDR SRAM interface. This value is specific to the implementation of the board.

5.1.6 ix_lkup_table_conf

The following structure is used to pass the configuration parameters when a new table is created.

```
typedef struct ix_lkup_s_table_conf {
    ix_lkup_table_type  tableType;
    int                 keySize;
    int                 maxWeight;
    int                 sizeHint;
    int                 dataSize;
    int                 dataLocation;
    int                 dataChannel;
} ix_lkup_table_conf;
```

Data Members

<code>tableType</code>	Type of table to create. This defines the search method and the data structures used for the table.
<code>keySize</code>	Size of the key and mask data in bits. Note: An implementation may round this up to a larger size for implementation efficiency. For example, if the caller specifies the key size is 29 bits, the implementation can round up to 32 bits for the storage if it makes it more convenient.
<code>maxWeight</code>	Specifies the acceptable range of the weights. For example if <code>maxWeight</code> is <code>n</code> , then the valid range of weights is 1 to <code>n</code> . A value of 0 for weight is invalid weight.
<code>sizeHint</code>	A hint provided to the library on the expected number of elements in the table. The library can use this to size the data structures appropriately. In the case of a hardware-assisted lookup this is used as the number of elements to reserve.

Data Members (Continued)

<code>dataSize</code>	Size of associated data that should be stored with the keys. This is the number of bits of data.
<code>dataLocation</code>	Tells the library where it should store the associated data. Possible values are: <ul style="list-style-type: none"> • <code>IX_LKUP_MEM_SRAM</code> indicates data is stored in SRAM. • <code>IX_LKUP_MEM_DRAM</code> indicates data is stored in DRAM. • <code>IX_LKUP_MEM_TCAM</code> indicates data is stored internal to the hardware device.
<code>dataChannel</code>	Indicates which of the memory channels specified with the <code>dataLocation</code> should be used.

5.1.7 `ix_lkup_cookie`

This is an opaque cookie that is passed between some of the API calls, primarily calls that are used for enumerating the contents of a table. The cookie is an opaque entity that is passed by the caller without interpretation.

The typedef is as follows:

```
typedef void * ix_lkup_cookie;
```

5.2 Lookup Management Library

5.2.1 Initialization APIs

There are two different initialization APIs, one for initializing an `ix_lkup` handle for a software lookup management library and another for supporting a TCAM based lookup management library. Having two entry points allow both libraries to co-exist within the same code space. Subsequent accesses to the library functions are provided through indirect function calls defined in the structures.

5.2.1.1 `ix_lkup_sw_init()`

This function is used to initialize the software lookup management library and get a handle to the library for subsequent operations.

C Syntax

```
ix_error ix_lkup_sw_init(
    int arg_LibVersion,
    ix_lkup *arg_phLkup
);
```


Input

<code>arg_LibVersion</code>	The API version the caller is expecting to use. For the current implementation this must be 1. In the future, the version number is used for backwards compatibility.
<code>arg_phLkup</code>	A pointer to a lookup handle. If the call is successful, the lookup library allocates a lookup handle with the initialized values.

Output/Returns

Return Value	<p>Returns <code>IX_SUCCESS</code> if successful or an <code>ix_error</code> token encapsulating one of the following error codes:</p> <ul style="list-style-type: none"> • <code>IX_SUCCESS</code> The library was successfully initialized. • <code>IX_LKUP_ERROR_UNSUPPORTED</code> The version of the library that was requested is not supported. • <code>IX_LKUP_ERROR_RESOURCES</code> Not enough resources are available to complete the request.
--------------	--

5.2.1.2 `ix_lkup_tcam_init()`

This function is used to initialize the TCAM lookup management library and returns a handle to the library for subsequent operations. When the call returns, the TCAM should be initialized and ready to accept new table creation operations. This API may be called once for each TCAM search machine being established in the system.

C Syntax

```
ix_error ix_lkup_tcam_init(
int arg_LibVersion,
ix_lkup_tcam_params *arg_pTcamParams,
void * arg_pVendorParams,
ix_lkup *arg_phLkup
);
```


Input

<code>arg_LibVersion</code>	The API version the caller is expecting to use. For the current implementation this must be 1. In the future the version number is used for backwards compatibility.
<code>arg_pTcamParams</code>	This parameter contains a pointer to a caller supplied parameter block common to all TCAM implemenations.
<code>arg_pVendorParams</code>	This is a pointer to a vendor-specific paramater block that supports vendor-specific extensions. If the vendor-specific features are not being used, the caller should pass NULL pointer for this argument. All implementations should accept NULL as a valid value for this argument.
<code>arg_phLkup</code>	This is a pointer to a lookup handle. If the call is successful, the lookup library updates this location with the handle.

Output/Returns

Return Value	<p>Returns <code>IX_SUCCESS</code> if successful or an <code>ix_error</code> token encapsulating one of the following error codes:</p> <ul style="list-style-type: none"> • <code>IX_SUCCESS</code> The library was successfully initialized. • <code>IX_LKUP_ERROR_UNSUPPORTED</code> The version of the library that was requested is not supported. • <code>IX_LKUP_ERROR_INVALID</code> Some of the parameters passed are invalid. • <code>IX_LKUP_ERROR_RESOURCES</code> Not enough resources are available to complete the request.
--------------	---

5.2.2 Table Macros

This section covers the main table functions as well as table management functions. The main macros are performed using an `ix_lkup` handle returned from the `ix_lkup_sw_init()` function. These functions are all accessed through macro calls using the `ix_lkup` handle. For each call, the macro call is documented, as well as the typedef for the function that is invoked.

The table management functions that are performed using an `ix_lkup_table` handle returned from a successful call to `IX_LKUP_CREATE_TABLE()`. These functions are all accessed through macro calls using the `ix_lkup_table` handle. For each call, the macro call is documented, as well as the typedef for the function that is invoked.

5.2.2.1 `IX_LKUP_CREATE_TABLE()`

This function is used to create a new instance of a search table. If the call is successful, then a handle to the new table is returned that can be used for operations on the table.

Macro Access

This is the macro used to access the table creation function. When called, the macro invokes the function below with the defined arguments.

```
IX_LKUP_CREATE_TABLE(arg_hLkup, arg_pTableConf, arg_VndrConf, arg_phTable);
```

C Syntax

```
typedef ix_error (*ix_lkup_create_table) (
    ix_lkup arg_hLkup,
    ix_lkup_table_conf *arg_pTableConf,
    void * arg_VndrConf,
    ix_lkup_table *arg_phTable
);
```

Input

<code>arg_hLkup</code>	The <code>ix_lkup</code> handle that was returned when the init operation completed.
<code>arg_pTableConf</code>	The table configuration state. This defines the type of table to create as well as the size, search methods etc. The details of the this data structure are defined in ix_lkup_table_conf .
<code>arg_VndrConf</code>	This is a pointer to vendor-specific table configuration information. This is intended to allow vendors to expose additional features of their devices. If the vendor-specific features are not being used, the caller should pass NULL pointer for this argument. All implementations should accept NULL as a valid value for this argument.
<code>arg_phTable</code>	This is a pointer to an <code>ix_lkup_table</code> that is filled if the operation completes successfully. If the call is successful, the lookup library allocates a table handle with the initialized values. The storage for the handle is managed by the library.

Output/Returns

Return Value	<p>Returns <code>IX_SUCCESS</code> if successful or an <code>ix_error</code> token encapsulating one of the following error codes:</p> <ul style="list-style-type: none"> <code>IX_SUCCESS</code> The table was successfully created. <code>IX_LKUP_ERROR_INVALID</code> The handle was invalid, or some of the table configuration information was invalid or not supported. <code>IX_LKUP_ERROR_RESOURCES</code> Not enough resources were available to create the requested table.
--------------	--

5.2.2.2 IX_LKUP_DESTROY_TABLE()

This function is called to destroy a table that was created earlier with [IX_LKUP_CREATE_TABLE\(\)](#). The application should remove all entries from the table before it is destroyed because the implementation may not be able to free these entries.

Macro Access

This macro is used to call the `ix_lkup_destroy_table()` function associated with the `ix_lkup` handle. When called, the macro invokes a function with the typedef defined below. Once the table is destroyed, the handle is de-allocated by the library; subsequently any further operations on the handle are illegal.

```
IX_LKUP_DESTROY_TABLE(arg_hLkup, arg_hTable);
```

C Syntax

```
typedef ix_error (*ix_lkup_destroy_table) (
    ix_lkuparg_hLkup,
    ix_lkup_tablearg_hTable
);
```

Input

<code>arg_hLkup</code>	The handle that was obtained through a successful call to <code>ix_lkup_sw_init()</code> or <code>ix_lkup_tcam_init()</code> .
<code>arg_hTable</code>	The handle for the table to be destroyed.

Output/Returns

Return Value	<p>Returns <code>IX_SUCCESS</code> if successful or an <code>ix_error</code> token encapsulating one of the following error codes:</p> <ul style="list-style-type: none"> • <code>IX_SUCCESS</code> The table was successfully destroyed. • <code>IX_LKUP_ERROR_INVALID</code> One of the handles was invalid.
--------------	--

5.2.2.3 IX_LKUP_FINI()

This function destroys a previous obtained `ix_lkup` handle. This function is typically called when the application is shutting down and is cleaning up. After successful completion of this call, the handle becomes invalid and subsequent calls to this handle cannot be made. Further, the tables created using this handle should not be used anymore and should be destroyed before this function is called.

Macro Access

This is the macro used to access the destroy function. When called, the macro invokes the function defined below.

```
IX_LKUP_FINI(arg_hLkup);
```

C Syntax

```
typedef ix_error (*ix_lkup_fini)
    ix_lkup arg_hLkup
);
```


Input

`arg_hLkup` The handle to destroy. This handle was obtained through a successful call to `ix_lkup_sw_init()` or `ix_lkup_tcam_init()`.

Output/Returns

Return Value Returns `IX_SUCCESS` if successful or an `ix_error` token encapsulating one of the following error codes:

- `IX_SUCCESS` The handle was successfully destroyed.
- `IX_LKUP_ERROR_INUSE` The handle has active tables in use.
- `IX_LKUP_ERROR_INVALID` The handle was invalid

5.2.2.4 IX_LKUP_ADD_ENTRY()

This function adds an entry to a table that was previously created by calling `IX_LKUP_CREATE_TABLE()`. The lookup management library implementation may return an error if it detects a duplicate but the function is not required to make this validation.

Macro Access

This is the macro used to access the add entry function for the table indicated by the table handle. When called, the macro invokes the function defined below.

```
IX_LKUP_ADD_ENTRY(arg_hTable, arg_pKey, arg_pMask, arg_Weight, arg_pData);
```

C Syntax

```
typedef ix_error (*ix_lkup_add_entry) (
    ix_lkup_table arg_hTable,
    ix_uint8 *    arg_pKey,
    ix_uint8 *    arg_pMask,
    int           arg_Weight,
    ix_uint8 *    arg_pData
);
```

Input

`arg_hTable` Handle of the table to update.

`arg_pKey` A pointer to the key data. The length of the key was defined when the table was created.

Input (Continued)

<code>arg_pMask</code>	A pointer to the mask data for this entry. The length of the mask was defined when the table was created.
<code>arg_Weight</code>	The weight for this entry. For EXACT match tables this must be 0. For weighted tables, this should be in the range defined when creating the table. For longest prefix match (LPM), the weight is used to differentiate between entries. Lower values have higher precedence than higher values of weight.
<code>arg_pData</code>	This is a pointer to the data to be associated with the key. This data is opaque to the lookup library and is only interpreted by the higher level applications.

Output/Returns

Return Value	Returns <code>IX_SUCCESS</code> if successful or an <code>ix_error</code> token encapsulating one of the following error codes: <ul style="list-style-type: none"> • <code>IX_SUCCESS</code> The entry was added to the table. • <code>IX_LKUP_ERROR_INVALID</code> Some of the parameters passed are not valid. • <code>IX_LKUP_ERROR_EXISTS</code> An entry with the same key, mask, weight, and data is already in the table. • <code>IX_LKUP_ERROR_CONFLICT</code> An entry with the same key, mask, weight but different data is in the table. • <code>IX_LKUP_ERROR_RESOURCES</code> Not enough resources are available to complete the operation. Either the maximum table size has been exceeded or other resources could not be allocated.
--------------	--

5.2.2.5 `IX_LKUP_REMOVE_ENTRY ()`

This function removes an entry that was previously added to the table. The entry is identified by the values used to add the entry.

Macro Access

This is the macro used to remove an entry from the existing table. When called, the macro invokes a function with the typedef defined below.

```
IX_LKUP_REMOVE_ENTRY (arg_hTable, arg_pKey, arg_pMask, arg_Weight);
```

C Syntax

```
typedef ix_error (*ix_lkup_remove_entry) (
    ix_lkup_table arg_hTable,
    ix_uint8 *    arg_pKey,
    ix_uint8 *    arg_pMask,
    int           arg_Weight
);
```


Input

<code>arg_hTable</code>	Handle of the table to update.
<code>arg_pKey</code>	A pointer to the key data of the item to removed. The length of the data was defined when the table was created.
<code>arg_pMask</code>	A pointer to the mask data of the entry to be removed. The length of the mask data was defined when the table was created.
<code>arg_Weight</code>	The weight for this entry to be removed. For EXACT match tables this must be 0. For weighted tables, this should be in the range defined when creating the table. For longest prefix match (LPM), the weight is used to differentiate between entries.

Output/Returns

Return Value	Returns <code>IX_SUCCESS</code> if successful or an <code>ix_error</code> token encapsulating one of the following error codes: <ul style="list-style-type: none"> <code>IX_SUCCESS</code> The entry was removed to the table. <code>IX_LKUP_ERROR_INVALID</code> The table handle was not valid. <code>IX_LKUP_ERROR_NOTFOUND</code> A matching entry was not found.
--------------	--

5.2.2.6 IX_LKUP_UPDATE_ENTRY()

This function updates the data associated with an element already in the table.

Macro Access

This is the macro used to call the `IX_LKUP_UPDATE_ENTRY` function associated with the table handle. The function definition is given below.

```
IX_LKUP_UPDATE_ENTRY(arg_hTable, arg_pKey, arg_pMask, arg_pWeight,
    arg_pNewData);
```

C Syntax

```
typedef ix_error (*ix_lkup_update_entry)(
    ix_lkup_table arg_hTable,
    ix_uint8 *    arg_pKey,
    ix_uint8 *    arg_pMask,
    int           arg_Weight,
    ix_uint8 *    arg_pNewData
);
```


Input

<code>arg_hTable</code>	Handle of the table to update.
<code>arg_pKey</code>	A pointer to the key data of the item to update. The length of the data was defined when the table was created.
<code>arg_pMask</code>	A pointer to the mask data of the entry to be updated. The length of the mask data was defined when the table was created.
<code>arg_Weight</code>	The weight for this entry to be updated. For EXACT match tables this must be 0. For weighted tables, this should be in the range defined when creating the table. For longest prefix match (LPM), the weight is used to differentiate between entries.
<code>arg_pNewData</code>	A pointer to the new data to associate with the specific key. The size of the data was specified when the table was created.

Output/Returns

Return Value	Returns <code>IX_SUCCESS</code> if successful or an <code>ix_error</code> token encapsulating one of the following error codes: <ul style="list-style-type: none"> • <code>IX_SUCCESS</code> The entry associated data was updated. • <code>IX_LKUP_ERROR_INVALID</code> The table handle was not valid. • <code>IX_LKUP_ERROR_NOTFOUND</code> A matching entry was not found so the update was not performed.
--------------	---

5.2.2.7 `IX_LKUP_SEARCH_TABLE()`

Calls to `IX_LKUP_SEARCH_TABLE` search a specific table and return the associated data if a match is found. As described in previous APIs, the search key is compared with the entries and masks. If multiple entries match, then the one with the lowest weight is returned.

Macro Access

This is the macro used to access `IX_LKUP_SEARCH_TABLE` function for the specified table entry. The function is defined below.

```
IX_LKUP_SEARCH_TABLE(arg_hTable, arg_pKey, arg_pData);
```

C Syntax

```
typedef ix_error (*ix_lkup_search_table) (
    ix_lkup_table arg_hTable,
    ix_uint8 *    arg_pKey,
    ix_uint8 *    arg_pData
);
```


Input

<code>arg_hTable</code>	Handle of the table to search.
<code>arg_pKey</code>	A pointer to the key data used for the search. The length of the key should correspond to the key size specified when the table was created.
<code>arg_pData</code>	This is a pointer to a location where the associated data is stored if the search was successful.

Output/Returns

Return Value	<p>Returns <code>IX_SUCCESS</code> if successful or an <code>ix_error</code> token encapsulating one of the following error codes:</p> <ul style="list-style-type: none"> • <code>IX_SUCCESS</code> The search was successful and a result was returned. • <code>IX_LKUP_ERROR_NOTFOUND</code> There was no entry that matched this search request. • <code>IX_LKUP_ERROR_INVALID</code> The handle was invalid.
--------------	---

5.2.2.8 `IX_LKUP_FIND_ENTRY()`

This function is used to search a table for an entry that matches the exact key and mask, weight combination. This is used to locate a specific entry for search types where multiple entries may match a specific search.

Macro Access

This is the macro used to access `ix_lkup_find_entry` function associated with the table handle. The function declaration is given below.

```
IX_LKUP_FIND_ENTRY(arg_hTable, arg_pKey, arg_pMask, arg_Weight, arg_pData);
```

C Syntax

```
typedef ix_error (*ix_lkup_find_entry)(
    ix_lkup_table arg_hTable,
    ix_uint8 *    arg_pKey,
    ix_uint8 *    arg_pMask,
    int           arg_Weight,
    ix_uint8 *    arg_pData
);
```


Input

<code>arg_hTable</code>	Handle for the table to search.
<code>arg_pKey</code>	A pointer to the key data used for the search. The length of the data was defined when the table was created.
<code>arg_pMask</code>	A pointer to the mask data to use for the search.
<code>arg_Weight</code>	This is the associated weight with the entry to find.
<code>arg_pData</code>	Points to a location where the associated data is stored if the search was successful. It is assumed that the caller allocated sufficient space data associated with the call.

Output/Returns

Return Value	<p>Returns <code>IX_SUCCESS</code> if successful or an <code>ix_error</code> token encapsulating one of the following error codes:</p> <ul style="list-style-type: none"> • <code>IX_SUCCESS</code> The search was successful and a result was returned. • <code>IX_LKUP_ERROR_NOTFOUND</code> There was no entry that matched this search request. • <code>IX_LKUP_ERROR_INVALID</code> The handle was invalid.
--------------	---

5.2.2.9 `IX_LKUP_READ_FIRST_ENTRY()`

This is a debug function that is used to read the items that are stored in the table. This function gets the first item in the table. To continue iterating through all the entries, the programmer calls this function once, and then repeatedly calls `IX_LKUP_READ_NEXT_ENTRY()` until all the entries have been enumerated.

Macro Access

This is the macro used to access `ix_lkup_read_first_entry()` function in the table handle. When called, the macro invokes the function defined below.

```
IX_LKUP_READ_FIRST_ENTRY(arg_hTable, arg_pKey, arg_pMask, arg_pWeight,
    arg_pData, arg_pCookie);
```

C Syntax

```
typedef ix_error (*ix_lkup_read_first_entry)(
    ix_lkup_table arg_hTable,
    ix_uint8 *    arg_pKey,
    ix_uint8 *    arg_pMask,
    int *         arg_pWeight,
    ix_uint8 *    arg_pData,
    ix_lkup_cookie *arg_pCookie
);
```


Input

<code>arg_hTable</code>	The handle to the table to read.
<code>arg_pKey</code>	A pointer where the key data is stored if the call is successful. The caller should make sure there is sufficient storage for the key size of the table.
<code>arg_pMask</code>	A pointer where the mask data is stored. The caller should ensure that this structure is of sufficient size to store the mask.
<code>arg_pWeight</code>	A pointer where the weight data should be stored.
<code>arg_pData</code>	A pointer to the location where the associated data is stored if the call is successful.
<code>arg_pCookie</code>	A pointer to an <code>ix_lkup_cookie</code> where the cookie is stored. The cookie is passed to a subsequent call to <code>IX_LKUP_READ_NEXT_ENTRY()</code> .

Output/Returns

Return Value	<p>Returns <code>IX_SUCCESS</code> if successful or an <code>ix_error</code> token encapsulating one of the following error codes:</p> <ul style="list-style-type: none"> <code>IX_SUCCESS</code> The call was successful and the data structures have been populated. <code>IX_LKUP_ERROR_INVALID</code> Some of the parameters passed are not valid. <code>IX_LKUP_ERROR_NOTFOUND</code> This is returned when there are no entries in the table. In this case none of the data structures to be filled in have valid data.
--------------	--

5.2.2.10 `IX_LKUP_READ_NEXT_ENTRY()`

This is a debug function that is used to read the items that are stored in the table. This function is repeatedly called after `IX_LKUP_READ_FIRST_ENTRY()` to enumerate the contexts of the table. Each call returns a cookie that is passed to the next call.

Macro Access

This is the macro used to access `ix_lkup_read_next_entry` function in the table handle. When called, the macro invokes the function defined below for the specified table.

```
IX_LKUP_READ_NEXT_ENTRY(arg_hTable, arg_OldCookie, arg_pKey, arg_pMask,
    arg_pWeight, arg_pData, arg_pNewCookie);
```

C Syntax

```
typedef ix_error (*ix_lkup_read_next_entry)(
    ix_lkup_table arg_hTable,
    ix_lkup_cookie arg_OldCookie,
    ix_uint8 * arg_pKey,
```



```

ix_uint8 *    arg_pMask,
int *         arg_pWeight,
ix_uint8 *    arg_pData,
ix_lkup_cookie * arg_pNewCookie
);

```

Input

<code>arg_hTable</code>	The handle to the table to read.
<code>arg_OldCookie</code>	Cookie returned by the last successful call to IX_LKUP_READ_FIRST_ENTRY() or IX_LKUP_READ_NEXT_ENTRY() .
<code>arg_pKey</code>	A pointer where the key data is stored if the call is successful. The caller should make sure there is sufficient storage for the key size of the table.
<code>arg_pMask</code>	A pointer where the mask data is stored. The caller should ensure that this structure is of sufficient size to store the mask.
<code>arg_pWeight</code>	A pointer where the weight data should be stored.
<code>arg_pData</code>	A pointer to the location where the associated data is stored if the call is successful.
<code>arg_pNewCookie</code>	A pointer to an <code>ix_lkup_cookie</code> where the cookie is stored. The cookie is passed to a subsequent call to IX_LKUP_READ_NEXT_ENTRY() .

Output/Returns

Return Value	<p>Returns <code>IX_SUCCESS</code> if successful or an <code>ix_error</code> token encapsulating one of the following error codes:</p> <ul style="list-style-type: none"> <code>IX_SUCCESS</code> The call was successful and the data structures have been populated. <code>IX_LKUP_ERROR_INVALID</code> Some of the parameters passed are not valid. <code>IX_LKUP_ERROR_NOTFOUND</code> This is returned when there are no more entries in the table.
--------------	---

5.2.2.11 [IX_LKUP_RESET_TABLE\(\)](#)

This function clears all the items in the table and resets it to its initial state. Note that the implementation tries to recover all the states, but it may not be possible in all implementations. The callers should try remove all known entries from the table before calling this function.

Macro Access

This is the macro used to access the `ix_lkup_reset_table` function associated with the table handle. When called, the macro invokes the function with the typedef defined below.

```
IX_LKUP_RESET_TABLE (arg_hTable);
```


C Syntax

```
typedef ix_error (*ix_lkup_reset_table)(
    ix_lkup_table * arg_hTable
);
```

Input

`arg_hTable` The handle of the table to update.

Output/Returns

Return Value Returns `IX_SUCCESS` if successful or an `ix_error` token encapsulating one of the following error codes:

- `IX_SUCCESS` The table was reset properly.
- `IX_LKUP_ERROR_INVALID` The table handle was not valid.

5.2.2.12 `IX_LKUP_SET_PROPERTY()`

This function allows the caller to set special attributes of the table. No attributes are currently defined but this function may be used by vendors to export special features of their devices such as timeout abilities, etc.

Macro Access

This is the macro used to access `ix_lkup_set_property` function associated with the table. When called, the macro invokes the function with the typedef defined below.

```
IX_LKUP_SET_PROPERTY (arg_hTable, arg_Property, arg_pValue);
```

C Syntax

```
typedef ix_error (*ix_lkup_set_property)(
    ix_lkup_table arg_hTable,
    int           arg_Property,
    void *        arg_pValue
);
```

Input

`arg_hTable` The handle of the table to update.

`arg_Property` The value of the property to set. This is defined by the specific implementation.

`arg_pValue` A pointer data associated with the property.

Output/Returns

Return Value	<p>Returns <code>IX_SUCCESS</code> if successful or an <code>ix_error</code> token encapsulating one of the following error codes:</p> <ul style="list-style-type: none"> • <code>IX_SUCCESS</code> The property was set. • <code>IX_LKUP_ERROR_INVALID</code> The table handle was invalid. • <code>IX_LKUP_ERROR_UNSUPPORTED</code> The defined property was not supported.
--------------	--

5.2.2.13 `IX_LKUP_GET_PROPERTY()`

This function allows the caller to get special attributes of the table. No attributes are currently defined, but this function may be used by vendors to export special features of their devices such as timeout abilities, etc.

Macro Access

This is the macro used to access `ix_lkup_get_property` function associated with the table. When called, the macro invokes the function with the typedef defined below.

```
IX_LKUP_GET_PROPERTY (arg_hTable, arg_Property, arg_pValue);
```

C Syntax

```
typedef ix_error (*ix_lkup_set_property)(
    ix_lkup_table arg_hTable,
    int          arg_Property,
    void         *arg_pValue
);
```

Input

<code>arg_hTable</code>	The handle of the table to update.
<code>arg_Property</code>	The value of the property to get. These are defined by the specific implementation.
<code>arg_pValue</code>	A pointer where the associated data is returned with a successful call.

Output/Returns

Return Value	<p>Returns <code>IX_SUCCESS</code> if successful or an <code>ix_error</code> token encapsulating one of the following error codes:</p> <ul style="list-style-type: none"> • <code>IX_SUCCESS</code> The property was successfully returned. • <code>IX_LKUP_ERROR_INVALID</code> The table handle was invalid. • <code>IX_LKUP_ERROR_UNSUPPORTED</code> The defined property was not supported.
--------------	--

5.2.2.14 IX_LKUP_GET_TABLE_INFO()

This function returns values that need to be used by the microengines:

Table Identifier – This identifier encodes the information needed by the microengines to perform the lookup. The contents and format of the ID are implementation-specific. It is the responsibility of the application using this API to get this value to the microengines that are calling the search APIs.

Table Data 1 and Table Data 2 – The table data that is going to be used by the microengine to find the associated data with an element. These values need to be passed to some of the microengine calls. The contents and format of the data are implementation-specific. It is the responsibility of the application using this API to get these values to the microengine.

Macro Access

This is the macro used to access `ix_lkup_get_table_info` function associated with the table handle. When called, the macro invokes the function defined below.

```
IX_LKUP_GET_TABLE_INFO (arg_hTable, arg_pTableId, arg_pTableData1,
    arg_pTableData2);
```

C Syntax

```
typedef ix_error (*ix_lkup_get_table_info) {
    ix_lkup_table arg_hTable,
    ix_uint32 *   arg_pTableId,
    ix_uint32 *   arg_pTableData1,
    ix_uint32 *   arg_pTableData2
};
```

Input

<code>arg_hTable</code>	The handle of the table to update.
<code>arg_pTableId</code>	A pointer to the location where the table ID is stored.
<code>arg_pTableData1</code>	A pointer to the location where the data1 information is stored.
<code>arg_pTableData2</code>	A pointer to the location where the data2 information is stored.

Output/Returns

Return Value	Returns <code>IX_SUCCESS</code> if successful or an <code>ix_error</code> token encapsulating one of the following error codes: <ul style="list-style-type: none"> <code>IX_SUCCESS</code> The table information was returned. <code>IX_LKUP_ERROR_INVALID</code> The table handle was not valid.
--------------	---

5.3 Microengine Hardware Lookup

This section describes the APIs available to microengine programmers for hardware managed tables. The design of this API assumes that all table management (creation, adds, deletes, etc.) is performed on the Intel XScale® microarchitecture. The microengines only have APIs to search the tables.

Because of the nature of the hardware devices, there is only a single set of APIs to search all the different table types. This differs from the software lookup APIs described in [Section 5.4](#), “Microengine Software Lookup” on page 303.

Since the tables are managed and created on the Intel XScale® microarchitecture, a certain amount of state must be passed to the microengines so that the tables can be searched. This state information is the `tableid` and the `data1` and `data2` implementation-specific information constants. The programmer must get this state after the table is created and pass the information to the microengines. One typical way of passing the data is to use import variables and patch the values after the tables have been initialized.

The API consists of 4 calls. The first is to build a handle to use when searching the table. The second call initiates the search, the third call completes the search, and the fourth call gets the associated data.

5.3.1 TCAM Lookup APIs

These APIs are used for all hardware managed tables.

5.3.1.1 `ix_tcam_lkup_build_handle()`

This function builds the handle that must be passed to the search functions. This handle is an implementation-specific value that encodes the table location as well as thread-specific information to keep track of multiple search contexts (if using a hardware search mechanism).

Microengine Assembler Syntax

```
ix_tcam_lkup_build_handle(out_handle, in_table_id, in_context, in_other);
```

Microengine C Syntax

```
void ix_tcam_lkup_build_handle(  
    void * out_handle,  
    ix_uint32 in_table_id,  
    void * in_context,  
    void * in_other);
```


Input

<code>in_table_id</code>	This is the table ID that was obtained on the Intel XScale® core by calling <code>IX_LKUP_GET_TABLE_INFO()</code> . This value was passed to the microcode. This encodes some implementation-specific data such as the table number, how the table is organized, etc.
<code>in_context</code>	This is a unique context value for each of the threads that call the search routines. In a TCAM implementation this is used to correlate the original request and the results. The format of this parameter is vendor-defined.
<code>in_other</code>	This is other application-specific information that may be passed in. For example, this may be a mask register ID for a TCAM implementation. A value of 0 should be passed if there is no other data to pass in. The format of this parameter is vendor-defined.

Output/Returns

<code>out_handle</code>	The handle that is passed to the <code>ix_tcam_lkup_start()</code> and <code>ix_tcam_lkup_complete()</code> calls. The caller can choose to build the handle before each reference or do this call at initialization time and save the result.
-------------------------	--

5.3.1.2 `ix_tcam_lkup_start()`

This function starts a search using the `in_key` to launch the search request. A separate call to `ix_tcam_lkup_complete()` is used to pick up the results

The caller should be aware that this call may perform a context swap allowing other threads to run before the call completes.

The programmer is responsible for making sure the `ix_tcam_lkup_complete()` call is not called too soon, so that the results are not ready yet. The definition of “too soon” is vendor-specific and is documented by them.

Microengine Assembler Syntax

```
ix_tcam_lkup_start(in_key, in_length, in_handle);
```

Microengine C Syntax

```
void ix_tcam_lkup_start(
    uint32 * in_key,
    int in_length,
    void * in_handle);
```


Input

<code>in_key</code>	An array of registers that holds the key information. This can either be write transfer or general purpose registers.
<code>in_length</code>	This number of registers that is used to hold the key. The exact key length was set up when the table was configured. This argument must be a constant.
<code>in_handle</code>	This is the table handle that was created by calling the <code>ix_tcam_lkup_build_handle()</code> API.

Output/Returns

None.

5.3.1.3 `ix_tcam_lkup_complete()`

This call completes a search that was started and returns the results.

The programmer needs to tune the code to make sure that this call is not called too early. If it is called too early, the implementation may need to perform several memory reads while polling to see if the operation has completed. The implementer should get information from the TCAM vendor on typical times to finish the request.

The caller should be aware that this call may perform a context swap allowing other threads to run before the call completes.

Microengine Assembler Syntax

```
ix_tcam_lkup_complete(out_hit, out_cookie, in_cookie_size, in_handle);
```

Microengine C Syntax

```
void ix_tcam_lkup_complete(
    int out_hit,
    uint32 * out_cookie,
    int in_cookie_size,
    void in_handle);
```

Input

<code>in_cookie_size</code>	This is the size of the cookie given in number of registers.
<code>in_handle</code>	The handle for the search built with <code>ix_tcam_lkup_build_handle()</code> . This must be the same handle that was passed to <code>ix_tcam_lkup_start()</code> .

Output/Returns

<code>out_hit</code>	The <code>out_hit</code> contains a boolean value that indicates whether the search was successful. The value is set to 1 if there is an entry that matches and 0 if there is no entry that matches.
<code>out_cookie</code>	This may contain temporary state that is passed to <code>ix_tcam_lkup_get_data()</code> . This is an register array of input transfer registers.

5.3.1.4 `ix_tcam_lkup_get_data()`

This macro returns the data associated with a successful search. The cost of calling this may vary depending on how data is stored. In some applications, the associated data may actually be obtained from the previous call and this macro just copies it if needed. In other implementations, this call may need to access memory to get its state and as such the thread may yield during execution of this call.

Microengine Assembler Syntax

```
ix_tcam_lkup_lpm_get_data(out_data, in_data_size, in_cookie, in_cookie_size,
    in_handle, in_mem_type, in_data1, in_data2);
```

Microengine C Syntax

```
void ix_tcam_lkup_lpm_get_data(
    uint32 * out_data,
    int in_data_size,
    uint32 * in_cookie,
    int in_cookie_size,
    void * in_handle,
    int in_mem_type,
    void * in_data1,
    void * in_data2);
```

Input

<code>in_data_size</code>	This is the size of the data that is needed given in registers. This must be a constant.
<code>in_cookie</code>	This is the cookie that was returned by <code>ix_tcam_lkup_complete()</code> .
<code>in_cookie_size</code>	This is the size of the cookie given in number of registers. This must be a constant.
<code>in_handle</code>	The handle for the search built with <code>ix_tcam_lkup_build_handle()</code> . This must be the same handle that was passed to <code>ix_tcam_lkup_complete()</code> .

Input (Continued)

<code>in_memtype</code>	<p>This is the type of memory associated with data. The value passed must be a constant from one of the following values:</p> <ul style="list-style-type: none"> • <code>IX_LKUP_MEM_TCAM</code> indicates locally attached memory. • <code>IX_LKUP_MEM_SRAM</code> indicates SRAM. • <code>IX_LKUP_MEM_DRAM</code> indicates DRAM.
<code>in_data1</code>	<p>This is one of the pieces of data information that was passed from the Intel XScale® core after the table was initialized. This value was obtained by calling <code>IX_LKUP_GET_TABLE_INFO()</code>.</p>
<code>in_data2</code>	<p>This is one of the pieces of data information that was passed from the Intel XScale® core after the table was initialized. This value was obtained by calling <code>IX_LKUP_GET_TABLE_INFO()</code>.</p>

Output/Returns

<code>out_data</code>	<p>This is an array of read/write transfer registers or general purpose registers that holds the resulting data.</p>
-----------------------	--

5.4 Microengine Software Lookup

This section describes the APIs that are used to search software tables. These APIs are provided as microcode macros and as micro-C libraries.

The APIs are divided into three parts, one set of APIs for each defined table type:

- The longest prefix match calls are used for searching tables created with type `IX_LKUP_TABLE_LPM` as the table type.
- The exact match calls are used for searching tables created with type `IX_LKUP_TABLE_EXACT` as the table type.
- The range match calls are used for searching tables created with type `IX_LKUP_TABLE_RANGE` as the table type.

All of the searches take a key and a table ID as input. The key is constructed by a higher-level application. The table ID is one of the fields in the `table_handle_t` structure. This structure is not user-accessible, it is used by the libraries to encode the location of the table to search.

5.4.1 Longest Prefix Match APIs

The longest prefix match calls are used for searching tables created with type `IX_LKUP_TABLE_LPM` as the table type.

5.4.1.1 `ix_sw_lkup_lpm_build_handle()`

This function builds the handle that must be passed to the search functions. This handle is an implementation-specific value that encodes the table location as well as thread-specific information to keep track of multiple search contexts (if using a hardware search mechanism).

Microengine Assembler Syntax

```
ix_sw_lkup_lpm_build_handle(out_handle, in_table_id);
```

Microengine C Syntax

```
void ix_sw_lkup_lpm_build_handle(
    _void * out_handle,
    uint32 in_table_id);
```

Input

<code>in_table_id</code>	This is the table ID that was obtained on the Intel XScale [®] core by calling <code>IX_LKUP_GET_TABLE_INFO()</code> . This value was passed to the microcode. This encodes some implementation-specific data such as the table number, how the table is organized, etc.
--------------------------	---

Output/Returns

<code>out_handle</code>	The handle that is passed to the <code>ix_sw_lkup_lpm_search()</code> call. The caller can choose to build the handle before each reference or do this call at initialization time and save the result.
-------------------------	---

5.4.1.2 `ix_sw_lkup_lpm_search()`

This function searches a longest prefix match table and returns the results. The caller should assume that this macro swaps out one or more times during the execution of the macro.

Microengine Assembler Syntax

```
ix_sw_lkup_lpm_search(out_hit, out_data, in_key, in_length, in_handle,
    in_data_size, in_data1, in_data2);
```

Microengine C Syntax

```
void ix_sw_lkup_lpm_search(
    int out_hit,
    uint32 * out_data,
    uint32 * in_key,
    int in_length,
    void * in_handle,
    int in_data_size,
```



```
void * in_data1,
void * in_data2);
```

Input

<code>in_key</code>	This is an array of general purpose registers that holds the key. It is assumed that the array is large enough to hold the appropriate number of byte specified below.
<code>in_length</code>	The length or the key data in bytes. This argument must be a constant.
<code>in_handle</code>	This is the table handle that was created by calling the <code>ix_sw_lkup_lpm_build_handle()</code> call.
<code>in_data_size</code>	The amount of data to read from the result. This is given in bytes. This must be a constant.
<code>in_data1</code>	This is one of the pieces of data information that was passed from the Intel XScale [®] core after the table was initialized. This value was obtained by calling <code>IX_LKUP_GET_TABLE_INFO()</code> .
<code>in_data2</code>	This is one of the pieces of data information that was passed from the Intel XScale [®] core after the table was initialized. This value was obtained by calling <code>IX_LKUP_GET_TABLE_INFO()</code> .

Output/Returns

<code>out_hit</code>	The <code>out_hit</code> contains a boolean that indicates if the search was successful or not. If the search was successful, then this is set to 1 (true), otherwise it is set to 0 (false).
<code>out_data</code>	This is an array of read transfer registers that is used to store the resulting data associated with the call. The results are only written here if the results are successful. The contents of <code>out_data</code> are undefined if the search fails.

5.4.2 Exact Match APIs

The exact match calls are used for searching tables created with type `IX_LKUP_TABLE_EXACT` as the table type.

5.4.2.1 `ix_sw_lkup_exact_build_handle()`

This function builds the handle that must be passed to the search functions. This handle is an implementation-specific value that encodes the table location as well as thread-specific information to keep track of multiple search contexts (if using a hardware search mechanism).

Microengine Assembler Syntax

```
ix_sw_lkup_exact_build_handle(out_handle, in_table_id);
```


Microengine C Syntax

```
void ix_sw_lkup_exact_build_handle(
    _void * out_handle,
    uint32 in_table_id);
```

Input

in_table_id	This is the table ID that was obtained on the Intel XScale® core by calling IX_LKUP_GET_TABLE_INFO() . This is typically passed to the application through an import variable.
-------------	--

Output/Returns

out_handle	The handle that is passed to the ix_sw_lkup_exact_search() calls. The caller can choose to build the handle before each reference or do this call at initialization time and save the result.
------------	---

5.4.2.2 [ix_sw_lkup_exact_search\(\)](#)

This function searches the exact match table and returns the results. The caller should assume that this macro swaps out one or more times during the execution of the macro.

Microengine Assembler Syntax

```
ix_sw_lkup_exact_search(out_hit, out_data, in_key, in_length, in_handle,
    in_data_size, in_data1, in_data2);
```

Microengine C Syntax

```
void ix_sw_lkup_exact_search (
    int out_hit,
    uint32 * out_data,
    uint32 * in_key,
    int in_length,
    void * in_handle,
    int in_data_size,
    void * in_data1,
    void * in_data2);
```


Input

<code>in_key</code>	This is an array of general purpose registers that holds the key. It is assumed that the array is large enough to hold the appropriate number of byte specified below.
<code>in_length</code>	The length of the key data in bytes. This argument must be a constant.
<code>in_handle</code>	This is the table handle that was created by calling the ix_sw_lkup_exact_build_handle() call.
<code>in_data_size</code>	The amount of data to read from the result. This is given in bytes. This must be a constant.
<code>in_data1</code>	This is one of the pieces of data information that was passed from the Intel XScale [®] core after the table was initialized. This value was obtained by calling IX_LKUP_GET_TABLE_INFO() .
<code>in_data2</code>	This is one of the pieces of data information that was passed from the Intel XScale [®] core after the table was initialized. This value was obtained by calling IX_LKUP_GET_TABLE_INFO() .

Output/Returns

<code>out_hit</code>	The <code>out_hit</code> contains a boolean that indicates if the search was successful or not. If the search was successful, then this is set to 1 (true), otherwise it is set to 0 (false).
<code>out_data</code>	This is the set of registers that is used to store the resulting data associated with the call. The results are only written here if the results are successful. The contents of <code>out_data</code> are undefined if the search fails.

5.4.3 Range Match APIs

The range match calls are used for searching tables created with type `IX_LKUP_TABLE_RANGE` as the table type.

5.4.3.1 [ix_sw_lkup_range_build_handle\(\)](#)

This function builds the handle that must be passed to the search functions. This handle is an implementation-specific value that encodes the table location as well as thread-specific information to keep track of multiple search contexts (if using a hardware search mechanism).

Microengine Assembler Syntax

```
ix_sw_lkup_range_build_handle(out_handle, in_table_id);
```


Microengine C Syntax

```
void ix_sw_lkup_range_build_handle(
    _void * out_handle,
    uint32 in_table_id);
```

Input

in_table_id	This is the table ID that was obtained on the Intel XScale® core by calling IX_LKUP_GET_TABLE_INFO() . This is typically passed to the application through an import variable.
-------------	--

Output/Returns

out_handle	The handle that is passed to the ix_sw_lkup_range_search() calls. The caller can choose to build the handle before each reference or do this call at initialization time and save the result.
------------	---

5.4.3.2 ix_sw_lkup_range_search()

This function searches the range matching table and returns the results. The caller should assume that this macro swaps out one or more times during the execution of the macro.

Microengine Assembler Syntax

```
ix_sw_lkup_range_search(out_hit, out_data, in_key, in_length, in_handle,
    in_data_size, in_data1, in_data2);
```

Microengine C Syntax

```
void ix_sw_lkup_range_search (
    int out_hit,
    uint32 * out_data,
    uint32 * in_key,
    int in_length,
    void * in_handle,
    int in_data_size,
    void * in_data1,
    void * in_data2);
```


Input

<code>in_key</code>	This is an array of general purpose registers that holds the key. It is assumed that the array is large enough to hold the appropriate number of byte specified below.
<code>in_length</code>	The length of the key data in bytes. This argument must be a constant.
<code>in_handle</code>	This is the table handle that was created by calling the <code>ix_sw_lkup_range_build_handle()</code> call.
<code>in_data_size</code>	The amount of data to read from the result. This is given in bytes. This must be a constant.
<code>in_data1</code>	This is one of the pieces of data information that was passed from the Intel XScale® core after the table was initialized. This value was obtained by calling <code>IX_LKUP_GET_TABLE_INFO()</code> .
<code>in_data2</code>	This is one of the pieces of data information that was passed from the Intel XScale® core after the table was initialized. This value was obtained by calling <code>IX_LKUP_GET_TABLE_INFO()</code> .

Output/Returns

<code>out_hit</code>	The <code>out_hit</code> contains a boolean that indicates if the search was successful or not. If the search was successful, then this is set to 1 (true), otherwise it is set to 0 (false).
<code>out_data</code>	This is the set of registers that is used to store the resulting data associated with the call. The results are only written here if the results are successful. The contents of <code>out_data</code> are undefined if the search fails.

5.5 Implementation Considerations

When implementing the library there are two common structures that all implementations need to export. It is expected that most implementations also need additional state in the handle that is not in the base data structure. The implementation can support this by defining their own data structures, which includes the base data structure at the beginning of the data structure.

Also, it is likely that TCAM vendors will provide additional functions that are not included in this set. The vendor can do this by adding additional function pointers to their own data structures and providing macros that de-reference them.

5.5.1 `ix_s_lkup`

This is the data structure that all implementations need to fill out and return when the library is initialized. Only the implementation should ever care about this data structure. The data structure contains all the function pointers for the operations that can be performed on an `ix_lkup` handle.

The definition of the structure is as follows. All of the functions were defined in previous sections.

```
typedef struct ix_s_lkup {  
    ix_lkup_create_table ldataCreateTable;  
    ix_lkup_destroy_table ldataDestroyTable;  
    ix_lkup_fini          ldataFini;  
} ix_s_lkup;
```

5.5.2 ix_s_lkup_table

This is the data structure that all implementations need to fill out and return when a table is created. Only the implementation should ever care about this data structure. The data structure contains function pointers for the operations that can be performed on an ix_lkup_table.

The definition of the structure is as follows. All of the functions were defined in previous sections.

```
typedef struct ix_s_lkup_table {  
    ix_lkup_add_entry      ltabAddEntry;  
    ix_lkup_remove_entry   ltabRemoveEntry;  
    ix_lkup_update_entry   ltabUpateEntry;  
    ix_lkup_search_table   ltabSearchTable;  
    ix_lkup_find_entry     ltabFindEntry;  
    ix_lkup_read_first_entry ltabReadFirstEntry;  
    ix_lkup_read_next_entry ltabReadNextEntry;  
    ix_lkup_reset_table    ltabResetTable;  
    ix_lkup_set_property   ltabSetProperty;  
    ix_lkup_get_property   ltabGetProperty;  
    ix_lkup_get_table_info ltabGetTableInfo;  
} ix_s_lkup_table;
```




Operating System Services Layer (OSSL) Support

6

The IXA SDK Tools CD provides additional library support for the Operating System Services Layer (OSSL). Applications based on the IXA SDK use the OSSL library's operating system independent APIs and data types for system services such as threads, semaphores, and memory management, etc.

For details concerning the use of these libraries and the specific APIs they support, see the *Intel® Internet Exchange Architecture (IXA) Software Reference Manual* located on the IXA SDK Tools CD.

The IXA SDK Tools CD provides additional library support for Intel XScale® core applications. This includes Intel XScale® core support for:

- Microengine Loader, which is used to load microcode images, created by the microcode linker `ucld`, to the appropriate microengines.
- Hardware Abstraction Layer (HAL), which generates code that interfaces to the Intel XScale® core hardware or the Transactor. An Intel XScale® core application that uses HAL can run in hardware mode or simulation mode, for increased code portability.
- Tools libraries, which contain additional APIs that provide support for debugging and remote debugging of microengine code from the Intel XScale® core.

For details concerning the use of these libraries and the specific APIs they support, see the *Intel® Internet Exchange Architecture (IXA) Software Reference Manual* located on the IXA SDK Tools CD.



Optimized Data Plane Libraries Support

8

The IXA SDK Tools CD provides additional library support for the optimized data plane libraries. These libraries consist of generic microengine software building blocks used to construct an application's microengine modules, called microblocks. The optimized data plane macro libraries are reusable software functions optimized for high performance and minimal executable code size.

For details concerning the use of these libraries and the specific APIs they support, see the *Intel® Internet Exchange Architecture Optimized Data Plane Libraries Reference Manual* located on the IXA SDK Tools CD.

9.1 Introduction

The Intel® IXA SDK contains a metadata configuration tool, which enhances the flexibility of the existing metadata structures defined within the IXA Portability Framework. This utility takes metadata configuration files as input and parses them, making decisions such as the total metadata length required, the ordering of metadata fields, etc.

The output of the metadata configuration tool is a standard `dl_meta.uc` file with all metadata fields assigned an offset, mask, and shift value. The outputted `dl_meta.uc` file also contains a group of accessor macros for all microblocks to get and set each field of the metadata independent of the other metadata fields. In addition, the tool generates block macros to initialize the entire block, load and flush cache to and from transfer registers. The tool also generates a `buffer.h` file which contains Intel XScale® core data structures.

9.2 Metadata Configuration Tool Process

Using the metadata configuration tool is a two-step process:

1. The application configuration file is given as input to the tool. The tool parses the application, dispatch loop, and microblock configuration files and generates an intermediate file called a list file. The list file contains the entire name of each field and each field's attributes.
2. The user can modify the list file and add hints for additional processing. For example, the user can specify the longword and offset information for a field or change its priority. This list file is provided as input to the tool and the tool generates two files as output:

- `dl_meta.uc` file with accessor macros
- `buffer.h` with Intel XScale® core data structures

Note: The two steps in the process are tightly coupled. In Step 2, you cannot change the field name or the results will be unexpected. If changes to the field name are required, you should modify the microblock configuration file and repeat Step 1.

9.2.1 Creating an Application Configuration File

You must create an application configuration file to use as input to the metadata configuration tool.

The application configuration file contains a list of dispatch loop configuration files (see [Section 9.2.2.2](#)).

9.2.1.1 Naming Conventions

There are no restrictions on the name of the application configuration file, however it is recommended that an abbreviated version of the application functionality is used.

The extension for the application configuration file must be `.cfg`.

9.2.1.2 Application Configuration File Contents

The application configuration files consists of a list of dispatch loop configuration files. You may add hints in these configuration files about the priority for the dispatch loop and size of the metadata fields. These hints will help the tool to decide the memory type for the metadata field.

Note: For IXA SDK Release 3.5, only the SRAM memory type is supported.

Example

```
ipv4_fwdr_oc48pos_rx_qm.cfg
```

Hint:

```
ipv4_fwdr_oc48pos_rx_qm
```

```
Priority: 2
```

9.2.2 Creating a Dispatch Loop Configuration File

You must create at least one dispatch loop file to use as input to the metadata configuration tool. Multiple dispatch loop files are allowed. The dispatch loop configuration file contains a list of microblock configuration files (see [Section 9.2.3.2](#)).

Note: The dispatch loop configuration file is not given directly to the tool as input; it is referenced in the application configuration file.

9.2.2.1 Naming Conventions

The file name should describe the application and microblocks which are used. For example, in the case of the IPv4 forwarder OC-48 POS pipeline ingress application, the dispatch loop configuration file name might look like: `ipv4_fwdr_oc48pos_rx_qm.cfg`

The extension for the dispatch loop configuration file must be `.cfg`.

9.2.2.2 Dispatch Loop Configuration File Contents

The dispatch loop configuration file contains the list of microblock configuration files that is used in the pipeline. You may add hints in the configuration files about the priority for the microblocks and size of the metadata fields. These hints will help the tool to decide the memory type for the metadata field.

Example

```
qm_cell_config.cfg
```

```
qm_core_config.cfg
```

```
qm_2800_config.cfg
```

Hint:

```
qm_cell_config
```

```
Priority: 2
```

Hint:


```
qm_2800_config  
Priority: 1
```

9.2.3 Creating a Microblock Configuration File

You must create at least one microblock configuration file to use as input to the metadata configuration tool. Multiple files are allowed.

Note: The microblock configuration file is not given directly to the tool as input; it is referenced in the dispatch loop configuration file.

9.2.3.1 Naming Conventions

The microblock configuration file name should start with <microblock name>_config. For example, for the queue manager qm_cell microblock, the microblock configuration file name is qm_cell_config.cfg.

The extension for the configuration file must be .cfg.

9.2.3.2 Microblock Configuration File Contents

The microblock configuration files contain the metadata registration information for each microblock. The metadata field property information is defined as follows:

- **Field name**
A string of ASCII characters which uniquely identifies the metadata field being referenced. Legal values for field name are listed in [Table 9-1](#). Note that there are additional approved metadata field values; the table lists a subset of the allowed values.
- **Priority**
Priority 0 indicates that the field is frequently used in the critical path. Priority 1 indicates that the field is less frequently used in the critical path. The priority attribute helps the tool decide on the memory location of the field.
- **Field access**
This attribute describes whether the field is read only (R) or written by the microblock (W).
- **Size**
Size of the field, specified in bits.

Each group of properties for a given field is called a record. A record must begin with the field name property. The other properties may be listed in any order after the field name.

Note: Table 9-1 lists a subset of the allowed values for metadata fields.

Table 9-1. Field Name Values

Field Name	Description
Offset	Offset to start of the data
buffer_size	Length of data in the current buffer
header_Type	Type of header at offset bytes into the buffer
free_list	Freelist identifier
rx_stat	Receive status flag
packet_size	Total packet size across multiple buffers
output_port	Output port on the egress processor
input_port	Input port on the ingress processor
nexthop_id	Next hop IP ID
fabric_port	Output port for fabric indicating the blade ID
flow_id	QOS flow id or MPLS label/flow id
class_id	Class ID
sw_next	Pointer to next packet
packet_next	Pointer to next packet

Example

```

Name: buffer_size
Priority: 1
FieldAccess: W
Size: 32

Name: fabric_port
Priority: 1
FieldAccess: W
Size: 8

Name: input_port
Priority: 1
FieldAccess: R
Size: 16

Hint:
buffer_size
longword_offset: 0 0

hint:
input_port
longword_offset: 1 8

```


9.3 List File Overview

The list file is an intermediate file generated by the metadata configuration tool. It specifies information like field name, size, and priority based on the configuration files given as input. You can modify the list file to change field attributes and also add hints about priority, longword, and offset in bits for a field. Once complete, this list file is again given as input to the metadata configuration tool to generate the final `dl_meta.uc` and `buffer.h` files.

Note: You cannot change any field names in the list file or the results will be unexpected. If changes to the field name are required, you must modify the microblock configuration file and re-run the tool.

9.4 `dl_meta.uc` and `buffer.h` File Overview

When the metadata tool is run using a list file as input, the tool generates microcode source files called `dl_meta.uc` and `buffer.h`. The `dl_meta.uc` file consists of get and set macros for each of the metadata fields specified in the microblock configuration files, block macros for dispatch loops, and constants for dispatch loops. The `buffer.h` header file is generated for Intel XScale® core applications. This file consists of a data structure of the newly formed metadata block based on the microblock configuration files.

9.4.1 `dl_meta.uc` File Description

The `dl_meta.uc` file defines a minimal set of metadata fields that satisfies the requirements of the microblock registration defined in the metadata configuration files. All metadata field names, field offsets, and accessor macros are fully defined within `dl_meta.uc`, and are consistent for all microblocks. `dl_meta.uc` also defines block macros, which are generated to read and write large portions of metadata into transfer registers.

The `dl_meta.uc` file is a Workbench source file, which contains all of the metadata field accessor macros. Each metadata field defined in the metadata configuration files contains a corresponding `get_()` and `set_()` macro for accessing the metadata field. Each microblock requiring access to the system metadata must `#include` the `dl_meta.uc` file.

9.4.1.1 Accessor Macros

The accessor macros allow a user to read and write individual fields of the metadata without requiring any knowledge of the field structure or storage location. Each get or set macro reads or writes one field of the metadata structure.

All get macros have a similar structure:

```
#macro dl_meta_get_<fieldname>(xfer or gpr or local)
.begin

.end
#endm
```

All set macros have a similar structure:

```
#macro dl_meta_set_<fieldname>( xfer or gpr or local)
.begin
```



```
.end
#endm
```

9.4.1.2 Block Macros

The `dl_meta.uc` file also contains block macros for allocating transfer registers, and reading/writing multiple-field blocks of metadata at once. The block macros exercise on an entire block.

```
#macro dl_meta_flush_cache
(wxfer_prefix, buf_handle, req_sig, sig_action, START_LW, NUM_LW)

#macro dl_meta_init_cache
(d0, d1, d2, d3, d4, d5, d6, d7)

#macro dl_meta_load_cache (buffer_handle, dl_meta, signal_number, START_LW,
NUM_LW)
```

9.4.1.3 Dispatch Loop Constants

The `dl_meta.uc` file also contains the following dispatch loop constants:

```
<dispatch_loop>_START_LW - indicates the starting longword location
<dispatch_loop>_NUM_LW - indicates the number of longwords for each dispatch loop
```

9.4.1.4 dl_meta.uc Example

The following code fragment shows a portion of a sample `dl_meta.uc` file.

```
#ifndef __DL_METADATA_UC__
define __DL_METADATA_UC__

#include <stdmac.uc>
#include <xbuf.uc>
#include <buf.uc>
// Define the number of LW of the metadata to be cached
#ifndef META_CACHE_SIZE
#define META_CACHE_SIZE 4

#endif
// Use SRAM xfer registers for metadata
#define dl_meta_sram $dl_meta

// Use DRAM xfer registers for metadata
#define dl_meta_dram $$dl_meta

// Use GPR for metadata
.reg dl_meta_gpr[META_CACHE_SIZE]

#ifdef USE_DL_THREAD_ID
.reg dl_exception
#endif
```



```

#define buffer_size_SRAM_OFFSET 0
#define buffer_size_SIZE 32
#define header_type_SRAM_OFFSET 32
#define header_type_SIZE 8
#define port_SRAM_OFFSET 40
#define port_SIZE 4
#define fabric_port_SRAM_OFFSET 44
#define fabric_port_SIZE 8
#define input_port_SRAM_OFFSET 52
#define input_port_SIZE 16

xbuf_alloc($dl_meta, 3, read_write)
/*****
*   dl_meta_get_buffer_size:
*       Description:
*       Outputs:   Value of buffer_size
*       Inputs:    None
*
*       Size:
*       1 instruction
*****/
#define dl_meta_get_buffer_size[buffer_size]
.begin
    xbuf_extract[buffer_size, dl_meta_sram, 0, buffer_size_SRAM_OFFSET,
buffer_size_SIZE]
.end
#endm

/*****
*   dl_meta_set_buffer_size:
*       Description:
*       Outputs:   Value of buffer_size
*       Inputs:    None
*
*       Size:
*       1 instruction
*****/
#define dl_meta_set_buffer_size[buffer_size]
.begin
    xbuf_insert[dl_meta_sram, buffer_size, 0, buffer_size_SRAM_OFFSET,
buffer_size_SIZE]
.end
#endm
.
.
.

```


9.4.2 buffer.h File Description

The `buffer.h` header file is generated for Intel XScale® core applications. This file consists of a data structure of the newly formed metadata block based on the microblock configuration files.

buffer.h Example

```
#if !defined(__BUFFER_H__)
#define __BUFFER_H__
#if defined(__cplusplus)
extern "C"
{
#endif /* end defined(__cplusplus) */
/**
 * DESCRIPTION: This symbol will declare all the fields and the layout of the
 * common
 * part of the buffer meta data. Applications that need extra fields
 * can do
 * so in the following manner without the expense of an extra
 * indirection.
 * typedef struct ix_s_atm_buffer_meta
 * {
 *     IX_DECLARE_HW_BUFFER_META_DATA
 *     ix_uint32      m_CellHeader;
 * } ix_atm_buffer_meta;
 */
#define IX_DECLARE_HW_BUFFER_META_DATA \
{
    /* Longword 0 */
    ix_uint32      m_buffer_next /* 32 bits*/
    /* Longword 1 */
    ix_uint16      m_offset      /* 16 bits*/
    ix_uint16      m_buffer_size /* 16 bits*/
    /* Longword 2 */
    ix_uint8       m_header_Type /* 8 bits*/
    ix_uint8       m_rx_stat     /* 4 bits*/
    ix_uint8       m_free_list   /* 4 bits*/
    ix_uint16      m_packet_size /* 16 bits*/
    /* Longword 3 */
    ix_uint16      m_output_port /* 16 bits*/
    ix_uint16      m_input_port  /* 16 bits*/
    /* Longword 4 */
    ix_uint8       m_nexthop_id_type /* 4 bits*/
    ix_uint8       m_fabric_port    /* 8 bits*/
    ix_uint16      m_nexthop_id     /* 16 bits*/
    /* Longword 5 */
    ix_uint8       m_color          /* 8 bits*/
    ix_uint32      m_flow_id        /* 24 bits*/
    /* Longword 6 */
    ix_uint16      m_class_id       /* 16 bits*/
}
```



```

        /* Longword 7 */
        ix_uint32    m_packet_next    /* 32 bits*/
    }

/**
 * TYPENAME: ix_hw_buffer_meta
 *
 * DESCRIPTION: This structure defines the information and the layout of the
 * common
 *              buffer meta data. If applications will need to add extra fields then
 * they
 *              can do so.
 *
 */
typedef struct ix_s_hw_buffer_meta
{
    IX_DECLARE_HW_BUFFER_META_DATA
} ix_hw_buffer_meta;
#ifdef __cplusplus
}
#endif /* end defined(__cplusplus) */
#endif /* end !defined(__BUFFER_H__) */

```

9.5 Using the Metadata Configuration Tool

9.5.1 System and Software Requirements

- System: Machine with Intel® P3 processor or higher
- Operating system: Windows 2000*, Windows XP*, or Linux*
- Cygwin package should be installed.
- Perl that comes with cygwin should be installed.
Note: This tool may not work with active perl.
- The environment variable PATH should be updated with the path to perl.exe

9.5.2 File Locations

The files that the metadata configuration tool uses and generates have the following characteristics:

- Each configuration file should list the exact full path to the configuration files it uses.
- The tool is located in <IXA_SDK>/src/utilities, where <IXA_SDK> indicates the installation location of the IXA SDK files.
- The metadata tool output files, dl_meta.uc and buffer.h, are generated and stored in the current location from where the tool is executed.

9.5.3 Invoking the Tool

To run the metadata configuration tool, perform the following steps:

1. Open cygwin bash shell.
2. Go to the directory where the tool is located, typically this is: <IXA SDK>/src/utilities, where <IXA SDK> indicates the installation location of the IXA SDK files.
3. Issue the following command:
`./metadatatool.pl -a <ApplicationConfigFile>.cfg`
4. The command in the step above generates an outputted list file with the following name: <ApplicationConfigFile>_list.lst
5. You have the option of modifying the list file at this time.

Note: You cannot change any field names in the list file or the results will be unexpected. If changes to the field name are required, you must modify the microblock configuration file and re-run the tool.

6. Issue the following command:
`metadatatool.pl -l < ApplicationConfigFile >_list.lst`
7. The command in the step above outputs the `buffer.h` and `dl_meta.uc` files.

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 Control 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 Intel XScale® core.
Folding	A software technique used by threads running on the same microengine, 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® 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 Intel XScale® core 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, and so on).
operating system	Acronym for <i>operating system</i> .
OSSL	Acronym for <i>operating system services library</i> .
Operating System Services Library	(OSSL) An operating system abstraction API used within the IXA SDK 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. (For details, see the *Intel® Internet Exchange Architecture Optimized Data Plane Libraries Reference Manual* located on the IXA SDK Tools CD.)

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

RBUF

A receive buffer.

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 <i>Fast Path</i> .
software pipeline	The MEv2 employs a software pipeline model in the fast path processing of packets. There are three different types of pipelines— <i>Context Pipeline</i> , <i>Functional Pipeline</i> , and <i>Mixed Pipeline</i> .
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

TBUF	A transmit buffer.
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 and 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 <i>weighted round robin</i> .

X

XBUF	A transfer buffer. In this manual an XBUF can use a transfer register, general-purpose register, and so on.
------	---

I

Intel XScale [®] core	The ARM architecture core processor in the IXP2400 and IXP2800 network processors.
--------------------------------	--

Y

Z