



Intel® Internet Exchange Architecture Software Building Blocks

Developer's Manual

March 2004



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® 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, 2004.

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	51
1.1	About this Manual.....	51
1.1.1	Microblocks	51
1.1.2	Core Components	52
1.2	Organization of the Manual	52
1.3	Other Sources of Information	52
	Applications	
2	System Data Structures and Design Choices	57
2.1	Buffer Handle	57
2.2	Packet Meta Data (Buffer Descriptor)	58
2.3	Optimizing DRAM Bank Scheduling for Buffers	59
2.4	Buffer Chaining.....	59
2.4.1	Flat Queueing (Cell Based Dequeue)	59
2.4.2	Hierarchical Queueing (Packet Based Dequeue).....	61
2.5	Statistics and Handling of 64-bit Counters	63
2.5.1	Using SRAM/Scratch Atomic Operations	64
2.5.2	Read-Modify-Write in Critical Sections.....	64
2.6	Implementation of Dropping Packets	65
2.7	Global Build Switches.....	65
3	System Data Structures for Packet Replication	67
3.1	Packet Replication.....	67
3.1.1	Buffering Design.....	67
3.1.2	Child Buffer.....	68
3.1.3	Cell Mode	68
3.1.4	Packet Mode	69
3.2	Parent and Child Meta Data (Buffer Descriptor).....	70
3.2.1	Cell Mode	70
3.2.1.1	Child Meta Data	70
3.2.1.2	Parent Meta Data	71
3.2.2	Packet Mode	72
3.2.2.1	Child Meta Data	72
3.2.2.2	Parent Meta Data	73
3.3	Child Buffer Freelist.....	73
3.4	Packet Copier.....	73
3.5	Dropping of Packet Copies.....	74
3.5.1	Dropping Packet Copies in Packet Mode	74
3.5.2	Dropping Packet Copies in Cell Mode	74
3.6	Differentiating Child Buffer from Parent Buffer	74
3.7	Build Switches	75

Microblocks 77

Receive

4	Packet RX Microblock	81
4.1	Overview	81
4.2	Assumptions and Dependencies	81
4.3	Packet RX State Machine	82
4.4	Data Structures	83
4.4.1	Receive Reassembly Context (RXC)	83
4.4.2	Statistics	84
4.4.3	Jump Table	84
4.5	Build Switches	85
4.6	Algorithm	85
4.6.1	Single Microengine Design	85
4.6.2	Two Microengine Design	86
4.7	Flow Chart For Single Microengine Design	87
4.8	Performance Analysis	91
4.8.1	Characterization Data	92
5	CSIX RX Microblock	95
5.1	Overview	95
5.2	Assumptions and Dependencies	95
5.3	Basic Algorithm	96
5.4	Optimizations to the Basic Algorithm	97
5.4.1	Optimizations for the Single C-Frame Packet	97
5.4.2	Optimizing the DRAM Bank Utilization	98
5.5	Extending the Design to Run on Two Microengines	98
5.6	Data Structures	99
5.6.1	Receive Reassembly Context (RXC) for One Microengine Design	99
5.6.2	Receive Reassembly Context (RXC) for Two Microengine Design	99
5.6.3	Lookup Key	100
5.6.4	Statistics	100
5.7	Build Switches	101
5.8	Performance Analysis	101
5.8.1	Characterization Data	102
6	ATM AAL5 RX Microblock	105
6.1	Overview	105
6.1.1	ATM Terminology	105
6.1.2	ATM Layer Functions	106
6.1.3	AAL5-SAR Functions	106
6.1.4	CPCS Functions	106
6.1.5	Design Overview	106
6.1.5.1	OC-48 (SPHY_1_32)	107
6.1.5.2	Quad OC-12 (SPHY_4_8) or OC-24/OC-12 (SPHY_1_32)	107
6.1.5.3	A Maximum of 2048 Ports Addressing	108
6.2	Configuration/Switches	108
6.3	Assumptions, Dependencies and Risks	110
6.4	Data Structures	111
6.4.1	Reassembly Context (RXC) or VC Info table	111
6.4.2	AAL5 RX Counters	113
6.4.3	Hash Table	113
6.4.3.1	Collisions Resolutions	115

6.5	Algorithm and Pseudocode	117
6.5.1	Issues and Challenges in the Implementation.....	117
6.5.2	Algorithm	118
6.5.2.1	Two Microengine Design.....	118
6.5.2.2	Single Microengine Design	120
6.6	Flow Chart (Two Microengine Design)	122
6.7	Performance Analysis	130
6.7.1	Two Microengine Design.....	130
6.7.2	Single Microengine Design (4 threads per OC-12 port)	131
6.8	Resource Usage.....	132
6.8.1	Control store	132
6.8.2	Registers and Signals	132
6.8.3	I/O Commands	133
6.8.4	Characterization Data.....	134
Transmit		
7	CSIX TX Microblock.....	141
7.1	Overview	141
7.2	Assumptions and Dependencies	141
7.3	Data Structures	142
7.3.1	Transmit Context (TxC)	142
7.3.2	Transmit Control Word (TCW).....	142
7.3.3	TM Header—per C-Frame)	143
7.3.4	Packet Header (added only to first c-frame).....	143
7.3.5	Reference CSIX Base Header—One Shortword.....	144
7.3.6	Reference CSIX Unicast Extension Header	144
7.3.7	Statistics	144
7.4	Design	145
7.4.1	Interleaving Multiple Requests	145
7.4.2	Optimizing for the Minimum Packet Case	146
7.4.3	Running the Microblock on Two Microengines.....	146
7.5	Flow Chart for Single Microengine Design	147
7.6	Performance Analysis	150
7.6.1	Characterization Data.....	151
8	Packet TX for SPHY and MPHY-4	155
8.1	Overview	155
8.2	Assumptions and Dependencies	156
8.3	Data Structures	156
8.3.1	Context-Relative Thread Execution Status Flag	156
8.3.2	TX Request—One Longword	157
8.3.3	Queue Structure for Each Port	157
8.3.3.1	Queue Entry Structure	157
8.3.3.2	Queue Descriptor Structure	158
8.3.4	Output Transmit Control Word—Two Longwords	159
8.3.5	Statistics	159
8.4	Build Switches	159
8.5	Design	160
8.6	Flow Chart.....	161
8.7	Performance Analysis	168

	8.7.1	Characterization Data	168
9		Packet TX for MPHY-16	173
	9.1	Overview	173
	9.2	Assumptions and Dependencies	174
	9.3	Data Structures	174
	9.3.1	Globals Stored in Absolute Registers.....	174
	9.3.2	Context Relative Thread Execution Status Flag—exe_stat_flag	174
	9.3.3	TX Request—One Longword	175
	9.3.4	Queue Structure for Each Port.....	175
	9.3.4.1	Queue Entry Structure	175
	9.3.4.2	Queue Descriptor Structure	176
	9.3.5	Output Transmit Control Word —Two Longwords	177
	9.3.6	Statistics.....	177
	9.4	Design	178
	9.5	Flow Chart.....	179
	9.6	Performance Analysis	184
	9.6.1	Characterization Data	184
10		Packet Transmit for OC-192 POS	189
	10.1	Overview	189
	10.2	Assumptions and Dependencies	189
	10.3	Data Structures	190
	10.4	Design	191
	10.5	Performance Analysis	192
	10.5.1	Characterization Data	192
11		ATM AAL5 TX Microblock	197
	11.1	Overview	197
	11.1.1	CPCS Functions.....	197
	11.1.2	AAL5-SAR Functions	198
	11.1.3	ATM Layer Functions	198
	11.1.4	Extended Port Addressing	199
	11.1.4.1	FPGA Addressing	199
	11.1.5	Design Overview	200
	11.1.5.1	OC-48	200
	11.1.5.2	Quad OC-12.....	201
	11.2	Assumptions, Dependencies and Risks	201
	11.3	Data Structures	202
	11.3.1	Packet Metadata	202
	11.3.2	Transmit Context (TxC)	202
	11.3.3	Counters.....	204
	11.3.4	Switches.....	204
	11.4	Algorithm	205
	11.4.1	OC-48—One Page Summary.....	205
	11.4.2	Quad OC-12—One page Summary	206
	11.4.3	OC-48—Second Level of Detail	207
	11.4.4	Quad OC-12—Second Level of Detail	209
	11.5	Performance Analysis	210
	11.5.1	OC-48.....	211
	11.5.2	Quad OC-12.....	211

	11.5.3 Characterization Data.....	212
	11.6 Possible Optimizations	215
12	Packet TX—Multiports Microblock	217
	12.1 Overview	217
	12.2 Assumptions and Dependencies.....	218
	12.3 Data Structures	218
	12.3.1 Globals Stored in Absolute Registers.....	219
	12.3.2 Context Relative Thread Execution Status Flag.....	219
	12.3.3 TX Request—One Long Word	220
	12.3.4 Packet Queue Structure for Each Port	220
	12.3.4.1 Packet Queue Entry for Each Port.....	220
	12.3.4.2 Port Status Info for Each Port—16 Long Words	221
	12.3.5 Output Transmit Control Word—2 Long Words	223
	12.3.6 Statistics	223
	12.4 Design	223
	12.4.1 Picked_from_tx_request.....	224
	12.4.2 Picked_from_virtual_queue_or_picked_none	226
	12.5 Performance Analysis	229
	12.5.1 Performance Analysis for IXP24XX.....	229
	12.5.2 Performance Analysis for IXP28XX.....	230
	12.5.3 Characterization Data.....	230
	Queue Manager	
13	Queue Manager For OC-48	
	Microblock237	
	13.1 Overview	237
	13.2 Assumptions/Dependencies.....	238
	13.3 Data Structures	238
	13.3.1 Queue Descriptor	238
	13.3.2 Packet Counts in Local Memory.....	239
	13.4 Build Switches	239
	13.5 Design	239
	13.6 Differences Between Cell and Packet QM	243
	13.6.1 Q-Array Hardware Configuration.....	243
	13.6.2 Hierarchical Queuing.....	243
	13.6.3 Dequeue Response.....	243
	13.6.4 Multiple Queues on Transmit	243
	13.7 Flow Chart.....	243
	13.8 Performance Analysis	246
14	Queue Manager For OC-192	
	Microblock247	
	14.1 Overview	247
	14.2 Assumptions/Dependencies.....	248
	14.3 Data Structures	248
	14.3.1 Queue Descriptor	248
	14.4 Build Switches	249
	14.5 Design	249
	14.6 Differences between OC-48 and OC-192 QMs.....	251

14.7	Performance Analysis	251
14.7.1	Characterization Data	252
15	Packet Queue Manager Microblock	255
15.1	Overview	255
15.2	Q-Array Hardware Configuration.....	255
15.3	Hierarchical Queuing.....	255
15.4	Dequeue Response	255
15.5	Multiple Queues on Transmit	256
15.6	Performance Analysis	256
15.6.1	Characterization Data	256
16	ATM Queue Manager Microblock	259
16.1	Overview	259
16.2	Assumptions/Dependencies.....	259
16.3	Data Structures	259
16.3.1	Queue Descriptor	260
16.3.2	Packet Counts in Local Memory.....	260
16.4	External Interfaces	261
16.4.1	LLCSNAP Encap and Cell QM.....	261
16.4.2	ATM QM and TM 4.1 Shaper	261
16.4.3	TM 4.1 Scheduler to ATM QM.....	262
16.4.4	ATM QM and AAL-5 TX	262
16.5	Build Switches	263
16.6	Design	263
16.7	Flow Chart.....	266
16.8	Performance Analysis	270
16.8.1	Characterization Data	271
Scheduler Microblocks		
17	Fabric Scheduler For OC-48	277
17.1	Overview	277
17.2	Assumptions, Dependencies and Risks	277
17.3	Data Structures	278
17.3.1	Queue and Queue Groups.....	278
17.3.2	Globals	279
17.3.3	Queue Group Data Structure	279
17.3.4	Queue Data Structure	279
17.4	Design Decomposition	280
17.4.1	Scheduler Thread.....	280
17.4.2	QM Message Handler Thread	280
17.4.3	Flow Control Handler Thread	280
17.4.4	Packets In Flight Handler Thread.....	281
17.5	Flow Chart for Scheduler Thread	282
17.6	Flow Chart for Flow Control Thread	285
17.7	Flow Chart for QM Message Handler Thread	285
17.8	Performance Analysis	287
18	Fabric Scheduler For OC-192	289
18.1	Overview	289

18.2	Assumptions, Dependencies and Risks	289
18.3	Data Structures	289
18.3.1	VoQ Data Structure	290
18.3.2	Globals	290
18.4	Design	291
18.4.1	Flow Control Handling	292
18.4.2	Packets In Flight Handling.....	293
18.5	Implementing a Hierarchical Scheduler.....	294
18.6	Performance Analysis	297
18.6.1	Characterization Data.....	297
19	OC-48 WRR/DRR Packet Scheduler	301
19.1	Overview	301
19.2	Assumptions, Dependencies and Risks	301
19.3	Design Decomposition	302
19.3.1	Scheduler thread	302
19.3.2	QM Message Handler thread	303
19.4	Flow Control	303
19.5	Intel XScale® core Interface	303
19.6	Data Structures	303
19.6.1	Port Specific Data Structure	304
19.6.2	Queue Specific Data Structure	304
19.6.3	Data Structures in Registers	305
19.7	Algorithm and Pseudo Code	305
19.7.1	Issues/Challenges in implementation.....	305
19.7.2	Scheduler Thread.....	306
19.7.3	QM Message Handling thread.....	309
19.7.4	Flow Chart for Scheduler Thread	311
19.7.5	Flow Chart for QM Message Handler Thread	313
19.8	Performance Analysis	315
20	OC-192 DRR Egress Scheduler	317
20.1	Overview	317
20.2	DRR Algorithm	317
20.2.1	Traditional DRR.....	317
20.2.2	The Pre-Sorted DRR algorithm	318
20.3	Assumptions and Dependencies.....	319
20.4	Design Decomposition	320
20.4.1	Enqueuing	320
20.4.2	Dequeuing	321
20.4.3	Class Schedule Block.....	321
20.4.3.1	Interface	321
20.4.3.2	Pseudo Code	322
20.4.4	Count Block	323
20.4.4.1	Interface	323
20.4.4.2	Pseudo Code	323
20.4.5	Port Schedule Block	324
20.4.5.1	Interface	324
20.4.5.2	Pseudo Code	324
20.5	Other Features	326
20.5.1	Flow Control	326

	20.5.2	WRED Support.....	326
20.6		Data Structures	326
	20.6.1	Queue Specific Data Structure.....	326
	20.6.2	Port Specific Data Structure.....	327
	20.6.3	Packets Counter for Each Round.....	328
	20.6.4	Current Round for Each Port.....	328
	20.6.5	Data Structures in Registers	328
20.7		Performance Analysis	329
	20.7.1	Characterization Data	329
21		Egress Queue Manager (DiffServ) Microblock	333
	21.1	Overview	333
	21.2	Assumptions and Dependencies.....	333
	21.3	Microblock Interfaces	333
	21.4	Data Structures	333
	21.5	Flow Chart.....	334
	21.5.1	Synchronization.....	334
	21.5.2	Algorithm	334
	21.6	Micro-code Budget	335
	21.6.1	Performance Analysis	335
	21.6.2	Memory Footprint Analysis.....	336
22		Egress Scheduler (DiffServ) Microblock	337
	22.1	Overview	337
	22.2	Assumptions and Dependencies.....	339
	22.3	Microblock Interfaces	340
	22.3.1	Input Microblock Variables	341
	22.3.2	Output Microblock Variables	341
	22.4	Design Decomposition	342
	22.5	Data Structures	342
	22.6	Flow Chart.....	344
	22.6.1	Inter-thread synchronization.....	344
	22.6.2	QM Message Handler Thread Algorithm.....	345
	22.6.3	Scheduler Thread Algorithm	348
	22.7	Micro-code Budget	351
	22.7.1	Performance Analysis	351
	22.7.2	Memory Footprint Analysis.....	351
23		TM4.1 Shaper and Scheduler	
		Microblock353	
	23.1	ATM TM4.1 overview	353
	23.1.1	What is TM4.1	353
	23.1.1.1	GCRA(T, τ).....	353
	23.1.2	Service Classes in TM4.1.....	354
	23.1.2.1	CBR	354
	23.1.2.2	rtVBR	354
	23.1.2.3	nrtVBR	354
	23.1.2.4	UBR (Plain UBR)	354
	23.1.2.5	UBR with PCR (UBR+)	354
	23.1.2.6	Differentiated UBR	354
	23.1.2.7	GFR	355

	23.1.2.8	ABR.....	355
23.2		Implications of TM4.1	355
23.3		Design Overview	355
	23.3.1	Software Blocks Overview.....	356
	23.3.2	Time Queue Data Structure (TQ).....	357
	23.3.2.1	SRAM Time Queues For Low Bit Rate Traffic	358
	23.3.3	Local Memory Time Queue for High Bit Rate Traffic.....	360
	23.3.3.1	What exactly are high and low bit rate VCs?	361
	23.3.4	TM4.1 Conformance for Low Bit Rate VCs	361
	23.3.4.1	GCRA Conformance	361
	23.3.4.2	Delay conformance	361
23.4		External Interfaces and Communication Data Structures	362
	23.4.1	Interface Between the QM and the GCRA	362
	23.4.2	Interface Between the GCRA and the Write-out/Scheduler	363
	23.4.3	Interface Between the Write-out/Scheduler and the QM	363
23.5		Design Details	364
	23.5.1	GCRA Shaper	364
	23.5.1.1	Data Structures	364
	23.5.1.2	Overview of Operation	365
	23.5.1.3	Shaping Decision for Low Bit Rate VCs.....	365
	23.5.1.4	Shaping Decision for High Bit Rate VCs.....	365
	23.5.1.5	Shaping Macro.....	365
	23.5.1.6	Pseudocode	366
	23.5.1.7	Divide by 53	367
	23.5.2	Write-out Block.....	368
	23.5.3	Data Structures	368
	23.5.3.1	Queues for Low Bit Rate VCQ traffic	368
	23.5.3.2	Queues for High Bit Rate VCQ traffic.....	368
	23.5.4	Overview of Operation.....	369
	23.5.4.1	Processing for Low bit rate VCs.....	369
	23.5.4.2	Processing for high bit rate VCs.....	369
	23.5.5	Write-out block Pseudocode	370
	23.5.6	Scheduler	371
	23.5.6.1	Data Structures	371
	23.5.7	Overview of Operation.....	373
	23.5.7.1	Operations Not on a per Cell Transmission Slot Basis	373
	23.5.8	Pseudocode	374
23.6		Flow Charts for Blocks	377
	23.6.1	GCRA Flow Chart.....	378
	23.6.2	F-GCRA Shaping Macro Flow Chart	379
	23.6.3	Shaper (Macro only) Flow Chart	380
	23.6.4	Write-out Block Flow Chart.....	381
	23.6.5	Scheduler Flow Chart.....	382
	23.6.6	Dequeue Flow Chart	383
23.7		Performance Analysis	384
23.8		2048 Ports Hierarchical Port-rate Shaping.....	384
	23.8.1	Data structures	384
	23.8.1.1	Port State	384
	23.8.1.2	Local Memory Map.....	385
	23.8.1.3	Static Port Schedule.....	386
	23.8.1.4	Local Memory Time queues.....	387

	23.8.2	Write-out Operation	389
	23.8.3	Scheduler Operation	389
23.9		Up to 8 Ports Hierarchical Port-rate Shaping	389
	23.9.1	Overview	389
	23.9.2	Scheduler Data Structures	390
	23.9.2.1	Port Information	390
	23.9.2.2	Local Memory Map	391
	23.9.3	Scheduler Operation	392
23.10		Performance Analysis	393
Forwarder			
24		IPv4 Forwarder Microblock	397
	24.1	Overview	397
	24.2	Assumptions and Dependencies	398
	24.3	Dependencies	399
	24.4	Configuration Options	400
	24.4.1	Build Switches	400
	24.4.2	Default Configuration	401
	24.5	RFC Compliance	402
	24.6	Data Structures	403
	24.6.1	Control Block	403
	24.6.2	Directed Broadcast Table	404
	24.6.3	Next Hop Information	405
	24.6.3.1	Flags	406
	24.6.3.2	Nexthop ID type	406
	24.6.3.3	Nexthop ID	407
	24.6.3.4	Fabric Port ID	407
	24.6.3.5	Output Port ID	407
	24.6.3.6	MTU	407
	24.6.4	Nexthop Database	407
	24.6.5	IPv4 Counters	408
	24.6.6	Route Table	409
	24.7	Longest Prefix Match Lookup	409
	24.7.1	Introduction	409
	24.7.2	Data Structures	409
	24.7.3	Lookup	412
	24.8	Intel® XScale™ Core Interface	414
	24.8.1	Symbols	414
	24.8.2	Exception Codes	414
	24.9	High Level Flow Chart for Microblock	416
	24.10	Performance Analysis	417
	24.10.1	Characterization Data	418
25		IPv6 Forwarder Microblock	419
	25.1	Overview	419
	25.2	Assumptions	419
	25.3	Dependencies	421
	25.4	Configuration Options	421
	25.4.1	Build Switches	421
	25.4.2	Default Configuration	422

25.5	RFC Compliance	423
25.6	Data Structures	423
25.6.1	L3 Next Hop Information	423
25.6.1.1	Valid	424
25.6.1.2	Generic Flags.....	425
25.6.1.3	Blade ID	425
25.6.1.4	Next-Hop ID Type	425
25.6.1.5	Next-Hop ID	425
25.6.1.6	Output Port.....	426
25.6.1.7	MTU	426
25.6.2	Intermediate Next Hop Information	426
25.6.2.1	Status	427
25.6.2.2	Generic Flags.....	427
25.6.2.3	Next-Hop Index1-4	427
25.6.3	Next-Hop Database	427
25.6.4	IPv6 Counters.....	428
25.6.5	Route Table	429
25.7	Longest Prefix Match Lookup	429
25.7.1	Introduction.....	429
25.7.2	Data Structures	430
25.7.2.1	Creating the Route Table	431
25.7.2.2	Route Add Example	432
25.7.3	Route Lookup using Longest Prefix Match (LPM)	432
25.7.4	Lookup Pseudo Code.....	432
25.7.4.1	Optimized Route Lookup Algorithm	433
25.7.4.2	Trie-Cache Example	436
25.7.4.3	Controlled Prefix Expansion.....	437
25.8	Intel XScale® core Interface	438
25.8.1	Symbols.....	438
25.8.2	Exception Codes	439
25.9	High Level Microblock Flow Charts	439
25.10	Performance Analysis	442
25.10.1	Characterization Data.....	444
26	IPv6 To IPv4 Tunneling Microblock	447
26.1	Overview	447
26.2	Assumptions.....	447
26.3	Dependencies	449
26.4	Configuration Options.....	449
26.4.1	Build Switches	449
26.4.2	Default Configuration.....	450
26.5	RFC Compliance	451
26.5.1	Decapsulation.....	451
26.5.2	Encapsulation.....	452
26.6	Statistics Counters	453
26.7	V6V4-Tunnel-Decap Microblock.....	453
26.7.1	Introduction.....	453
26.7.1.1	Header Cache and Metadata Requirements.....	454
26.7.2	Data Structures	455
26.7.2.1	Tunnel Next-Hop information	455
26.7.2.2	Ingress Source List	456

	26.7.2.3	Trie Table Option	458
	26.7.3	Process Flow	458
	26.7.3.1	Tunneling Header Checks	460
	26.7.3.2	Automatic Tunneling Verification	461
	26.7.3.3	6to4 Tunneling Verification	462
	26.7.3.4	Source Address Validation Using the Ingress Source List..	463
26.8		V6V4-Tunnel-Encap Microblock.....	464
	26.8.1	Introduction	464
	26.8.1.1	Header Cache and Metadata Requirements	464
	26.8.2	Data Structures	465
	26.8.2.1	Tunnel Next-Hop Information	465
	26.8.3	Process Flow	466
	26.8.3.1	Obtaining the Tunnel Endpoint Addresses	468
	26.8.3.2	Creating the IPv4 Header	469
26.9		XScale Interface	470
	26.9.1	Symbols	470
	26.9.2	Exception Codes	471
26.10		Performance Analysis	471
27		IPv6 To IPv4 Translation Microblock	473
	27.1	Overview	473
	27.1.1	Traditional (Outbound) NAT-PT	473
	27.1.1.1	Basic NAT-PT	473
	27.1.1.2	NAPT-PT	474
	27.1.2	Two-way (Bi-directional) NAT-PT	474
	27.1.3	Application Level Gateway (ALG)	474
27.2		Assumptions.....	474
	27.2.1	Dependencies	475
	27.2.2	Configuration Options	477
	27.2.2.1	Build Switches.....	477
	27.2.3	Default Configuration	477
	27.2.4	RFC Compliance.....	477
	27.2.5	Statistics Counters	478
27.3		Overview of NAT-PT Microblock	478
27.4		Data Structures	479
	27.4.1	Header Cache and Meta-Data Requirements	479
	27.4.2	Translation Data Structures	480
	27.4.2.1	NAT Table	480
	27.4.2.2	NAPT Table	482
	27.4.3	Process Flow	486
27.5		XScale Core Interface	492
	27.5.1	Symbols	492
	27.5.2	Exception Codes	493
	27.5.3	Performance Analysis	494
	27.5.4	Characterization Data	494
		Layer 2	
28		Layer-2 Decapsulation and Classify	503
	28.1	PPP Decapsulation and Classify	503
	28.2	Ethernet Decapsulation, Classify and Filter	503

28.3	LLCSNAP Decapsulation/Classify.....	505
28.4	Performance Analysis	506
28.4.1	Characterization Data.....	507
29	Layer-2 Encapsulation	513
29.1	PPP Encapsulation.....	513
29.2	Ethernet Encapsulation	513
29.3	LLCSNAP Encapsulation	513
29.4	Performance Analysis	514
29.4.1	Ethernet Encap.....	514
29.4.2	Ethernet Encap Characterization Data.....	515
29.4.3	PPP Encap	517
29.4.4	LLCSNAP Encap.....	520
29.4.4.1	LLCSNAP Encap Characterization Data.....	520
	DiffServ	
30	6-tuple Exact Match Classifier Microblock.....	525
30.1	Overview	525
30.2	Assumptions and Dependencies.....	526
30.2.1	Configuration Options.....	527
30.2.1.1	Build Switches.....	527
30.2.1.2	Default Configuration	527
30.3	Microblock Design	527
30.3.1	Functionality	527
30.3.2	Microblock Interfaces	528
30.3.2.1	Input Microblock Variables	528
30.3.2.2	Output Microblock Variables	528
30.3.2.3	Imported Variables	529
30.3.3	Data Structures	530
30.3.4	Flow Chart.....	532
30.3.4.1	Synchronization.....	532
30.3.4.2	Classification algorithm	532
30.3.4.3	Statistics Gathering	536
30.4	Microcode Budget	537
30.4.1	Performance Analysis	537
30.4.2	Memory Footprint Analysis.....	538
31	Three Color Meter Microblock.....	539
31.1	Overview	539
31.2	Functionality	541
31.3	Assumptions and Dependencies.....	541
31.3.1	Assumptions.....	541
31.3.2	Configuration Options.....	542
31.3.2.1	Build Switches.....	542
31.3.2.2	Default Configuration	542
31.4	Microblock Interfaces	543
31.4.1	Input Microblock Variables	543
31.4.2	Output Microblock Variables	543
31.4.3	Imported Variables	543
31.5	Data Structures	544

31.6	Flow Chart.....	547
31.6.1	Synchronization.....	547
31.6.2	SRTCM Algorithm	549
31.6.2.1	Per-packet Token Updates	550
31.6.2.2	Color-aware Metering	551
31.6.2.3	Statistics Gathering.....	552
31.6.3	TRTCM Algorithm.....	553
31.6.3.1	Per-packet Token Updates	553
31.6.3.2	Color-aware Metering	555
31.6.3.3	Statistics Gathering.....	556
31.7	Micro-code Budget.....	557
31.7.1	Performance Analysis	557
31.7.2	Memory Footprint Analysis.....	558
32	DSCP Marker Microblock.....	561
32.1	Overview	561
32.2	Microblock Interfaces	561
32.2.1	Input Microblock Variables	561
32.2.2	Output Microblock Variables	561
32.3	Flow Chart.....	562
32.3.1	Synchronization.....	562
32.3.2	Marking Algorithm	562
32.4	Micro-code Budget.....	563
32.4.1	Performance Analysis	563
32.4.2	Memory Footprint Analysis.....	564
33	DSCP Classifier Microblock	565
33.1	Overview	565
33.2	Microblock Design.....	565
33.2.1	Functionality	565
33.2.2	Assumptions and Dependencies.....	566
33.2.3	Configuration Options	566
33.2.3.1	Build Switches.....	566
33.2.3.2	Default Configuration	566
33.2.4	Microblock Interfaces	567
33.2.4.1	Input Microblock Variables.....	567
33.2.4.2	Output Microblock Variables	567
33.2.4.3	Imported Variables.....	568
33.2.5	Data Structures	568
33.2.6	Flow Chart.....	570
33.2.6.1	Synchronization	570
33.2.6.2	Classification Algorithm.....	570
33.2.6.3	Statistics Gathering.....	573
33.2.7	Micro-code Budget	574
33.2.7.1	Performance Analysis	574
33.2.7.2	Memory Footprint Analysis	575
34	Weighted Random Early Detection (WRED) Microblock	577
34.1	Overview	577
34.1.1	RED 93.....	577
34.1.1.1	Average queue length.....	578

	34.1.1.2	Packet Drop Probability.....	578
	34.1.2	RED99.....	579
34.2		Functionality	579
34.3		Assumptions and Dependencies.....	580
	34.3.1	Configuration Options.....	580
	34.3.1.1	Build Switches.....	580
	34.3.1.2	Default Configuration	581
34.4		Microblock Interfaces	581
	34.4.1	Input Microblock Variables	581
	34.4.2	Output Microblock Variables	581
	34.4.3	Imported Variables	582
34.5		Data Structures	582
34.6		Flow Chart.....	587
	34.6.1	Synchronization.....	587
	34.6.2	WRED Algorithm	589
	34.6.2.1	WRED for low-speed queues.....	589
	34.6.2.2	WRED for high-speed queues	595
34.7		Micro-code Budget	596
	34.7.1	Micro-code Budget for WRED Low-speed Queues.....	596
	34.7.1.1	Performance Analysis.....	596
	34.7.1.2	Memory Footprint Analysis.....	597
	34.7.2	Micro-code Budget for WRED High-speed Queues	597
	34.7.2.1	Performance Analysis	597
	34.7.2.2	Memory Footprint Analysis.....	598

MPLS Microblocks

35	FTN Forwarder Microblock	601
	35.1	Overview
	35.2	Functionality
	35.3	Assumptions and Dependencies.....
	35.3.1	Assumptions.....
	35.3.2	Component Interfacing
	35.4	Microblock Interfaces
	35.4.1	Input Microblock Variables
	35.4.2	Output Microblock Variables
	35.5	Data Structures and Forwarding Algorithm
	35.5.1	Overview
	35.5.2	NHLFE Table.....
	35.5.3	NHLFE Set Table
	35.5.4	NHLFE Counters Table
	35.5.5	Forwarding Algorithm
	35.5.6	Initial Label Stack Building
	35.5.7	Allocation to Microengines
	35.5.8	Thread Ordering and Synchronization
	35.5.9	FTN Forwarder Micro-code Budget.....
	35.5.9.1	Performance Analysis.....
	35.5.9.2	Memory Footprint Analysis.....
	35.5.10	FTN Forwarder Characterization Data

36	ILM Forwarder Microblock	619
36.1	Overview	619
36.2	Functionality	619
36.3	Assumptions and Dependencies	620
36.3.1	Assumptions	620
36.3.2	Component Dependencies	621
36.4	Microblock Interfaces	621
36.4.1	Input Microblock Variables	621
36.4.2	Output Microblock Variables	622
36.5	Data Structures and Forwarding Algorithm	622
36.5.1	Overview	622
36.5.2	ILM Table	624
36.5.3	ILM_NHLFE Set	625
36.5.4	Input Segment Counters	626
36.5.5	InPort Counters	627
36.5.6	Forwarding Algorithm	627
36.5.7	Allocation to Microengines	633
36.5.8	Thread Ordering and Synchronization	633
36.5.9	ILM Forwarder Micro-code Budget	634
36.5.9.1	Performance Analysis	634
36.5.9.2	Memory Footprint Analysis	635
36.5.10	ILM Forwarder Microblock Characterization Data	636
	Services Microblocks	
37	Packet Copier Microblock	641
37.1	Overview	641
37.2	Requirements	642
37.3	Data Flow	643
37.4	External Interfaces	643
37.4.1	Packet Copier Request Message	644
37.4.2	Packet Copier Response Message	644
37.5	Internal Data Structures	644
37.6	Packet Replication Algorithm	645
37.7	Startup/Shutdown	646
37.8	Performance Analysis	646
38	Freelist Manager	651
38.1	Overview	651
38.2	Assumptions and Dependencies	651
38.3	Algorithm	651
38.3.1	Overview	651
38.3.2	Three-Level Hierarchial Implementation	652
38.3.3	Summary	652
38.4	Data Structures	652
38.5	Performance Analysis	652
	Core Components	657

39	Core Components Overview.....	659
39.1	Overview	659
39.1.1	Functional and Data Flow.....	659
39.1.2	Functional APIs Design Concept.....	661
39.2	APIs for Dynamic Property Updates.....	661
39.2.1	Dynamic Properties and Clients	662
39.2.2	Property Updates API and Data Structures.....	662
39.2.2.1	Properties Data Structure.....	662
39.2.3	Property ID	663
39.2.4	Property API Generic Prototype	663
39.2.5	Current Behavior of Property API.....	663
39.3	Handler Registration.....	664
39.3.1	Support for the IXA Portability Framework and Core Components Infra-structure.....	664
39.3.1.1	Usage of init() and fini() Functions	664
39.3.2	Operating System Independence of Core Components.....	665
39.4	High-Level Overview of the Core Components	666
39.4.1	Applications	666
39.4.1.1	IPv4 Application	667
39.4.1.2	DiffServ Application.....	668
39.4.1.3	MPLS Application.....	669
39.4.1.4	IPv6 Application	670
39.4.2	Building Block Core Components.....	672
39.4.2.1	POS RX.....	672
39.4.2.2	IPv4 Forwarder.....	672
39.4.2.3	Queue Manager (QM)	672
39.4.2.4	Scheduler	673
39.4.2.5	CSIX TX	673
39.4.2.6	CSIX RX.....	674
39.4.2.7	ATM/POS TX	674
39.4.2.8	Ethernet RX.....	674
39.4.2.9	Ethernet TX	675
39.4.2.10	L2 Table Manager	675
39.4.2.11	Route Table Manager	675
39.4.2.12	Six-Tuple Classifier	676
39.4.2.13	Single Rate Three Color Meter	676
39.4.2.14	Weighted Random Early Detection (WRED).....	677
39.4.2.15	Queue Manager for DiffServ	677
39.4.2.16	Scheduler for DiffServ	677
39.4.2.17	IPv6 Forwarder.....	677
39.4.2.18	IPv6 to IPv4 Tunneling	678
39.4.2.19	Route Table Manager for IPv6.....	678
39.4.2.20	Stack Driver.....	679
39.4.2.21	System Application.....	679
39.4.2.22	Message Helper and Support Library	679
39.4.2.23	SoftSAR Core Components	680
39.4.3	MPLS Forwarder Core Component.....	681
40	System Application	683
40.1	Overview	683
40.2	Assumptions.....	683
40.3	Design	684

40.3.1	Initialization Sequence	684
40.3.2	Shutdown and Re-initialization	685
40.3.2.1	Re-initialization.....	686
40.3.2.2	Start and Shut Down API	686
40.3.3	Load Microcode and Start Microengines.....	686
40.3.3.1	Loading Microcode and Starting Microengines API	687
40.3.4	User Initialization/Shutdown Hooks.....	687
40.3.4.1	Initialization/Shutdown Hooks	687
40.3.5	Core Component Infrastructure Initialization	688
40.3.6	Execution Engines	688
40.3.7	Core Components	689
40.3.8	Scratch Rings	690
40.3.9	Property Master.....	691
40.3.9.1	Mastered Properties.....	691
40.3.10	Support Facilities.....	691

Receive Components

41	POS RX Core Component.....	695
41.1	Overview	695
41.2	Assumptions and Dependencies.....	695
41.2.1	Assumptions.....	695
41.2.2	Dependencies	695
41.3	Data flow	696
41.4	Configuration and Initialization	696
41.4.1	Static Configuration Data	696
41.4.2	Patching Symbols and Memory Allocation.....	697
41.5	External API	697
41.5.1	Core Component Infrastructure API	697
41.5.2	Messaging API	697
41.5.3	Library API	698
42	CSIX RX Core Component	699
42.1	Data flow	699
42.2	Assumptions and Dependencies.....	700
42.2.1	Assumptions.....	700
42.2.2	Dependencies	700
42.3	Configuration and Initialization	701
42.4	External API	701
42.4.1	Core Component Infrastructure API	701
42.4.2	Messaging API	701
42.4.3	Library API	702
43	Ethernet RX Core Component.....	703
43.1	Overview	703
43.2	Assumptions and Dependencies.....	703
43.2.1	Assumptions.....	703
43.2.2	Dependencies	703
43.3	Data flow	704
43.3.1	Data Input and Output.....	704
43.3.2	Packet Data Flow	705

43.4	Configuration and Initialization	706
43.4.1	Dynamic Configuration Data	706
43.4.2	Static Configuration Data	706
43.4.3	Patching Symbols.....	707
43.5	Modularity.....	707
43.6	External API	708
43.6.1	Core Component Infrastructure API	708
43.6.2	Messaging API	708
43.6.3	Library API.....	709

Transmit Components

44	CSIX TX Core Component.....	713
44.1	Overview	713
44.2	Data flow	713
44.3	Assumptions and Dependencies.....	714
44.3.1	Assumptions.....	714
44.3.2	Dependencies	714
44.4	Configuration and Initialization	714
44.5	External API	715
44.5.1	Core Component Infrastructure API	715
44.5.2	Messaging API	715
44.5.3	Library API.....	716
45	ATM/POS TX Core Component.....	717
45.1	Overview	717
45.2	Data flow	717
45.3	Assumptions and Dependencies.....	718
45.3.1	Assumptions.....	718
45.3.2	Dependencies	718
45.4	Configuration and Initialization	719
45.4.1	Dynamic Configuration Data	719
45.4.2	Static Configuration Data	720
45.4.3	ATM/POS Media Card Operation Mode	720
45.4.4	Patching Symbols.....	722
45.5	External API	724
45.5.1	Core Component Infrastructure API	724
45.5.2	Messaging API	724
45.5.2.1	Library API	724
46	Ethernet TX Core Component	725
46.1	Overview	725
46.2	Assumptions and Dependencies.....	725
46.2.1	Assumptions.....	725
46.2.2	Component Dependencies.....	726
46.3	Data flow	726
46.3.1	Packet Input and Output.....	726
46.3.2	Packet Data Flow	727
46.4	Configuration and Initialization	728
46.4.1	Dynamic Configuration Data	728
46.4.2	Static Configuration Data	729

	46.4.3	Table Creations	729
	46.4.3.1	Local Interface Table	729
	46.4.3.2	L2 Table	729
	46.4.3.3	ARP Cache	729
	46.4.4	Patching Symbols	730
	46.4.5	Modularity.....	731
46.5		External API	732
	46.5.1	Data Structures	732
	46.5.2	Core Component Infrastructure API.....	732
	46.5.3	Messaging API	733
	46.5.4	Library API	733
47		Ethernet ARP Module	735
	47.1	Assumptions and Dependencies.....	736
	47.1.1	Assumptions.....	736
	47.1.2	Dependencies	736
	47.2	Data flow	737
	47.2.1	ARP Generation Data Flow	737
	47.2.2	ARP Reception Data Flow	738
	47.3	External API	739
	47.3.1	Error Codes	739
	47.4	Modularity.....	740
	47.4.1	External API	741
		Queue Manager Components	
48		Queue Manager Core Component	745
	48.1	Overview	745
	48.2	Configuration and Initialization	745
	48.2.1	Configuration	745
	48.2.2	Configuration Items	746
	48.2.3	Initialization	746
	48.3	Data Flow	747
	48.4	External API	748
	48.4.1	Core Component Infrastructure API	748
	48.4.2	Functional API	748
	48.4.2.1	Messaging API	748
	48.4.2.2	Library API	748
49		Queue Manager (DiffServ) Core Component	749
	49.1	Overview	749
		Scheduler Components	
50		Scheduler Core Component	753
	50.1	Overview	753
	50.2	Configuration and Initialization	754
	50.2.1	Configuration	754
	50.2.2	CSIX Scheduler.....	755
	50.2.3	Packet Scheduler	755
	50.2.4	Configuration Items	755

	50.2.5 Initialization.....	756
	50.3 Core Component Infrastructure API	757
51	Scheduler (DiffServ) Core Component.....	759
	51.1 Overview	759
	Forwarder Components	
52	IPv4 Forwarder Core Component.....	763
	52.1 Overview	763
	52.2 Assumptions and Dependencies	763
	52.2.1 Assumptions	763
	52.2.2 Dependencies	764
	52.3 Configuration and Initialization	764
	52.4 IPv4 Forwarder Core Component Modules.....	766
	52.4.1 ICMP	767
	52.4.1.1 Supported ICMP Error Messages	767
	52.4.1.2 Rate Limiting	768
	52.4.2 Forwarding and IP Header Validation Module.....	769
	52.4.2.1 Forwarding	769
	52.4.2.2 IP Header Validation	769
	52.4.2.3 Identifying Packets for Local Delivery	771
	52.4.3 IP Options Handling	771
	52.4.4 Fragmentation support	772
	52.4.5 Packet Handling Module	773
	52.4.6 Message Handling Module	773
	52.5 External API	773
	52.5.1 Data Structures, Types and Macros.....	773
	52.5.2 Core Component Infrastructure API	774
	52.5.3 Messaging API	774
	52.5.4 Library API.....	775
53	IPv6 Forwarder Core Component.....	777
	53.1 Data Flow	777
	53.2 Assumptions and Dependencies	777
	53.2.1 Assumptions	777
	53.2.2 Component Dependencies	778
	53.3 Configuration and Initialization	778
	53.4 Modularity.....	780
	53.4.1 Forwarding and IP Header validation Module	781
	53.4.1.1 Forwarding	781
	53.4.1.2 IP Header Validation	782
	53.4.1.3 Identifying packets for local delivery	783
	53.4.2 Packet Handling Module	784
	53.4.3 Message Handling Module	784
	53.4.4 ICMPv6.....	784
	53.4.4.1 RFC2463 MUST Features for ICMPv6 Error Message Process- ing.....	785
	53.4.5 Neighbor Discovery	785
	53.4.5.1 Supported Neighbor Discovery Messages	786
	53.4.5.2 Address Resolution	786

	53.4.6	Address Autoconfiguration	786
53.5		External API	787
	53.5.1	Data Structures	787
	53.5.2	Core Component Infrastructure API	787
	53.5.3	Message Helper API	788
	53.5.4	Library API	789
54		IPv6 To IPv4 Tunneling	
		Core Component	791
54.1		Overview	791
54.2		Data Flow	791
54.3		Assumptions and Dependencies	792
	54.3.1	Dependencies	792
54.4		Configuration and Initialization	792
	54.4.1	Configuration Parameters	792
		54.4.1.1 Size of End Tunnel Next Hop Table	792
		54.4.1.2 Size of Start Tunnel Next Hop Table	792
		54.4.1.3 Format of Ingress Source Validation List	792
		54.4.1.4 Size Hint for Ingress Source Validation List	793
	54.4.2	Initialization	793
54.5		Modularity	793
	54.5.1	Initialization Module	794
	54.5.2	Message Handling Module	794
	54.5.3	Packet Handling Module	794
		54.5.3.1 Microblock Packet Handler	795
		54.5.3.2 IPv4 Packet Handler	795
		54.5.3.3 IPv6 Packet Handler	795
	54.5.4	Decapsulation Module	796
	54.5.5	Encapsulation Module	796
	54.5.6	Reassembly Support Module	796
	54.5.7	ICMP Error Message Support Module	796
		54.5.7.1 Packet Too Big Messages	796
		54.5.7.2 Other Error Messages	796
	54.5.8	Tunneling Header Validation Support Module	796
		54.5.8.1 RFC 2893 IPv4 Source Address Checks	797
		54.5.8.2 RFC 2893 IPv6 Source Address Checks	797
		54.5.8.3 RFC 2893 Automatic Tunnel Embedded Address Checks	797
		54.5.8.4 RFC 3056 6to4 Embedded Address Checks	797
	54.5.9	Configuration Support Module	797
54.6		Bindings	797
54.7		External API	798
	54.7.1	Data Structures, Types and Macros	798
	54.7.2	Core Component Infrastructure API	799
	54.7.3	Message Helper API	799
54.8		Library API	801
55		NAT-PT Translation Core Components	803
	55.1	Dependencies	803
	55.2	High-Level Architecture	804
		55.2.1 Main Module	804
		55.2.2 DNS Application Level Gateway	805

55.2.2.1	DNS Name to Address Query from IPv4 Realm	805
55.2.2.2	DNS Name to Address Query from IPv6 Realm	805
55.2.3	FTP Application Level Gateway	805
55.2.3.1	IPv4 FTP Client communicates with IPv6 FTP Server	805
55.2.3.2	IPv6 FTP Client communicates with IPv4 FTP Server	805
55.2.3.3	Packet Header update	806
55.2.4	Specific Design Details	806
55.2.5	NAT-PT Operation	806
55.2.6	NAPT-PT Operation	806
55.2.7	Static Mapping of Addresses	806
55.2.8	Static Port Mapping	807
55.3	Configuration and Initialization	807
55.3.1	Configuration Parameters	807
55.3.1.1	NAT-PT Table Size	807
55.3.1.2	NAT-PT Hash Table Size	807
55.3.1.3	NAT-PT Hash Collision Table Size	807
55.3.1.4	NAT-PT Prefix for IPv6 domain	807
55.3.2	Initialization	808
55.3.2.1	NAT-PT Table Base	808
55.3.2.2	NAT-PT V6V4 Hash Table Base	808
55.3.2.3	NAT-PT V4V6 Hash Table Base	808
55.3.2.4	NAT-PT Hash Collision Table Base	808
55.3.2.5	Blade ID	808
55.3.2.6	48-bit Hash Multiplier	808
55.3.2.7	128-bit Hash Multiplier	808
55.3.2.8	NAT-PT Prefix for IPv6 domain	808
55.4	External API	809
55.4.1	Data Structures, Types and Macros	809
55.4.2	Core Component Infrastructure API	809
55.4.3	Messaging API	810
55.4.4	Library API	811
55.5	Modularity	811
55.5.1	Initialization Module	812
55.5.2	Message Handling Module	812
55.5.3	Packet Handling Module	812
55.5.3.1	Microblock Packet Handler	812
55.5.4	Fragmentation & Reassembly Support Module	813
55.5.5	ICMP Translation Module	813
55.5.6	Translation Module	813
55.5.7	DNS ALG Module	813
55.5.8	FTP ALG Module	813
55.5.9	Packet Sending Support Module	813
55.5.10	Configuration Support Module	814
55.6	Bindings	814

DiffServ Components

56	Six-Tuple Classifier Core Component	817
56.1	Overview	817
56.2	Data and Control Flow	818
56.2.1	Packet Inputs	818

	56.2.2	Packet Outputs.....	818
	56.2.3	Message Inputs.....	819
56.3		Assumptions, Dependencies and Risks.....	819
	56.3.1	Assumptions.....	819
	56.3.2	Dependencies	819
56.4		Configuration and Initialization	820
	56.4.1	Static Configuration Data	820
	56.4.2	Dynamic Configuration Data	822
	56.4.3	Patching Symbols	822
	56.4.4	Initialization and Shutdown Data Flow	822
56.5		External API	823
	56.5.1	Data Structures	823
	56.5.2	Core Component Infrastructure API.....	824
	56.5.3	Message Helper API	824
	56.5.4	Library API	824
56.6		Modularity.....	825
57		Three Color Meter Core Component.....	827
	57.1	Overview	827
	57.2	Assumptions, Dependencies and Risks.....	827
	57.2.1	Assumptions.....	827
	57.2.2	Dependencies	827
57.3		Configuration and Initialization	828
	57.3.1	Static Configuration Data	828
	57.3.2	Dynamic Configuration Data	829
	57.3.3	Patching Symbols	829
	57.3.4	Initialization and Shutdown Data Flow	829
57.4		Data and Control Flow	831
	57.4.1	Packet Inputs	831
	57.4.2	Packet Outputs.....	831
	57.4.3	Message Inputs.....	831
57.5		External API	832
	57.5.1	Data Structures	832
	57.5.2	Core Component Infrastructure API.....	832
	57.5.3	Message Helper API	832
	57.5.4	Library API	833
57.6		Modularity.....	833
58		Weighted Random Early Detection (WRED) Core Component	835
	58.1	Overview	835
	58.2	Assumptions, Dependencies and Risks.....	835
	58.2.1	Assumptions.....	835
	58.2.2	Dependencies	835
58.3		Configuration and Initialization	836
	58.3.1	Static Configuration Data	836
	58.3.2	Dynamic Configuration Data	837
	58.3.3	Patching Symbols	837
	58.3.4	Initialization and Shutdown Data Flow	837
58.4		Data and Control Flow	839
	58.4.1	Packet Inputs	839

	58.4.2	Packet Outputs.....	839
	58.4.3	Message Inputs.....	839
58.5		External API.....	840
	58.5.1	Data Structures.....	840
	58.5.2	Core Component Infrastructure API.....	840
	58.5.3	Message Helper API.....	840
	58.5.4	Library API.....	841
58.6		Modularity.....	841
59		DSCP Classifier Core Component.....	843
	59.1	Overview.....	843
	59.1.1	Data and Control Flow.....	843
		59.1.1.1 Packet Inputs.....	844
		59.1.1.2 Packet Outputs.....	844
		59.1.1.3 Message Inputs.....	844
	59.2	Assumptions, Dependencies and Risks.....	845
		59.2.1 Assumptions.....	845
		59.2.2 Dependencies.....	845
	59.3	Configuration and Initialization.....	846
		59.3.1 Static Configuration Data.....	846
		59.3.2 Dynamic Configuration Data.....	847
		59.3.2.1 Patching Symbols.....	847
		59.3.3 Initialization and Shutdown Data Flow.....	847
	59.4	External API.....	848
		59.4.1 Data Structures.....	848
		59.4.2 Core Component Infrastructure API.....	849
		59.4.3 Message Helper API.....	849
		59.4.4 Library API.....	849
	59.5	Modularity.....	850
		Support Libraries	
60		Route Table Manager.....	855
	60.1	Overview.....	855
	60.2	Usage Model.....	855
		60.2.1 Using Routes and Next Hops.....	855
		60.2.2 Initialization.....	857
		60.2.3 Pre-assigned Next Hop Identifiers.....	857
		60.2.4 Default Route.....	857
	60.3	Design Criteria.....	858
		60.3.1 Data Storage and Retrieval.....	858
		60.3.2 Guaranteed Table Validity.....	858
		60.3.3 Multiple Table Support.....	858
		60.3.4 Control Plane Design.....	859
		60.3.5 Microengine Timing.....	859
		60.3.6 High-level Route Table.....	859
		60.3.7 Lookup Library Initialization.....	859
		60.3.8 Single Direct Client.....	860
		60.3.9 Duplicate Routes.....	860
	60.4	Modularity.....	860
	60.5	External API.....	861

	60.5.1	Data Structures and Types	862
	60.5.2	Macros	862
	60.5.3	Core Component Infrastructure API	862
61		Route Table Manager for IPV6	
		Core Component	865
	61.1	Overview	865
	61.2	Usage Model	865
	61.2.1	Using Routes and Next Hops	865
	61.2.2	Initialization	867
	61.2.3	Pre-assigned Next Hop Identifiers.....	867
	61.2.4	Default Route	867
	61.2.5	Support for equal cost next hops	867
	61.3	Design Criteria	867
	61.3.1	Data Storage and Retrieval.....	868
	61.3.2	Guaranteed Table Validity.....	868
	61.3.3	Multiple Table Support	868
	61.3.4	Control Plane Design	868
	61.3.5	High-level Route Table.....	868
	61.3.6	Lookup Library Initialization.....	868
	61.3.7	Single Direct Client.....	869
	61.3.8	Duplicate Routes.....	869
	61.4	External API	869
	61.4.1	Data Structures, Types, and Macros.....	869
	61.5	Core Component Infrastructure API.....	870
62		L2 Table Manager	871
	62.1	Design Considerations	871
	62.2	L2 Table Entries	871
	62.3	Data Flow and Behavior.....	873
	62.4	Multi-Client Support.....	874
	62.5	Microblock Synchronization	874
	62.6	Initialization and Usage	874
	62.7	Modularity.....	875
	62.8	External API	876
	62.8.1	Data Structures, Types, and Macros.....	876
	62.8.2	Library Functions.....	876
63		Message Helper and Support Library	879
	63.1	Overview	879
	63.2	Assumptions.....	879
	63.3	Overview	879
	63.3.1	Message Helper Library	881
	63.3.2	Support Library.....	882
	63.4	Usage.....	882
	63.4.1	Client Usage.....	882
	63.4.1.1	Example	883
	63.4.2	Core Component Usage	884
	63.5	Message Support Library Internal Design.....	885
	63.5.1	Core Component Infrastructure.....	885

	63.5.1.1	Initialization	885
	63.5.1.2	Shutdown	885
	63.5.1.3	Call-Table	886
63.6	Data Flow		888
	63.6.1	Asynchronous Call: Data Flow	888
	63.6.2	Synchronous Call: Data Flow	890
63.7	Resource Manager Only System		891
	63.7.1	Initialization	891
	63.7.2	Internal Operation	891
	63.7.3	Asynchronous and Fire and Forget	892
	63.7.4	Synchronous	892
63.8	Message Support Library API		892

Stack Driver Component

64	Stack Driver	895
64.1	Overview	895
64.2	Assumptions and Dependencies	895
	64.2.1 Assumptions	895
	64.2.2 Dependencies	895
64.3	Stack Driver	896
	64.3.1 Stack Driver Design	896
	64.3.1.1 Stack Driver Design	897
	64.3.1.2 Packet Flow	898
	64.3.1.3 Synchronizing Properties	898
	64.3.2 Design of the Core Component Module	899
	64.3.2.1 Core Component Module Design	899
	64.3.2.2 Execution Context	900
	64.3.3 External API	900
	64.3.3.1 Core Component Module Data Structures and Types	900
	64.3.3.2 Core Component Infrastructure API	901
	64.3.3.3 Core Component Infrastructure Separation	901
	64.3.3.4 Initialization	902
	64.3.3.5 Shutdown	902
	64.3.3.6 Packet and Message Processing API	903
	64.3.3.7 Properties API	903
	64.3.4 Packet Classifier Design	903
	64.3.4.1 Theory of Operation	903
	64.3.4.2 Packet Classifier Design	904
	64.3.4.3 Packet Classifier Data Structures	904
	64.3.4.4 Packet Classifier External API	905
	64.3.4.5 Packet Classifier Data Flow	905
	64.3.5 Outgoing Packet Classifier Design	906
	64.3.5.1 Outgoing Packet Classifier Data Structures	906
	64.3.5.2 Outgoing Packet Classifier Internal API	906
	64.3.5.3 Outgoing Packet Classifier Data Flow	907
	64.3.5.4 Outgoing Packet Classifier Design Scalability	907
	64.3.6 VIDD for VxWorks	908
	64.3.6.1 VIDD System Data Structures	909
	64.3.6.2 VIDD Local Data Structures	909
	64.3.7 MUX Interface API	909
	64.3.7.1 VIDD System Function Calls	910

64.3.7.2	MUX APIs used by the VIDD	911
64.3.8	Packet Processing—VxWorks Example.....	911
64.3.8.1	Core Component Module Side.....	911
64.3.8.2	VIDD Side	912
64.3.8.3	Outgoing Packet Processing Pseudocode.....	912
64.3.8.4	Pseudocode for Outgoing Packets—in the Core Component Module.....	913
64.3.9	VIDD for Linux*	913
64.3.9.1	VIDD System Data Structures for Linux.....	913
64.3.9.2	VIDD System API for Linux.....	913
64.3.9.3	VIDD Linux Driver Support API.....	914
64.3.10	Transport Module Design.....	914
64.3.10.1	Transport Module Data Structures.....	914
64.3.10.2	Transport Module External API.....	914
64.3.11	Start-up Configuration File Requirements	916

SoftSAR Components

65	SoftSAR Core Components	919
65.1	Architecture Overview	919
65.1.1	Hardware Architecture Overview.....	919
65.1.2	General Software Architecture Overview	920
65.1.3	ATM Building Blocks	921
65.1.3.1	Microblocks Overview	921
65.1.3.2	ATM Core Components Overview	922
65.2	Functionality	923
65.2.1	Handle and Handle Manager Concept.....	923
65.3	ATM Objects Control.....	923
65.4	Plug-in Core Components	924
65.4.1	Plug-in Registering.....	924
65.5	Support for Single/Dual IXP Hardware Configuration	925
65.6	Multi-Instances Support	926
65.7	SoftSAR Core Components	927
65.7.1	Dynamic Behavior	928
65.7.2	SAR Control Main Core Component	930
65.7.2.1	Functionality	930
65.7.2.2	Assumptions and Dependencies	930
65.7.2.3	Decomposition	931
65.7.2.4	Data and Control Flow	931
65.7.2.5	Configuration and Initialization	933
65.7.3	External API	933
65.7.3.1	Data Structures	933
65.7.3.2	Core Component Infrastructure API.....	934
65.7.3.3	Messaging API.....	934
65.7.3.4	Library API	935
65.7.3.5	Plug-in API	935
65.8	SAR Control Agent Core Component Design	935
65.8.1	Assumptions and Dependencies.....	936
65.8.1.1	Assumptions	936
65.8.1.2	Dependencies	936
65.8.2	Decomposition	936
65.8.3	Data and Control Flow	937

	65.8.4 Configuration and Initialization	937
65.9	ATM RX Core Component	938
	65.9.1 Assumptions and Dependencies	938
	65.9.1.1 Assumptions	938
	65.9.1.2 Dependencies	938
	65.9.2 Shared Data Structures	939
	65.9.3 Data flow	939
	65.9.3.1 Data Input and Output	939
	65.9.3.2 Synchronization	939
	65.9.4 Configuration and Initialization	940
	65.9.4.1 Static Configuration Data	940
	65.9.4.2 Hash Mechanism for VC Searching	941
	65.9.5 Startup/Shutdown	941
	65.9.6 External API	941
	65.9.6.1 Core Component Infrastructure API	942
65.10	ATM TX Core Component	942
	65.10.1 Data flow	942
	65.10.2 Assumptions and Dependencies	942
	65.10.3 Assumptions	942
	65.10.4 Dependencies	943
	65.10.5 Configuration and Initialization	943
	65.10.5.1 Dynamic Configuration Data	944
	65.10.6 Static Configuration Data	944
	65.10.7 Startup/Shutdown	944
	65.10.8 External API	944
	65.10.8.1 Core Component Infrastructure API	944
65.11	TM4.1 Core Component	945
	65.11.1 Assumptions and Dependencies	945
	65.11.2 Shared Data Structures	945
	65.11.3 Data and Control Flow	946
	65.11.3.1 Support for Port Shaping Table	946
	65.11.3.2 Support for HBR VC	948
	65.11.4 Configuration and Initialization	948
	65.11.5 Startup/Shutdown Discussion	948
	65.11.6 External API	949
	65.11.6.1 Core Component Infrastructure API	949

MPLS Components

66	MPLS Forwarder Core Component	953
	66.1 Overview	953
	66.2 Data Flow	953
	66.3 Assumptions and Dependencies	954
	66.3.1 Assumptions	954
	66.3.2 Component Dependencies	955
	66.3.3 IPv4 and MPLS Core Component Cooperation	955
	66.3.4 Configuration and Initialization	956
	66.3.4.1 Static Configuration Data	956
	66.3.4.2 Dynamic Configuration Data	956
	66.3.4.3 Patching Symbols	956
	66.4 Modularity	958

66.5	External API	959
66.5.1	Data Structures and Types	960
66.5.2	Core Component Infrastructure API	960
66.5.3	Message Helper API	961
66.5.4	Library API	961
A	Network Simulator	963
A.1	Overview	963
A.2	Architecture	963
A.2.1	Connecting NetSim to the IXP Workbench	965
A.2.2	Assigning NetSim to an IXP Simulation Port.....	965
A.3	Initializing NetSim.....	966
A.4	NetSim Workbench Script Reference	967
A.4.1	void netsim_verbose(int errorlevel)	967
A.4.2	void netsim_load_tcs(char* tcs_filename).....	967
A.4.3	void netsim_start(void)	967
A.5	XML Basics	967
A.5.1	XML Documents.....	968
A.5.2	Node Element	968
A.5.3	Node Attributes	968
A.5.4	Empty Node	969
A.5.5	Document Type Definition (*.DTD).....	969
A.5.5.1	Defining Elements in the DTD	969
A.5.5.2	Defining Attributes in the DTD.....	971
A.5.5.3	Importing DTDs from an External File	972
A.5.5.4	Importing DTDs from Multiple Files	973
A.5.5.5	Defining Constant Values Using ENTITY Definitions	973
A.6	NetSim TCS and DTD Configuration	974
A.6.1	TCS File Structure for NetSim.....	974
A.6.1.1	Sections of TCS File.....	974
A.6.2	Creating and Assigning Streams to a Device/Port	975
A.6.2.1	The Device and Port Attributes	975
A.7	Miscellaneous Features	978
A.7.1	Inducing Frame/Cell Errors	978
A.7.2	File Logging.....	979
A.8	Traffic Capture with Validation	982
A.8.1	TCS Configuration for Capture with Validation.....	982
B	AAL-2 Receive Microblock	985
B.1	Overview	985
B.1.1	AAL-2 CPS.....	986
B.1.2	AAL-2 SSCS	986
B.1.3	Design Overview	986
B.1.3.1	Packet Sequencing through Thread Synchronization	986
B.1.3.2	Buffer Freelist	986
B.1.3.3	CPS and SSSAR Processing	987
B.2	Assumptions.....	987
B.3	Configuration Options	987
B.4	Data Structures	988
B.4.1	AAL-2 CPS Receive Context (CPS RXC) Table	988
B.4.2	CID Receive Context (CID RXC) Table.....	988

	B.4.3	Hash Tables	988
	B.4.4	CRC-5 Tables.....	990
	B.4.5	Counters.....	990
	B.4.6	Local Memory Queue	990
B.5		Algorithm	991
	B.5.1	Cell RX	991
	B.5.2	Thread Locking and Cell Queuing.....	991
	B.5.3	CPS/SSSAR Processing.....	993
	B.5.4	Performance Analysis	997
C		AAL-2 Transmit Microblock.....	999
	C.1	Overview	999
		C.1.1 Design Overview	1001
		C.1.1.1. Packet Sequencing through Thread Synchronization	1001
		C.1.1.2. SSSAR/CPS SDU Processing	1004
		C.1.1.3. SSSAR Sublayer Functions	1005
		C.1.1.4. CPS Sublayer Functions	1006
		C.1.1.5. Timer CU Handling.....	1010
	C.2	Configuration/Switches.....	1013
		C.2.1 Assumptions, Dependencies and Risks	1013
	C.3	Data Structures	1013
		C.3.1 AAL2 Transmit VC Context	1014
		C.3.2 CRC-5 Tables.....	1015
		C.3.3 Timer CU Data Structure.....	1015
		C.3.4 Counters.....	1016
	C.4	Performance Analysis	1016
D		Glossary	1017

Figures

2-1	Flat Queueing on Ingress IXP2400 (single packet).....	60
2-2	Flat Queueing on Ingress IXP2400 (multiple packets).....	61
2-3	Hierarchical Queueing on Egress IXP2400 (single packet)	62
2-4	Hierarchical queueing on the Egress IXP2400 (Multiple Packets)	63
3-1	Packet Replication Example.....	68
3-2	Packet Replication in Cell Mode.....	69
3-3	Packet Replication in Packet Mode.....	70
3-4	RX Status Field Definition	75
4-1	Packet RX State Machine	82
4-2	Packet Receive on Two Microengines	86
4-3	Design for the First Receive Microengine	86
4-4	Design for the First Receive Microengine	87
4-5	POS Receive Flowchart: Page 1 of 3.....	88
4-6	POS Receive Flowchart: Page 2 of 3.....	89
4-7	POS Receive Flowchart: Page 3 of 3.....	90
6-1	Structure of Sub-layers in AAL5.....	105
6-2	Three Stage Functional Pipeline for Two Microengine AAL5 RX Design.....	107
6-3	Four Stages Running on Two Microengines Independently.....	107

6-4	A Maximum of 2048 Ports Addressing Scheme.....	108
6-5	Collisions Resolutions	116
6-6	IXP2000 ATM RX Flow Chart	122
6-7	Pipe Stage 1 Start: Phase 2 Flow Chart	123
6-8	SOP and EOP Flow Chart.....	124
6-9	EOP Only Flow Chart (Page 1 of 2)	125
6-10	EOP Only Flow Chart (Page 2 of 2)	126
6-11	MOP Flow Chart.....	127
6-12	SOP Only Flow Chart	128
6-13	Pipe Stage 3: Phase 6 Flow Chart	129
7-1	CSIX TX Microblock Running on Two Microengines	146
7-2	CSIX Transmit: Flowchart for Phase 1 of Two-Phase Scheme	148
7-3	CSIX Transmit: Flowchart for Phase 2 of Two-Phase Scheme	149
8-1	SPHY Packet TX Flow Chart: Part 1 of 6	162
8-2	SPHY Packet TX Flow Chart: Part 2 of 6	163
8-3	SPHY Packet TX Flow Chart: Part 3 of 6	164
8-4	SPHY Packet TX Flow Chart: Part 4 of 6	165
8-5	SPHY Packet TX Flow Chart: Part 5 of 6	166
8-6	SPHY Packet TX Flow Chart: Part 6 of 6	167
9-1	Packet Transmit—MPHY-16 Configuration—Flowchart Page 1 of 5	179
9-2	Packet Transmit—MPHY-16 Configuration—Flowchart Page 2 of 5	180
9-3	Packet Transmit—MPHY-16 Configuration—Flowchart Page 3 of 5	181
9-4	Packet Transmit—MPHY-16 Configuration—Flowchart Page 4 of 5	182
9-5	Packet Transmit—MPHY-16 Configuration—Flowchart Page 5 of 5	183
10-1	High-level Design for the First Microengine	191
10-2	High-level Design for the Second Microengine	191
11-1	Format of CPCS-PDU	197
11-2	Format of the 53-byte ATM Cell	198
11-3	Extended Port Addressing	199
11-4	Format of the FPGA Header	199
11-5	Two Stage Functional Pipeline for OC-48	200
11-6	Two MEs Running Independently	201
12-1	picked_from_tx_request.....	225
12-2	picked_from_virtual_queue_or_picked_none Path	227
13-1	Format of Queue Descriptor.....	238
13-2	Ingress Queue Manager: Phase Initialize	240
13-3	Ingress Queue Manager: Phase 1	241
13-4	Ingress Queue Manager: Phase 2	242
13-5	Ingress Queue Manager: Phase 3	242
13-6	Ingress Queue Manager: Flow Chart Page 1 of 2.....	244
13-7	Ingress Queue Manager: Flow Chart Page 2 of 2.....	245
14-1	QM OC-192 Format of Queue Descriptor	248
14-2	Basic Functionality of the Queue Manager	250
16-1	Format of Queue Descriptor.....	260
16-2	Initialize	264
16-3	Phase 1	264
16-4	Phase 2	265
16-5	Phase 3	265
16-6	Queue Manager Operations Algorithm (Page 1 of 5).....	266
16-7	Queue Manager Operations Algorithm (Page 2 of 5).....	267

16-8	Queue Manager Operations Algorithm (Page 3 of 5).....	268
16-9	Queue Manager Operations Algorithm (Page 4 of 5).....	269
16-10	Queue Manager Operations Algorithm (Page 5 of 5).....	270
17-1	Hierarchical Bit Vector for Ingress Scheduler.....	278
17-2	Threads in the CSIX scheduler	280
17-3	Format of a CSIX Flow Control Frame	281
17-4	Scheduler Thread: Flowchart Page 1 of 2.....	283
17-5	Scheduler Thread: Flowchart Page 2 of 2.....	284
17-6	Flow Control Thread: Flowchart	285
17-7	QM Message Handler Thread: Flowchart	286
18-1	High Level Algorithm for the Scheduler	291
18-2	Processing of Enqueue Requests	291
18-3	Processing of Dequeue Requests.....	292
18-4	Format of a CSIX Flow Control Frame	293
18-5	Implementing a Hierarchal Scheduler (1 of 2).....	295
18-6	Implementing a Hierarchal Scheduler (2 of 2).....	296
19-1	High Level Components in the Egress Scheduler.....	302
19-2	Scheduler Thread: Flowchart Page 1 of 2.....	311
19-3	Scheduler Thread: Flowchart Page 2 of 2.....	312
19-4	Queue Manager Message Handler Thread: Flowchart Page 1 of 2.....	313
19-5	Queue Manager Message Handler Thread: Flowchart Page 2 of 2.....	314
20-1	High Level Components in the Egress Scheduler.....	320
21-1	Queue Data Structures in SRAM (maintained by Queue Manager).....	334
22-1	Existing implementation of WRR/DRR egress scheduler.....	337
22-2	Enhanced Egress Scheduler with Hierarchic WRR/SP/DRR	338
22-3	Interfaces between Scheduler and Collaborating Microengines	340
22-4	Scheduler Microblock Decomposition	342
22-5	Common Data Structures, Shared between Two Scheduler Threads	343
22-6	Synchronization between scheduler threads.....	345
22-7	QM Message Handler Thread	347
22-8	Scheduler Main Loop	348
22-9	Queue Group Scheduling (Strict Priority)	349
22-10	Queue Scheduling (Deficit Round Robin)	350
23-1	GCRA(T,t) on the Arrival of a New Cell.....	353
23-2	TM4.1 Architecture overview.....	356
23-3	Time Queues concept	357
23-4	Time Queue Data structure in SRAM for Low Bit Rate VCs	358
23-5	Time Queue in Local Memory for Very High Bit Rate VCs.....	360
23-6	External Interfaces to TM 4.1 Blocks.....	362
23-7	GCRA Flow Chart.....	378
23-8	F-GCRA Shaping Macro Flow Chart	379
23-9	Shaper (Macro only) Flow Chart	380
23-10	Write-out Block Flow Chart.....	381
23-11	Scheduler Flow Chart.....	382
23-12	Dequeue Flow Chart	383
23-13	Sample Port Shaping Table	387
23-14	Correlation between the PortShaping table and the HBR TQ table	388
24-1	IPv4 Microblock Dependencies	400
24-2	Directed Broadcast Table Layout.....	404
24-3	Nextthop Database Layout.....	408

24-4	Route Table Lookup	410
24-5	Trie Table with One Route	411
24-6	Longest Prefix Match Dual Lookup	412
24-7	Trie Table with Two Routes	413
24-8	High Level Flow Chart for IPv4 Microblock	416
24-9	IPv4 Microblock Processing Nexthop Information	417
25-1	IPv6 Microblock Dependencies	421
25-2	Structure of a <i>Real</i> Next-Hop Entry.....	424
25-3	Structure of an Intermediate Next-Hop Entry	426
25-4	Next-Hop Database Layout.....	427
25-5	Structure of an IPv6 Address	430
25-6	Route Table Data Structures—Trie-Blocks and Trie-Entries	430
25-7	Route Table Data Structures—Multi-Way Tree which Trie-Blocks	431
25-8	Representation of a 40-Bit Route Entry 0x3fff020304 Using Trie-Blocks	432
25-9	Pseudocode for the Basic Lookup Algorithm.	433
25-10	Flow Chart for the Lookup Algorithm.....	435
25-11	Route Tables with Prefix Optimization	436
25-12	Trie Table Representation for Routing Prefix FFFE:201::/32.....	437
25-13	Trie Table Representation for Routing Prefix FFFE:200::/32.....	438
25-14	IPv6 Forwarder Microblock Flowchart.....	440
25-15	IPv6 Header Validation Flowchart.....	441
25-16	IPv6 Microblock Processing Next-Hop Information Flowchart.....	442
26-1	IPv6-IPv4 Tunneling Microblock Dependencies.....	449
26-2	Ingress Source List Blocks.....	457
26-3	Process Flow for V6V4-Tunnel-Decap Microblock.....	459
26-4	Process Flow for V6V4-Tunnel-Decap Header Validation	460
26-5	Process Flow for V6V4-Tunnel-Decap Validation for Automatic Tunneling	461
26-6	Process Flow for V6V4-Tunnel-Decap Validation for 6to4 Tunneling	462
26-7	Process Flow for Tunnel Ingress Source Validation	463
26-8	Process Flow for V6V4-Tunnel-Encap Microblock.....	467
26-9	Obtaining the IPv4 Tunnel Endpoint Addresses	469
27-1	Dependencies of the IPv6 Translation Microblock and Related Modules	476
27-2	NAT State.....	481
27-3	NAPT Table and Indexes Used for Access.....	484
27-4	Process Flow for NAT-PT Microblock	487
27-5	IPv4 to IPv6 Translation (Page 1 of 2)	489
27-6	IPv4 to IPv6 Translation (Page 2 of 2)	490
27-7	IPv6 to IPv4 Translation (Page 1 of 2)	491
27-8	IPv6 to IPv4 Translation (Page 2 of 2)	492
28-1	Hash Table Entry for MAC Filtering	503
28-2	Header Type Field of Packet Metadata.....	506
30-1	Hash Table and Statistics Table Organization	530
30-2	Hash Entry Layout.....	531
30-3	6-tuple Classifier Algorithm (Page 1 of 3)	534
30-4	6-tuple Classifier Algorithm (Page 1 of 3)	535
30-5	6-tuple Classifier Algorithm (Page 3 of 3)	536
31-1	Color-aware and Color-blind Modes of SRTCM.....	540
31-2	Color-aware and Color-blind Modes of TRTCM.....	541
31-3	TCM Data Structures with Statistics.....	544
31-4	TCM Data Structures without Statistics.....	545

31-5	SRTC Meter Data Structures in Local Memory	545
31-6	TCM Inter-thread Synchronization (Page 1 of 2).....	548
31-7	TCM Inter-thread Synchronization (Page 2 of 2).....	549
31-8	SRTCM Token Update Macro	550
31-9	SRTC Metering Macro.....	552
31-10	TRTCM Token Update Macro	554
31-11	TRTC Metering Macro.....	556
32-1	DSCP Marking Algorithm	563
33-1	DSCP Rule Table Organization.....	568
33-2	Classification Result and Statistics Entry Layout	569
33-3	DSCP Classifier Algorithm (Page 1 of 3).....	571
33-4	DSCP Classifier Algorithm (Page 2 of 3).....	572
33-5	DSCP Classifier Algorithm (Page 3 of 3).....	573
34-1	RED Dropping Function	578
34-2	WRED Data Structures in SRAM for Low-speed Queues.....	583
34-3	WRED Data Structures in SRAM for High-speed Queues	584
34-4	Local Memory Data Structures for Low-speed Queues	585
34-5	Local Memory Data Structures for High-speed Queues.....	586
34-6	WRED Inter-thread Synchronization	588
34-7	WRED Algorithm for Low-speed Queues.....	590
34-8	WRED algorithm - phase 1.....	591
34-9	WRED algorithm - phase 2.....	594
34-10	WRED Algorithm for High-speed Queues	595
35-1	FTN Forwarder Data Structures	605
35-2	FTN Forwarder Forwarding Algorithm - Phase 1	609
35-3	FTN Forwarder Forwarding Algorithm - Phase 2	610
35-4	MPLS Stack Entry Format.....	611
35-5	IP and MPLS Microblocks on the Same Microengines	612
35-6	FTN Forwarder Thread Synchronization	613
36-1	ILM Forwarder Data Structures	623
36-2	ILM Forwarder Algorithm - Phase 1	628
36-3	ILM Forwarder Algorithm - Phase 2 (Page 1 of 4).....	629
36-4	ILM Forwarder Algorithm - Phase 2 (Page2 of 4).....	630
36-5	ILM Forwarder Algorithm - Phase 2 (Page 3 of 4).....	631
36-6	ILM Forwarder Algorithm - Phase 2 (Page 4 of 4).....	632
36-7	ILM Reserved Label Range Processing.....	633
37-1	Data Flow through Packet Copier	643
39-1	POS IPv4 Software Architecture	667
39-2	DiffServ Application Overview	668
39-3	MPLS Application Overview	669
39-4	Software Components for IPv4/IPv6 Forwarding and V6/V4 Tunneling.....	671
40-1	System Application Overview	684
40-2	Execution Engine Repository Structure.....	689
40-3	Core Component Repository Structure	690
42-1	CSIX RX Core Component.....	699
43-1	Ethernet RX Component Dependencies	703
43-2	Ethernet RX Core Component Data Flow	705
43-3	Modularity of Ethernet RX Core Component.....	707
44-1	CSIX TX Core Component	713
45-1	ATM/POS TX Core Components	718

46-1	Ethernet TX Component Dependencies	725
46-2	Ethernet TX Core Component Data Flow	727
46-3	Modularity of Ethernet TX Core Component	731
47-1	Ethernet ARP Component Dependencies	736
47-2	Ethernet ARP Generation Data Flow	737
47-3	Ethernet ARP Reception Data Flow	738
47-4	Modularity of the Ethernet ARP Module	740
48-1	Initialization Flow of the Queue Manager Core Componen	747
50-1	Scheduler Core Component Data Flow	754
50-2	Scheduler Core Component Initialization Data Flow	757
52-1	IPv4 Forwarder Core Component Modules	767
52-2	IIPv4 Forwarder Core Component Bindings	771
53-1	IPV6 Forwarder Core Component Dependencies	778
53-2	Modularity of IPv6 Core Component	781
53-3	IPV6 Forwarder Core Component Bindings	783
54-1	Tunneling Core Component Dependencies	792
54-2	Modularity of Tunneling Core Component	794
54-3	Tunneling Core Component Bindings	798
55-1	Translation Core Component Dependencies	804
55-2	Translation Core Component Architecture	804
55-1	Modularity of Translation Core Component	812
55-2	Translation Core Component Bindings	814
56-1	Data Flow of 6-tuple Classifier Core Component	818
56-2	6-tuple Classifier Core Component Dependencies	820
56-3	Initialization of a 6-tuple Classifier Core Component	822
56-4	Shutdown of a 6-tuple Classifier Core Component	823
56-5	6-Tuple Classifier Core Component Achitecture	826
57-1	TCM Core Component Dependencies	828
57-2	Initialization of TCM Core Component	829
57-3	Shutdown of TCM Core Component	830
57-4	Data flow of the TCM Core Component	831
57-5	TCM Core Component Architecture	834
58-1	WRED Core Component Dependencies	836
58-2	Initialization of the WRED Core Component	837
58-3	Shutdown of the WRED Core Component	838
58-4	Data flow of the WRED Core Component	839
58-1	WRED Core Component Architecture	842
59-1	Data flow of DSCP Classifier Core Component	843
59-2	DSCP Classifier Core Component Dependencies	845
59-3	Initialization of a DSCP Classifier Core Component	847
59-4	Shutdown of a DSCP Classifier Core Component	848
59-5	SRTCM Core Component Architecture	851
60-1	Adding Information using the Route Table Manager	856
60-2	Routes and Next Hops	856
60-3	Route Table Manager Initialization	857
60-4	Modularity of the Route Table Manager	860
60-5	Route Table Manager Initialization	861
61-1	IPv6 RTM—Adding Information	866
61-2	Routes and Next Hops—IPV6 RTM	866
61-3	RTMv6 Initialization	867

62-1	L2 Table Manager Entry State Diagram	872
62-2	L2 Table Manager in IXA SDK 3.1	873
62-3	L2 Table Manager Usage	875
63-1	Messaging Overview	880
63-2	Asynchronous Call: Data Flow	888
63-3	Synchronous Call: Data Flow	890
64-1	Stack Driver Packet Flow	896
64-2	Stack Driver Modularity	898
64-3	Stack Driver Internal Modularity	899
64-4	Stack Driver Internal Initialization and Registration	902
65-1	Example of Hardware with Two IXP Processors	919
65-2	Example of Hardware with Single IXP Processor	920
65-3	General SoftSAR Software Overview	920
65-4	SoftSAR Software Components	921
65-5	SAR Control Architecture Overview	924
65-6	Place of Software Blocks in Single IXP Hardware Platform	925
65-7	Place of Software Blocks in Dual IXP Hardware Platform	926
65-8	SAR Control Decomposition	927
65-9	Create Operation Data Flow	928
65-10	Remove Operation Data Flow	929
65-11	Read Statistics Operation Data Flow	929
65-12	SoftSAR Main Core Components Dependencies	931
65-13	SAR Control MAIN Core Components Decomposition	931
65-14	Block Ordering Convention	932
65-15	SoftSAR Agent Core Components Dependencies	936
65-16	SAR Control Agent Core Component Decomposition	937
66-1	MPLS Forwarder Core Component Packet Data Flow	954
66-2	MPLS Core Component Logical Sub-modules and Interactions	958
A-1	NetSim Framework Architecture	964
A-2	Devices and Bus Connections—Media/Switch Fabric Connections	965
A-3	Assign Media Bus Port Input	966
B-4	AAL-2 Sub-layers	985
B-5	ATM-AAL-2 Data Indication	991
B-6	Local Memory Queues	992
B-7	Algorithm—AAL-2 Queue Cell Microblock	993
B-8	AAL-2 CPS-SSSAR Microblock Flowchart (page 1 of 3)	994
B-9	AAL-2 CPS-SSSAR Microblock Flowchart (page 2 of 3)	995
B-10	AAL-2 CPS-SSSAR Microblock Flowchart (page 3 of 3)	996
C-11	CPS and SSCS Interaction	999
C-12	Format of AAL-2 CPS-Packet	1000
C-13	Format of SSSAR-PDU	1001
C-14	Pool of Threads Synchronization With CAM/LM	1002
C-15	AAL-2 TX Transmit Request Sourcing and Thread Synchronization	1003
C-16	AAL-2 Sublayer Selection and Sourcing from Local Memory Thread Queue	1005
C-17	AAL-2 Transmit SSSAR Sublayer Functions	1006
C-18	AAL-2 CPS Transmit Sublayer (1 of 3)	1008
C-19	AAL-2 CPS Transmit Sublayer (2 of 3)	1009
C-20	AAL-2 CPS Transmit Sublayer (3 of 3)	1010
C-21	Interface Between Timer Thread and SSSAR/CPS Threads	1011
C-22	Timer Processing Thread Flow	1012

C-23	Timer CU Data Structures in SRAM and Local Memory	1015
------	---	------

Tables

2-1	Format of the Buffer Handle	57
2-2	Buffer Handle Format for Q-Array in Packet Mode	57
2-3	Packet Metadata Format	58
2-4	Build Switches that May be Applied to all Projects	65
3-1	Child Meta Data in Cell Mode	71
3-2	Parent Meta Data in Cell Mode when Packet is Replicated	72
3-3	Child Meta Data in Packet Mode	72
3-4	Parent Meta Data in Packet Mode when Packet is Replicated	73
3-5	Rx Status Valid Values	75
4-1	Receive Reassembly Context	83
4-2	Statistics Counter Offsets from Base for Port	84
4-3	Jump Table Entries	84
4-4	Compile Time Build Switches Relevant to the Packet Receive Microblock	85
4-5	Performance Analysis of the Block	91
4-6	Packet RX Block Performance Analysis: I/O Operations for min Packet	91
4-7	Packet RX Microblock Characterization Data	92
5-1	CSIX Receive Algorithm for a Single Microengine Design	97
5-2	CSIX RX Algorithm Running on Two Microengines	98
5-3	CSIX Receive Reassembly Context for one Microengine	99
5-4	CSIX Receive Reassembly Context for First Microengine	99
5-5	CSIX Receive Reassembly Context for Second Microengine	100
5-6	CSIX Receive Lookup Key for One Microengine Design	100
5-7	CSIX Receive Lookup Key for Two Microengine Design	100
5-8	CSIX Receive: Statistics Counter Offsets from Base for VOQ	100
5-9	Compile Time Options Used in the One Microengine CSIX Receive Microblock ...	101
5-10	Compile Time Options Used in the Two Microengine CSIX Receive Microblock....	101
5-11	CSIX RX—Budget and Cycle Count for One Microengine Design	101
5-12	CSIX RX—Budget and Cycle Count for Two Microengine Design	101
5-13	CSIX RX Performance Analysis—I/O Operations for Min Packet Worst Case .	102
5-14	CSIX Rx Microblock Characterization Data	102
6-1	An Example of Split Port Number Bits	108
6-2	AAL5 RX Microblock Pre-Processor Switches	109
6-3	RXC Data Structure	111
6-4	Valid Combinations	112
6-5	Port Based Counters	113
6-6	VC Based Counters	113
6-7	CRC32 From Two LW Used to Find a Hash Table	114
6-8	Buckets Structure in the Primary Hash Table	114
6-9	Buckets Structure in the Secondary Hash Table	114
6-10	VC_key#	115
6-11	Instruction Estimates—Two Microengine Design	130
6-12	Break up of the Critical Path Cycles for the Worst Case Scenario	130
6-13	Instruction Estimates and I/O Usage for ATM RX Microblock	131

6-14	SRAM Memory	132
6-15	Local Memory	132
6-16	Registers and Signals—Two Microengine Design	132
6-17	Registers and Signals—Single Microengine Design	133
6-18	I/O Commands—Two Microengine Design	133
6-19	I/O Commands—Single Microengine Design	134
6-20	ATM AAL5 RX Microblock Characterization Data	134
7-1	CSIX Transmit Context.....	142
7-2	CSIX Transmit Control Word.....	143
7-3	CSIX Traffic Manager Header Added for Each c-frame	143
7-4	CSIX Traffic Manager Header Added for First c-frame	143
7-5	Reference CSIX Base Header - 1 Shortword.....	144
7-6	Reference CSIX Unicast Extension Header.....	144
7-7	CSIX Transmit Statistics.....	144
7-8	Logical Phases of CSIX TX Block	145
7-9	Logical Phases of CSIX TX Block Interleaving for Multiple Requests.....	146
7-10	CSIX Transmit Performance Analysis: Worst Case Cycle Count.....	150
7-11	Instruction Cycle Counts in the Critical Path	150
7-12	CSIX Transmit Performance Analysis: I/O Operations.....	151
7-13	CSIX Tx Microblock Characterization Data	151
8-1	Possible Solutions Based on Three Packet Transmit Modes.....	155
8-2	Context-Relative Thread Execution Status Flag—exe_stat_flag	156
8-3	TX Request - One Longword.....	157
8-4	Queue Entry Structure.....	157
8-5	Queue Descriptor Structure.....	158
8-6	Output Transmit Control Word - Two Longwords.....	159
8-7	Compile Time Options Used in the CSIX Receive Microblock	159
8-8	Packet TX Performance Analysis - Worst Case Cycle Count	168
8-9	Packet TX Performance Analysis - Worst Case I/O Operations	168
8-10	Packet TX for SPHY and MPHY-4 Microblock Characterization Data	168
9-1	Packet TX for MPHY-16: Globals Stored in Absolute Registers	174
9-2	Context Relative Thread Execution Status Flag - exe_stat_flag	174
9-3	TX Request - One Longword.....	175
9-4	Queue Entry Structure.....	175
9-5	Queue Descriptor Structure.....	176
9-6	Output Transmit Control Word - Two Longwords.....	177
9-7	Packet TX Statistics	177
9-8	Performance Analysis - Instruction Estimate.....	184
9-9	Performance Analysis - I/O Operations	184
9-10	Packet TX for MPHY-16 Microblock Characterization Data	184
10-1	Three-word NN Ring between the first and the Second Microengine	190
10-2	Six-word NN Ring for Non-sop M-packet	190
10-3	Instruction Cycle Counts in the Critical Path for Two Microengine Design	192
10-4	Performance Requirements for OC-192 POS—Two Microengine Design	192
10-5	Packet TX for OC-192 POS Microblock Characterization Data	192
11-1	Format of 8-byte CPCS Trailer.....	197
11-2	Format of 5-byte Cell Header	198
11-3	Format of TXC1	202
11-4	Format of TXC2.....	203
11-5	Counters Available via the Counters Build Switch	204

11-6	AAL5 TX Microblock Switches	204
11-7	OC-48 Algorithm - Pipestage 1	205
11-8	OC-48 Algorithm - Pipestage 2	206
11-9	Quad OC-12 Algorithm Pipestage 1	206
11-10	Quad OC-12 Algorithm Pipestage 2	206
11-11	OC-48 Performance Analysis	211
11-12	OC-48 Critical Path Cycles per Stage	211
11-13	Quad OC-12 Performance Analysis	211
11-14	Quad OC-12 Critical Path Cycles per Stage	212
11-15	AAL5 TX Microblock Resource Usage	212
11-16	AAL5 TX Microblock Characterization Data	212
11-17	Number of Cycles in Worst Case for Tasks Performed in Critical Path	215
12-1	Globals Stored in Absolute Registers	219
12-2	Context Relative Thread Execution Status Flag	219
12-3	Transmit Request—One Long Word	220
12-4	Packet Queue Entry for Each Port	220
12-5	Structure of Port Status for a Port in the Local Memory	221
12-6	Output Transmit Control Word—2 Long Words	223
12-7	32-bit Statistics Counters in SRAM	223
12-8	Instruction Estimate for IXP24XX Packet Transmit Microblock	229
12-9	I/O Operations Performed in Different Phases	229
12-10	Instruction Estimate for IXP28XX Packet Transmit Microblock	230
12-11	I/O Operations Performed in Different Phases of the Microblock	230
12-12	Packet TX-Multiports Microblock Characterization Data	230
13-1	Compile Time Options Used in the Queue Manager	239
13-2	Ingress Queue Manager Performance Analysis: Worst Case Cycle Count	246
13-3	Ingress Queue Manager Performance Analysis: I/O Operations	246
14-1	Compile Time Options Used in the Queue Manager	249
14-2	Differences between the OC-48 and OC-192 Queue Manager	251
14-3	QM OC-192—Worse Cycle Count Estimate for Different Phases	251
14-4	IQM OC-192—I/O Operations Performed in the Different Phases	251
14-5	OC-192 Queue Manager Microblock Characterization Data	252
15-1	Egress Queue Manager Performance Analysis: Cycle Count	256
15-2	Packet Queue Manager Microblock Characterization Data	256
16-1	Scratch Ring (LCSNAP Encap and ATM QM)—Three Long Words	261
16-2	NN Ring (ATM QM and TM 4.1 Shaper)—Two Long Words	261
16-3	Scratch Ring (TM 4.1 Scheduler and ATM QM)—One Long Word	262
16-4	Scratch Ring (ATM QM and AAL-5 TX)—Two Long Words	262
16-5	ATM QM Compile Time Switches	263
16-6	Cycle Count	271
16-7	I/O Operations Performed in Different Phases of ATM QM	271
16-8	ATM Queue Manager Microblock Characterization Data	271
17-1	Ingress Scheduler Globals	279
17-2	Queue Group Fields	279
17-3	Queue Fields	279
17-4	Instruction Cycle Estimates for Scheduler Components	287
18-1	Two-word VoQ in Local Memory	290
18-2	Globals Stored in Absolute Registers Shared by all Threads	290
18-3	Format of the Active List Register	290
18-4	Instruction Cycle Count for the Scheduler	297

18-5	Fabric Scheduler For OC-192 Microblock Characterization Data	297
19-1	Egress Scheduler: Port-Specific Fields	304
19-2	Egress Scheduler: Queue-Specific Fields.....	305
19-3	Egress Scheduler: Registers for Data Structures.....	305
19-4	Egress Scheduler Performance Analysis: Cycle Counts.....	315
20-1	virtual Queue with 4 longwords Data Structure	327
20-2	Data Structure Stored on Each Port.....	327
20-3	Data Structures Stored in Registers.....	328
20-4	OC-192 DRR—Worse Cycle Count Estimate	329
20-5	OC-192 DRR Microblock Characterization Data	329
21-1	Cycle Count Table (including unfilled defers)	335
21-2	I/O Latency Analysis Table.....	335
21-3	Memory Access Summary Table	335
21-4	SRAM Footprint.....	336
21-5	Scratchpad Footprint	336
21-6	Code Store Footprint	336
22-1	Transition Message Format (NN ring between QM and Scheduler)	341
22-2	Flow Control Message Format (Scheduler xfer registers).....	341
22-3	Dequeue Message Format (scratch ring between Scheduler an QM)	342
22-4	Port Record	343
22-5	Queue Record	344
22-6	Global Register.....	344
22-7	Cycle Count Table (including unfilled defers)	351
22-8	I/O Latency Analysis Table.....	351
22-9	Memory Accesses Summary Table.....	351
22-10	Code Store Footprint	351
23-1	GRCA Write-out Block for Each Cell for Low-bit Rate VCs.....	363
23-2	GRCA Write-out Block for Plain UBR w/priority VCs.....	363
23-3	GRCA Write-out Block or Each Cell for High-bit Rate VCs	363
23-4	Definition of Variables Stored in Local Memory	371
23-5	UBR w/priority Table Entry	372
23-6	Instruction Counts for the Blocks.....	384
23-7	Local Memory Map.....	385
23-8	Performance-oriented Port State Data Structure	390
23-9	Updated Local Memory Map	391
23-10	Worst and Average Cycle Count Estimate for the Shaper Microblock	393
23-11	Worst and Average Cycle Count Estimate for the Scheduler Microblock	393
23-12	I/O Operations Performed in the worst Case by the Shaper and Scheduler.....	394
24-1	Listing of Build Switches.....	400
24-2	Table of Next Microengine Values	401
24-3	RFC1812 <i>MUST</i> Check Items.....	402
24-4	RFC1812 <i>SHOULD</i> Check Items.....	402
24-5	Optional RFC2644 Check Items.....	403
24-6	Control Block Layout	404
24-7	Next Hop Information Associated with Each Route—Uncompressed Format ..	405
24-8	Next Hop Information Associated with Each Route—Compressed Format	406
24-9	Packet Action Flags.....	406
24-10	IPv4 Counters and SRAM Offsets.....	408
24-11	Trie Entry Layout.....	409
24-12	List of IPv4 Microblock Symbols.....	414

24-13	IPv4 Microblock Exceptions	414
24-14	IPv4 Forwarder Cycle Counts	418
24-15	IPv4 Forwarder I/O Latency Analysis for Minimum Packet	418
25-1	Compile Time Build Switches.....	421
25-2	Table of Next ME Numbers	422
25-3	RFC2460 “Must” Checks Performed in the Microblock.....	423
25-4	RFC2373 <i>Should</i> Checks Performed in the Microblock	423
25-5	Fields of a <i>Real</i> Next-Hop Entry.....	424
25-6	Generic Flags.....	425
25-7	Next-Hop ID Types.....	425
25-8	Fields of an Intermediate Next-Hop Entry	426
25-9	Counter Offsets in Incoming Statistics Block	428
25-10	Counter Offsets in Outgoing Statistics Block	429
25-11	Longest Prefix Match Algorithm Performance.....	437
25-12	IPv6 Symbols.....	438
25-13	IPv6 Exception Codes.....	439
25-14	IPv6 Cycle Count Analysis	443
25-15	IPv6 I/O Latency Analysis for Min Packet	443
25-16	IPv6 Forwarder Microblock Characterization Data.....	444
26-1	IPv6 to IPv4 Tunneling Build Switches.....	449
26-2	Next ME Values	450
26-3	RFC 3056 <i>Must</i> Checks Performed in the Microblock	451
26-4	RFC 2893 <i>Should</i> Items for IPv4 Source Address	451
26-5	RFC 2893 <i>Should</i> Items for IPv6 Source Address	451
26-6	RFC 2893 <i>Must</i> Items for IPv4 Destination Address.....	452
26-7	RFC 3056 <i>Must</i> Items for IPv6 Source and Destination Address	452
26-8	Tunnel Next-Hop Entry for V6V4-Tunnel-Decap Microblock.....	455
26-9	Format of Ingress Source List Block	456
26-10	Next Hop Fields.....	465
26-11	Tunnel Next-Hop Flags for V6V4-Tunnel-Encap Microblock	466
26-12	IPv4 Header Creation for Encapsulated Packets.....	469
26-13	Symbols	470
26-14	Exception Codes	471
26-15	Cycle Count Analysis	471
26-16	I/O Latency Analysis for Min Packet	471
27-1	Buffer Parameters Defined.....	476
27-2	Compile Time Build Switches.....	477
27-3	Counters and Offsets from the Base Address.....	478
27-4	NAT Table Data Format	480
27-5	V6-V4 NAT Hash Index.....	482
27-6	Data Format	482
27-7	IPv4 Address Used for NAPT-PT	483
27-8	V6-V4 NAPT Hash Index	485
27-9	Symbols Required for NAT-PT Microblock and the Core component to Work ..	493
27-10	Exception Codes Generated by the NAT-PT Microblock	493
27-11	Translation Microblock Cycle Count Analysis	494
27-12	I/O Latency Analysis (IPv4-TCP)	494
27-13	I/O Latency Analysis (IPv6-TCP)	494
27-14	IPv6 to IPv4 Tunnel Decap Microblock Characterization Data	495
27-15	IPv6 to IPv4 Tunnel Encap Microblock Characterization Data.....	497

28-1	Layer-2 Decapsulation Worst Case Instruction Count	506
28-2	Ethernet Decap Microblock Characterization Data	507
28-3	PPP Decap Microblock Characterization Data	509
29-1	Format of an Entry in the Layer-2 Table for LLC SNAP	514
29-2	Layer-2 Encapsulation Worst Case Cycle Count	514
29-3	Layer-2 Encapsulation I/O Operations	514
29-4	Ethernet Encap Microblock Characterization Data	515
29-5	PPP Encap Worst Case Cycle Count	517
29-6	PPP Encap I/O Operations	517
29-7	PPP Encap Microblock Characterization Data	518
29-8	LLC SNAP Encap Microblock Characterization Data	520
30-1	Build Switches for IPv4 6-tuple Classifier Microblock	527
30-2	Input Variables Consumed	528
30-3	Output Variables Modified	528
30-4	Variables Imported by this Block	529
30-5	Hash Entry Definition	531
30-6	Cycle Count Table (including unfilled defers)	537
30-7	I/O Latency Analysis Table	537
30-8	Memory Accesses Summary Table	538
30-9	DRAM/SRAM Footprint	538
30-10	Code Store Footprint	538
31-1	Build Switches for TCM Microblock	542
31-2	Input Variables Consumed by SRTCM	543
31-3	Output Variables Modified by SRTCM	543
31-4	Variables Imported by this Block	543
31-5	SRTCM Table Entry Definition	546
31-6	Cycle Count Table for SRTCM Version (including unfilled defers)	557
31-7	Cycle Count Table for TRTCM Version (including unfilled defers)	557
31-8	Cycle Count table for SRTCM and TRTCM version (including unfilled defers)	558
31-9	I/O Latency Analysis Table (for all versions)	558
31-10	Memory Access Summary Table	558
31-11	SRAM Footprint on Ingress IXP 2400	558
31-12	SRAM Footprint on Egress IXP 2400	559
31-13	Code Store Footprint	559
32-1	Variables Consumed by DSCP Marker	561
32-2	Variables Modified by DSCP marker	561
32-3	Cycle count table (including unfilled defers)	563
32-4	Code Store footprint	564
33-1	Build Switches for DSCP Classifier Microblock	566
33-2	Input Variables Consumed	567
33-3	Output Variables Modified	567
33-4	Variables Imported by this Block	568
33-5	Classification Result Entry Definition	569
33-6	Cycle Count Table (including unfilled defers)	574
33-7	I/O Latency Analysis Table	574
33-8	SRAM Footprint	575
33-9	Code Store Footprint	575
34-1	Build Switches for WRED Microblock	580
34-2	Input Variables Consumed by WRED	581
34-3	Output Variables Modified by WRED	581

34-4	Variables Imported by this Block.....	582
34-5	WRED Table Entry Definition.....	586
34-6	Cycle Count Table (including unfilled defers).....	596
34-7	I/O Latency Analysis Table—WRED Low-speed Queues.....	596
34-8	Memory Access Summary Table	596
34-9	SRAM Footprint.....	597
34-10	Code Store Footprint.....	597
34-11	Cycle Count Table (including unfilled defers).....	597
34-12	I/O Latency Analysis Table - WRED for High-Speed Queues	598
34-13	Memory Access Summary Table	598
34-14	SRAM Footprint.....	598
34-15	Code Store Footprint.....	598
35-1	Input Variables Consumed by FTN Forwarder.....	603
35-2	Output Variables Modified by FTN Forwarder.....	604
35-3	NHLFE Entry	606
35-4	NHLFE Set Entry.....	607
35-5	NHLFE Counters Table Entry	608
35-6	Cycle Count Table (including unfilled defers) - UNIFORM Tunnel Mode.....	614
35-7	Cycle Count Table (including unfilled defers) - PIPE Tunnel Mode	614
35-8	I/O Latency Analysis Table	614
35-9	Data Structures Footprint.....	615
35-10	Code Store Footprint.....	615
35-11	FTN Forwarder Microblock Characterization Data.....	615
36-1	Input Variables Consumed by ILM Forwarder.....	621
36-2	Output Variables Modified by ILM Forwarder.....	622
36-3	ILM Table Entry.....	624
36-4	ILM_NHLFE Set Entry.....	625
36-5	InSegment Counters Table Entry.....	626
36-6	Input Port Counters Table	627
36-7	Cycle Count Table (including unfilled defers) for Operation SWAP	634
36-8	Cycle Count Table (including unfilled defers) for Operation POP	634
36-9	Cycle Count Table (including unfilled defers) for Operation POP_FORWARD	634
36-10	Cycle Count Table (including unfilled defers) for Operation SWAP_PUSH	635
36-11	I/O Latency Analysis Table	635
36-12	Data Structures Footprint.....	635
36-13	Code Store Footprint.....	635
36-14	ILM Forwarder Microblock Characterization Data.....	636
37-1	Packet Copier Microblock Build Switches	641
37-2	Packet Copier Microblock Symbols.....	642
37-3	Data Flow through Packet Copier	643
37-4	Packet Copier Request Message.....	644
37-5	Packet Copier Response Message.....	644
37-6	Packet Copy Context Queue Entry	644
37-7	Packet Replication Algorithm Running on One Microengine	645
37-8	Packet Replication Algorithm Running on Two Microengines.....	646
37-9	Cycle Counts for Packet Replication on Two Microengines.....	646
37-10	Packet Copier Microblock Characterization Data.....	647
38-1	Freelist Manager Microblock Algorithm.....	652
38-2	Budget and Cycle Count	652
38-3	I/O Operations for Minimum Packet Worst Case	653



38-4	Freelist Manager Microblock Characterization Data	653
39-1	Dynamic Properties and Clients	662
39-2	Properties Data Structure	662
39-3	Property API	663
39-4	Core Component Functionality Mapped to Different Framework Layers	666
40-1	Re-initialization Procedure Options	686
40-2	Loading Microcode and Starting Engines	686
40-3	Loading Microcode and Starting Engines	687
40-4	User Initialization and Shutdown Hooks	687
40-5	Core Component Configuration Parameters	689
41-1	POS RX Core Component Static Configuration Data	696
41-2	POS RX Core Component Patch Symbols	697
41-3	POS RX Core Component Infrastructure API	697
41-4	POS RX Core Component Messaging API	697
41-5	POS RX Core Component Library API	698
42-1	CSIX RX Core Component	701
42-2	CSIX RX Core Component Infrastructure API	701
42-3	CSIX RX Core Component Messaging API	701
42-4	CSIX RX Core Component Library API	702
43-1	Ethernet RX Core Component Dynamic Configuration Data	706
43-2	Ethernet RX Core Component Static Configuration Data	706
43-3	Symbols and Memory Base Addresses to Patch Packet RX Microblock	707
43-4	Symbols and Memory Base Addresses L2 Decap Microblock	707
43-5	Ethernet RX Core Component Functions	708
43-6	Ethernet Interface RX Messaging Functions	708
43-7	Ethernet Interface RX Library Functions	709
44-1	Patch Symbols for CSIX TX Core Component	715
44-2	CSIX TX Core Component Infrastructure API	715
44-3	CSIX TX Core Component Messaging API	715
44-4	CSIX TX Core Component Library API	716
45-1	ATM/POS TX Core Component Dynamic Configuration Data	719
45-2	ATM/POS Interface TX Configuration Parameters	720
45-3	ATM/POS Media Card Operation Mode	721
45-4	Supported Line Side Interface and Media Interface for each OC Mode	722
45-5	Symbols and Memory Base Addresses Patched into ATM TX Microblock	722
45-6	Symbols and Memory Base Addresses Patched into POS TX Microblock	723
45-7	ATM/POS TX Core Component Infrastructure API	724
45-8	ATM/POS TX Core Component Messaging API	724
45-9	ATM/POS TX Core Component Library API	724
46-1	Ethernet TX Core Component Input Sources	727
46-2	Dynamic Configuration Data	728
46-3	Ethernet TX Core Component Static Configuration Data	729
46-4	Symbols and Memory Base Addresses to be Patched	730
46-5	Ethernet TX Core Component Data Structures	732
46-6	Ethernet TX Core Component Infrastructure API	732
46-7	Ethernet TX Messaging Functions	733
46-8	Ethernet TX Library API	733
47-1	Ethernet ARP Library API	741
48-1	Queue Manager Core Component Configuration Items	746
48-2	Queue Manager Core Component Infrastructure API	748

48-3	Queue Manager Core Component Messaging API.....	748
48-4	Queue Manager Core Component Library API	748
50-1	Scheduler Core Component Configuration Items	755
50-2	Scheduler Core Component Infrastructure API.....	757
51-1	New Patching Symbols in Scheduler (DiffServ) Core Component.....	759
52-1	IPv4 Forwarder Core Components Statistics.....	765
52-2	RFC 1812 Checks for IP Addresses	770
52-3	IPv4 Forwarder Core Component Data Structures, Types, and Macros.....	773
52-4	IPv4 Forwarder Core Component Infrastructure API	774
52-5	IPv4 Forwarder Message Helper API.....	774
52-6	IPv4 Forwarder Library API.....	775
53-1	Counter Offsets in Incoming Statistics Block	779
53-2	Counter Offsets in Outgoing Statistics Block	780
53-1	IPv6 Forwarder Data Structures, Types and Macros	787
53-2	IPv6 Forwarder Core Component Infrastructure API.....	787
53-3	IPv6 Forwarder Message Helper API.....	788
53-4	IPv6 Forwarder Library API	789
54-1	IPv6-IPv4 Tunneling Data Structures, Types and Macros.....	798
54-2	IPv6 to IPv4 Tunneling Core Component Infrastructure AP	799
54-3	IPv6 to IPv4 Tunneling Core Component Message Helper API.....	799
54-4	IPv6 to IPv4 Tunneling Library API	801
55-1	Data Structures, Types and Macros in Translation Core Components.....	809
55-2	Translation Core Component Infrastructure API	809
55-3	Message Helper API in the Translation Core Component	810
55-4	Library API in the Translation Core Component	811
56-1	Static Configuration Items in 6-tuple Classifier	820
56-2	Data Structures for Configuring Exact-match Rules	823
56-3	6-Tuple Classifier Core Component Infrastructure API.....	824
56-4	6-Tuple Classifier Core Component Message Helper API.....	824
56-5	6-Tuple Classifier Core Component Library API	824
56-6	6-Tuple Classifier Core Component Modules	825
57-1	Static Configuration Items in TCM	828
57-2	Variables Imported by this Block.....	829
57-3	TCM Core Component Data Structures	832
57-4	TCM Core Component Infrastructure API	832
57-5	TCM Core Component Message Helper API	832
57-6	TCM Core Component Library API	833
57-7	TCM Core Component Modules	833
58-1	Static Configuration Items in WRED core component	836
58-2	Variables Imported by this Block.....	837
58-1	WRED Core Component Data Structures for Message Helper and Library APIs... 840	
58-2	WRED Core Component Data Structures for Message Helper and Library APIs... 840	
58-3	WRED Message Helper API	840
58-4	WRED Library API	841
58-3	WRED Core Component Modules	841
59-1	Mapping between Output Identifiers	844
59-2	Static Configuration Items in 6-tuple Classifier	846
59-3	SRTCM Core Component Data Structures	848

59-4	SRTCM Core Component Infrastructure API	849
59-5	SRTCM Core Component Message Helper API	849
59-6	SRTCM Core Component Library API	849
59-7	SRTCM Core Component Modules.....	850
60-1	Route Table Manager Data Structures and Types.....	862
60-2	Route Table Manager Macros.....	862
60-3	Route Table Manager API.....	862
61-1	RTMv6 Data Structures, Types and Macros	869
61-2	RTMv6 Core COmponent Infrastructure API.....	870
62-1	L2 Table Entry	872
62-2	Data Structures, Types, Definitions and Enumerations in L2TM.....	876
62-3	L2 Table Manager Library API	876
63-1	Components of the Message Helper System.....	880
63-2	Types of Calls Supported by the Message Helper	881
63-3	ix_msup_call_table_s Structure Members	887
63-4	ix_msup_call_s Structure Members - Asynchronous and Synchronous	887
63-5	ix_msup_call_s Structure Members - Asynchronous only.....	887
63-6	ix_msup_call_s Structure Members - Synchronous only	887
63-7	Message Support Library API	892
64-1	Stack Driver Core Component Data Structures and Types.....	900
64-2	Stack Driver Core Component Infrastructure API	901
64-3	Stack Driver Core Component Infrastructure Separation API.....	901
64-4	Stack Driver Packet and Message Processing API.....	903
64-5	Stack Driver Properties API.....	903
64-6	Stack Driver Packet Classifier Data Structures	904
64-7	Stack Driver Packet Classifier External API	905
64-8	Stack Driver Outgoing Packet Classifier Data Structures	906
64-9	Stack Driver Outgoing Packet Classifier Internal API.....	907
64-10	Stack Driver VIDD System Data Structures	909
64-11	Stack Driver VIDD Local Data Structures.....	909
64-12	Required Functions for the Stack Driver MUX Interface	909
64-13	VIDD System API.....	910
64-14	VIDD MUX API.....	911
64-15	VIDD System Data Structures for Linux	913
64-16	VIDD System API for Linux	913
64-17	VIDD Linux Driver Support API	914
64-18	Transport Module Data Structures	914
64-19	Transport Module External API	915
65-1	General Rules for Calling Sequences	933
65-2	SAR Control MAIN Configuration Items	933
65-3	SAR Data Structures and Data Type Definitions.....	933
65-4	SAR Core Component Infrastructure API.....	934
65-5	SAR Messaging API.....	934
65-6	SAR Library API	935
65-7	SAR Control Plug-in API	935
65-8	Configuration Parameters	937
65-9	ATM RX Core Component Static Configuration Properties.....	940
65-10	ATM RX Core Component Patch Symbols	940
65-11	IXP2400 16 ports—Key for Hash Table	941
65-12	2048 ports—Key for Hash Table	941

65-13	Core Component Infrastructure API	942
65-14	ATM/POS TX Core Component Dynamic Configuration Data	944
65-15	ATM Interface TX Configuration Parameters	944
65-16	ATM TX Core Component Infrastructure API.....	944
65-17	List of Shared Tables	945
65-18	TM4.1 Core Component supported Core Component Infrastructure API	949
66-1	Static Configuration Data	956
66-2	Patching Symbols for ILM Forwarder Microblock.....	956
66-3	Patching Symbols for FTN Forwarder Microblock.....	957
66-4	Data Types and Structures	960
66-5	MPLS Core Component Infrastructure API	960
66-6	MPLS Message Helper API	961
66-7	MPLS Library API.....	961
A-1	DTD Qualifiers and Syntax.....	970
A-2	DTD: Value Description.....	971
B-3	Configuration Switches	987
B-4	CPS RXC Data Structure	988
B-5	CID RXC Data Structure	988
B-6	Primary Hash Lookup Table.....	989
B-7	Secondary Hash Lookup Table	989
B-8	AAL-2 Receive Microblocks CRC-5 SRAM Tables	990
B-9	AAL-2 RX Counters.....	990
B-10	Cycle Counts and I/O Latencies.....	997
C-11	AAL2 TX Supported Pre-Processor Switches	1013
C-12	AAL-2 Transmit VC Context Entry	1014
C-13	AAL-2 TX Microblocks CRC-5 SRAM tables.....	1015
C-14	AAL2 Transmit Counters.....	1016
C-15	Cycle Counts and I/O Latencies.....	1016

Revision History

Date	Revision	Description
July 2003	1.4	SDK 3.1 Beta updates
November 2003	IXA SDK 3.5	<p>Added new chapter on System Data Structures for Packet Replication.</p> <p>Added the following new microblocks: Packet Copier and Freelist Manager.</p> <p>Added new chapter for MPLS Core Component.</p> <p>Moved AAL2 Tx and Rx microblock information into Appendix B.</p> <p>Updated Chapter 64, "Stack Driver" with Linux* support.</p> <p>Updated Chapter 5, "CSIX RX Microblock" with new information for two microengine design.</p> <p>Added Microblock Characterization Data table to most microblock chapters.</p>
March 2004	IXA SDK 3.51	Added Microblock Characterization Data table to IPv4 Forwarder microblock

1.1 About this Manual

This Developer's Manual details the design for various building blocks implemented for the Intel® IXP2400 Network Processor and the Intel® IXP2800 Network Processor.

The Intel® Internet Exchange Architecture Software Development Kit (Intel® IXA SDK) software infrastructure provides a consistent, unified framework to develop the slow and fast path components of data plane processing.

There are two types of building blocks:

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

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

- Saves time by providing robust infrastructure software and APIs,
- Saves time by providing re-configurable building blocks,
- Permits portability across IXA network processors,
- Provides an ideal structure for 3rd party plug-in application modules.

For information about the IXA Portability Framework, refer to the *Intel® Internet Exchange Architecture Portability Framework Developer's Manual*.

1.1.1 Microblocks

The Intel® Internet Exchange Architecture Software Development Kit (Intel® IXA SDK) specifies how to develop modular software building blocks (microblocks) in microC or microcode on the microengines. These blocks may be combined with other software blocks provided by Intel (and possibly third parties) to build different applications such as line-cards for MPLS, DSLAMs, Multi-Service Switches and so on. Associated with each microblock is a slow path component on the Intel XScale® core processor (core component) that configures the microblock and handles exception packets. The IXA Portability Framework also specifies how to write the Intel XScale® core component and how to interface it with the corresponding microblock.

This document focuses on software building blocks for

- Sending and Receiving packets over POS, Ethernet and ATM media interfaces
- IPv4, IPv6 Forwarding, MPLS and DiffServ
- Sending and Receiving packets over the CSIX switch fabric
- Scheduling and queuing packets, Traffic Management and Shaping

It also illustrates how these blocks may be combined to create simple applications, such as POS IPv4, Ethernet and ATM.

1.1.2 Core Components

The core components are executed on the Intel XScale[®] core processor and are counterparts of the microblocks. Core components configure microblocks and handle packets that the microengines don't have cycles to process. They also provide a programming interface to higher-level applications such as the Control Plane PDK.

Core components conform to the rules and APIs of the Core Component Infrastructure. The data flow of core components is designed to reflect the packet pipeline built by the microblocks on the microengines. In addition to core components, there are also support libraries, such as the Route Table Manager and Message Helper Support Library, which are used by the core components.

The System Application is responsible for initializing all modules of the system.

1.2 Organization of the Manual

This manual consists of two distinct sections as follows:

- Microblocks
 - Microblocks (Packet RX, IPv4, etc.)
- Core Components
 - Application Overview
 - Core Components (POS RX, IPv4, Stack Driver, and so on)
 - Support Libraries (Route Table Manager, Message Helper)
 - System Application.

1.3 Other Sources of Information

This manual is part of the IXA Portability Framework documentation set, which also includes the following documents:

- *Intel[®] Internet Exchange Architecture Software Building Blocks Reference Manual*
- *Intel[®] Internet Exchange Architecture (IXA) Software Building Blocks Applications Design Guide*
- *Intel[®] Internet Exchange Architecture Portability Framework Reference Manual*
- *Intel[®] Internet Exchange Architecture Portability Framework Developer's Manual*
- *Intel[®] Internet Exchange Architecture Software Development Kit Software Framework Getting Started Guide*

In addition, the following documents are also useful as references:

Acronym	Title	Date
[RFC2684]	Multi-protocol Encapsulation over ATM Adaptation Layer 5	
[RFC791]	Internet Protocol Specification	
[RFC1812]	Requirements for IP Version 4 Routers	
[RFC2644]	Changing the Default for Directed Broadcasts in Routers	
[TM4.1]	Traffic Management Specification, Version 4.1, March 1999. ATM Forum document af-tm-0121.000.	March 1999
[TM4.1_UBR]	Addendum to TM 4.1: Differentiated UBR, July 2000. ATM Forum document af-tm-0149.000.	July 2000



Applications

The Intel® Internet Exchange Architecture Software Development Kit (Intel® IXA SDK) includes example code that demonstrates network processor features and data flow and can be used to jump-start customer application development. Software building blocks include supported, tested, documented, and RFC-compliant (where applicable) software components that can be plugged into customer-developed products and optionally extended to fit unique customer needs. Applications that demonstrate how to use production building blocks are available for Ethernet, Packet over SONET, and ATM on all Intel® IXP2XXX network processors.

Refer to [*Intel® Internet Exchange Architecture Software Building Blocks Applications Manual*](#) for a description.

System Data Structures and Design Choices

2

This section describes some system-wide data structures that are used by all applications in this document. It also describes certain design choices that are relevant across application.

2.1 Buffer Handle

Every buffer consists of a buffer data area in DRAM and a buffer descriptor in SRAM (there is a 1-1 mapping between DRAM buffers and SRAM buffer descriptors). A buffer handle of 32 bits identifies every buffer uniquely. [Table 2-1](#) shows the format of this buffer handle.

Table 2-1. Format of the Buffer Handle

LW	Bits	Size	Field	Description
0	31:31	1	EOP flag	This buffer includes EOP (end of packet)
	30:30	1	SOP flag	This buffer includes SOP start of packet)
	29:24	6	Cell Count	Cell Count for buffer (used in cell mode only)
	23:0	24	Address	Address in SRAM of buffer descriptor in long words

The buffer handle MUST NOT be 0, since it may be queued into a scratch ring in a ME-ME message—for example, QM enqueue message.

When the Q-Array is set up in packet mode (see [Section 15.2, “Q-Array Hardware Configuration” on page 255](#)), the cell count and SOP flag in the buffer handle are ignored. They may be replaced by an approximation of the packet length if DRR scheduling is being used. In this case, the packet length is stored in the 7 bits of the handle. For example, for POS frames with an MTU of 9k, the value stored in the 7 bit field is multiplied by 128 to get the approximate packet length.

[Table 2-2](#) lists the format of the buffer handle for the packet mode.

Table 2-2. Buffer Handle Format for Q-Array in Packet Mode

LW	Bits	Size	Field	Description
0	31:31	1	EOP flag	This buffer includes EOP
	30:24	7	Packet Length	Packet length in units of 128 (pre-defined chunks of bytes)
	23:0	24	Address	Address in SRAM of buffer descriptor in long words

2.2 Packet Meta Data (Buffer Descriptor)

The first 8 bytes (2 long words) of the packet metadata (buffer descriptor) are always the same for every application. The other fields (a word at a time) may be re-ordered and some of the words may be omitted and new words added depending on the needs of the individual application. To facilitate interoperability of microblocks, all fields in the metadata must be accessed via dispatch loop macros (see the *Intel® Internet Exchange Architecture Portability Framework Developer's Manual*). If the layout of the metadata is changed, these macros need to be updated. If MicroC is used (or on the Intel XScale® core), a recompile of code with the new structure ensures correctness as long as there are no dependencies in the code on the relative layout of the metadata words.

Table 2-3 shows the layout of the packet metadata for this application. This structure is 32 bytes or 8 long words.

Table 2-3. Packet Metadata Format

LW	Bits	Size	Field	Description
0	31:0	32	buffer_next	Buffer handle of next buffer in the chain
1	31:16	16	buffer_size	Buffer size in bytes
	15:0	16	offset	Offset of the start of data in the buffer in bytes
2	31:28	16	packet_size	Total packet size across buffers
	15:12	4	free_list_id	Free list ID for buffer
	11:8	4	rx_stat	Receive Status Flag
	7:0	8	header_type	Type of header at offset bytes into the packet
3	31:16	16	input_port	Input port on the ingress processor
	15:0	16	output_port	Output port on the egress processor
4	31:16	16	next_hop_id	Next hop IP ID
	15:8	8	fabric_port	Output port for fabric, indicating destination blade
	7:4	4	reserved	reserved
	3:0	4	nexthop_id_type	ID specifying in which table to look up the next_hop_id
5	31:24	8	color	QoS color
	23:0	24	flow_id	Flow ID (QoS flow ID or MPLS label/flow ID)
6	31:16	16	class_id	Class ID
	15:0	16	reserved	reserved
7	31:0	32	packet_next	Pointer to next packet (unused in cell mode)

The metadata for the first buffer in a packet chain contains all these fields. For the remaining buffers, only the first two long words are relevant. The rest are not used.

For this application, the data portion of each buffer is located in DRAM and is 2048 bytes in size. This size is configurable as long as it is set to a power of two. Only one buffer free-list is used and accordingly the Free list ID is always 0. For the first buffer in a packet chain, the offset is initially set to 128 or more bytes (see [Section 2.3](#)) to leave headroom for additional headers to be prepended to the packet.

2.3 Optimizing DRAM Bank Scheduling for Buffers

This optimization is only relevant to the IXP2400 and is not required on the IXP2800.

The IXP2400 has one DRAM channel with four banks. The DRAM memory banks are interleaved to improve concurrency and bandwidth utilization. Bits 7 and 8 of the command address are used as the bank select bits. This means that every 128 bytes of DRAM falls into a different bank.

In order to spread the memory accesses uniformly across banks, the following scheme is used. For the SOP mpacket for every frame, the starting offset into the DRAM buffer at which the packet data is written is rotated among four values—128, 256, 384 and 512. This implies that for a stream of minimum size packets, the first packet is written into a buffer at offset 128, the second packet into another buffer at offset 256, and so on.

This optimization is only relevant to the IXP2400 and is not required on the IXP2800. The IXP2800 has 3 RDRAM channels with 4 banks each. An RDRAM channel is selected based on MOD-3 of bits 31:7 of the DRAM address. The bank within a channel is selected based on a function that uses 2 user-specified bits (in the RDRAM_CONTROL CSR) of the DRAM address. Since the buffer size is 2048 bytes, based on the MOD-3 arithmetic a sequence of contiguous buffers automatically interleave across different channels. Bank interleaving within a channel is also handled by the default setting of the RDRAM_CONTROL CSR.

2.4 Buffer Chaining

Since the maximum size of a POS packet is 9k and the size of each buffer is 2k, some packets may be stored in a chain of buffers. There are two types of buffer chaining possible:

- Flat queuing
One single link list is used for both packets and buffers within a packet. This mode is appropriate when the Q-Array hardware is used in the cell-dequeue mode of operation.
- Hierarchical queuing
In this mode, there is a separate link list for buffers within a packet and a separate link list for packets. The Q-Array hardware is used to maintain the link list of packets. The link list of buffers within a packet is maintained in software. This mode is appropriate when the Q-Array hardware is used in the packet dequeue mode of operation.

2.4.1 Flat Queueing (Cell Based Dequeue)

Flat queueing is used on the Ingress IXP2400, since the dequeue and transmit of c-frames is done one cell at a time where a cell is a c-frame. The Q-Array hardware is used to link buffer descriptors together using the first 4 bytes of the buffer descriptor. As explained earlier, one single link list is maintained for packets and buffers within a packet. SOP/EOP bits are used to mark the start and end of a packet.

Figure 2-1 shows a single packet spanning two buffers A and B enqueued on a queue in the Q-Array. In the figure below, the packet is the only packet on the queue.

Figure 2-1. Flat Queueing on Ingress IXP2400 (single packet)

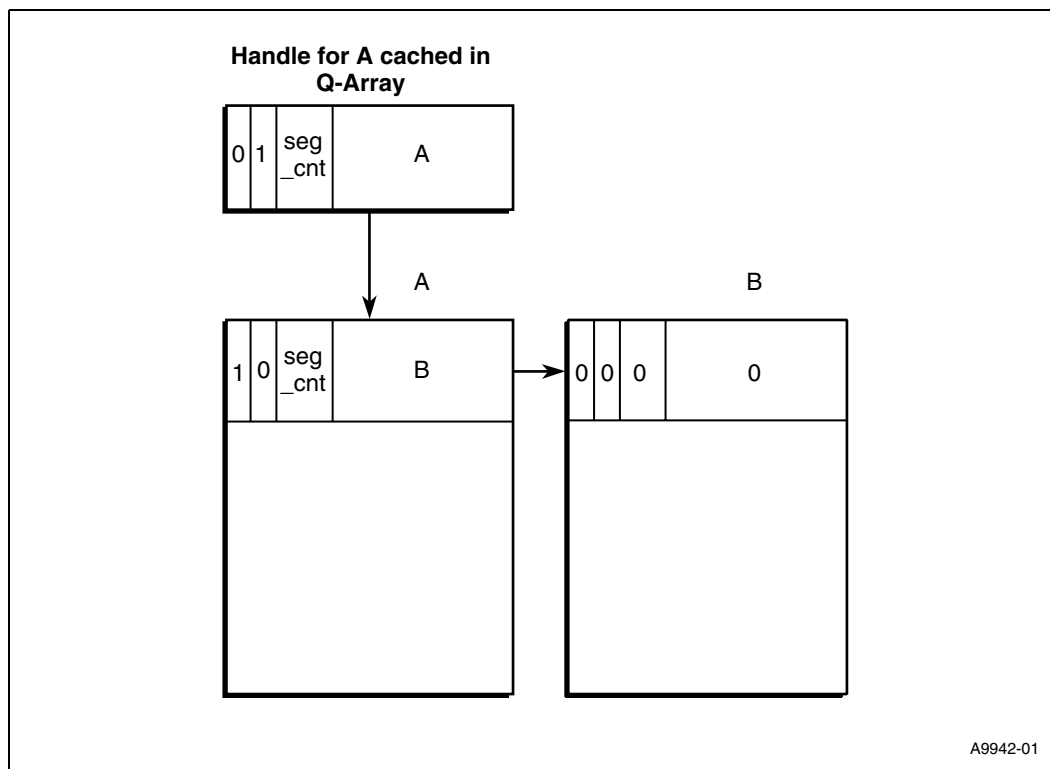


Figure 2-2. Flat Queueing on Ingress IXP2400 (multiple packets)

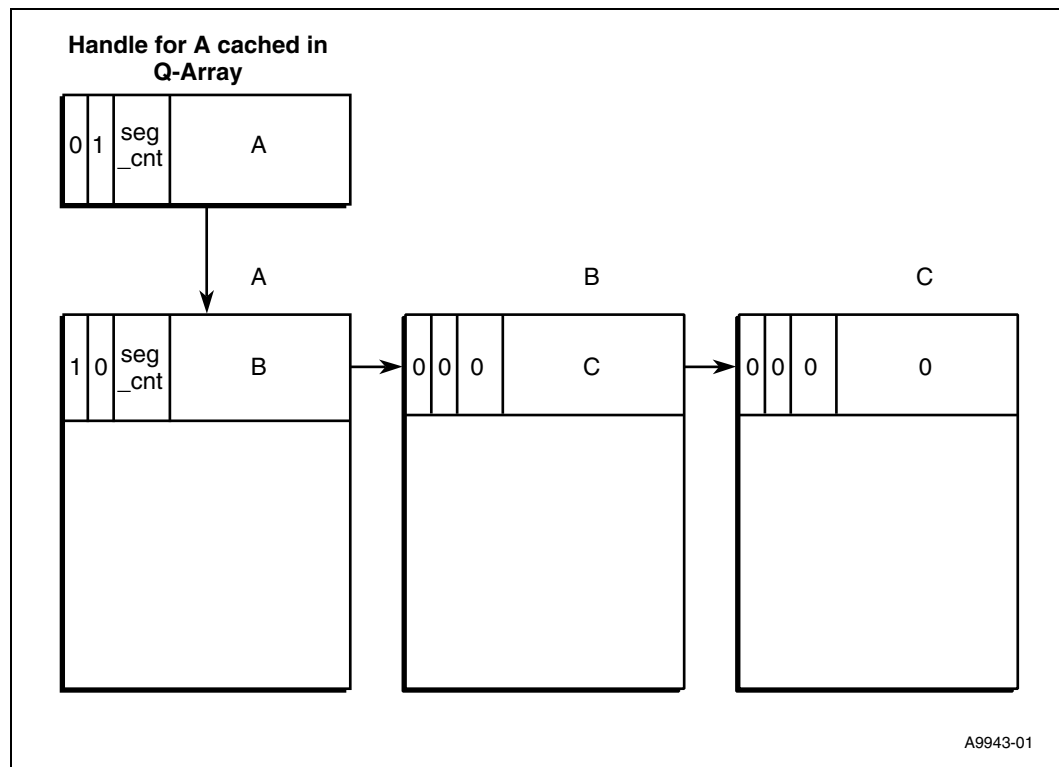


Figure 2-2 illustrates flat queueing with three buffer descriptors A, B and C, which hold two packets. The buffer descriptor A is the head of the link list. The handle for A is cached in the Q-Array. Based on the SOP/EOP bits in the handle for A, we can deduce that A is the buffer descriptor for a buffer that contains the start of the first packet. The first 4 bytes of the buffer descriptor A, point to the handle for buffer descriptor B. Based on this handle, we can deduce that B is the buffer descriptor for a buffer that contains the second part of the first packet. As we traverse down the chain, we can figure out that the buffer C contains a complete packet, which is the second packet in the link list. The link list is terminated at C.

2.4.2 Hierarchical Queueing (Packet Based Dequeue)

Hierarchical queueing is used on the Egress IXP2400, since the dequeue and transmit of POS frames is done one packet at a time. As explained earlier, two link lists are maintained – one that uses the Q-Array hardware and queues up packets (using the `packet_next` field in the buffer descriptor), and the other that uses the 4 byte `buffer_next` pointer in the buffer descriptor to queue up multiple buffers within a packet.

Figure 2-3 shows how a single packet containing 2 buffers would be chained and queued using hierarchical queuing.

Figure 2-3. Hierarchical Queuing on Egress IXP2400 (single packet)

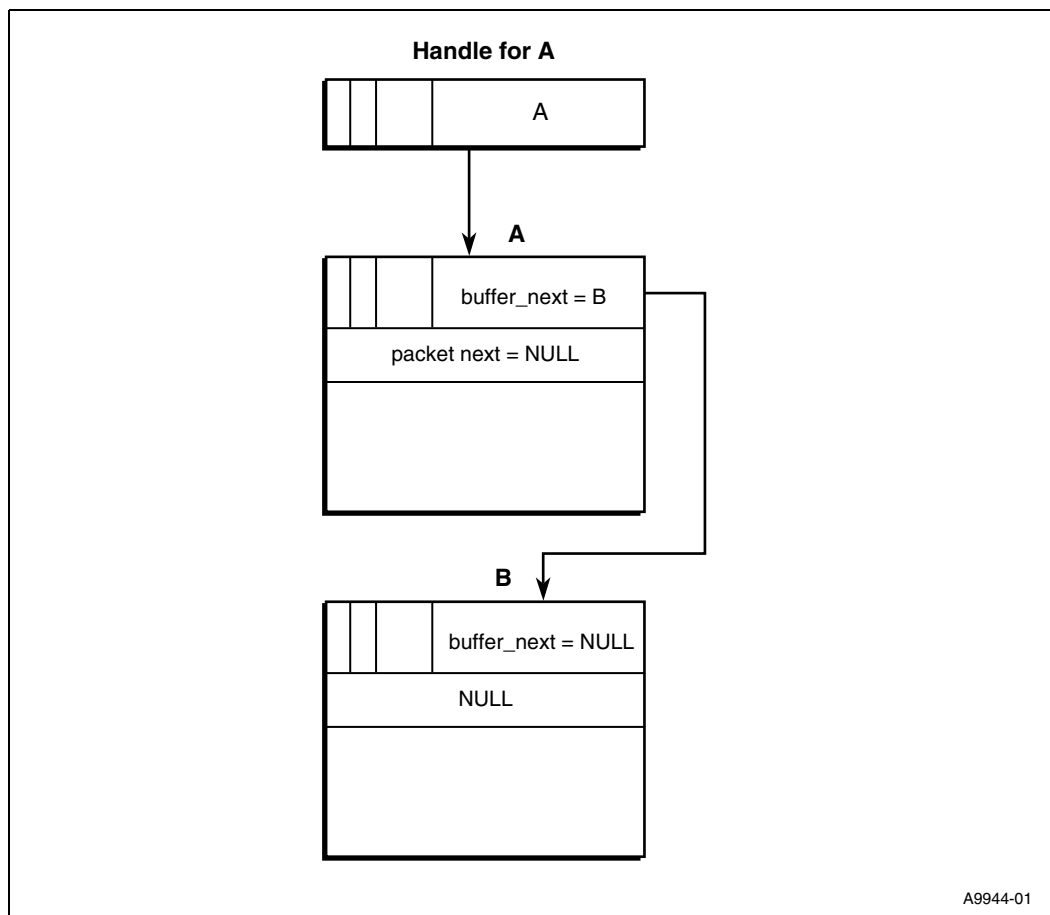
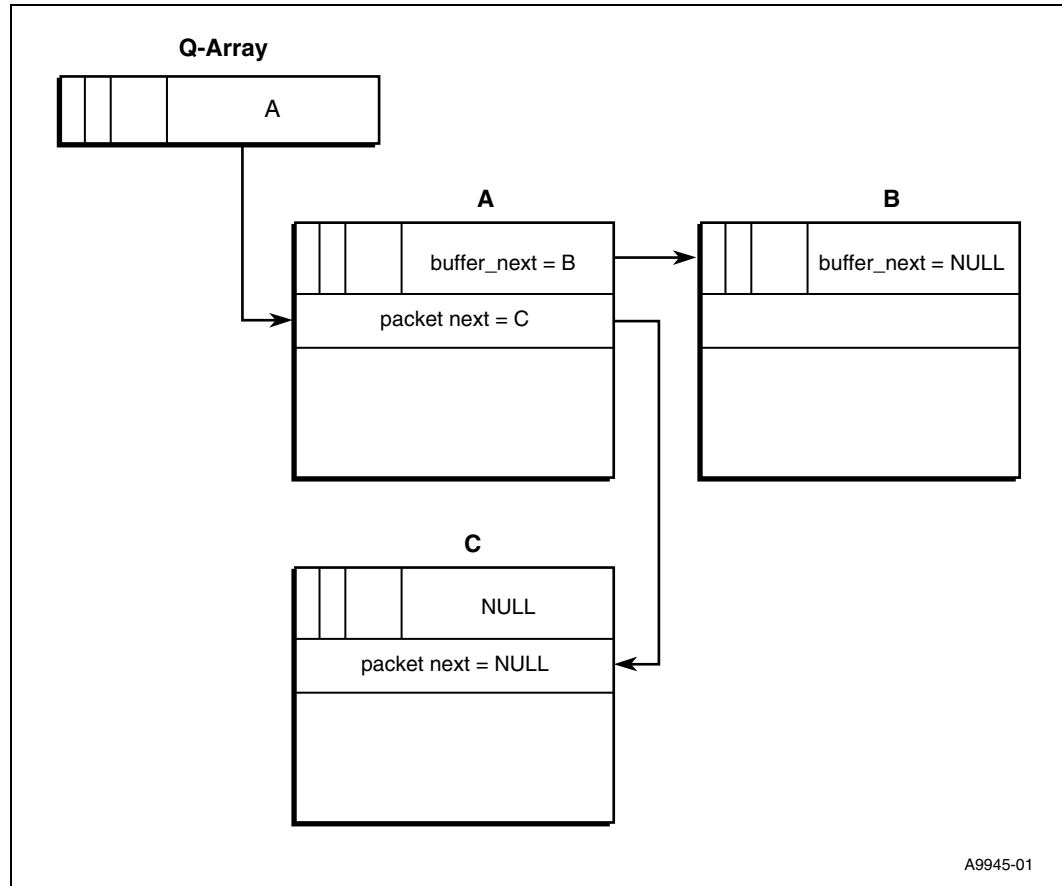


Figure 2-4 illustrates how the buffer descriptors A, B, and C containing two packets would be linked together using hierarchical queuing. In this mode, the Q-Array ignores the segment count and SOP/EOP bits in the buffer handle (the SRAM Control and Status Registers need to be programmed appropriately to configure the Q-Array in this mode).

Figure 2-4. Hierarchical queuing on the Egress IXP2400 (Multiple Packets)



2.5 Statistics and Handling of 64-bit Counters

This section describes two ways of handling 64-bit counters.

2.5.1 Using SRAM/Scratch Atomic Operations

Each microblock maintains statistics specific to the block e.g. packets received, dropped etc. The microblock always maintains a 32-bit counter in a location patched in by the corresponding Intel XScale® core component. This counter can be incremented very efficiently using the sram/scratch atomic increment operation, which is essentially fire and forget.

Note: SRAM atomic operations have a significant performance penalty on the current version of the IXP. Therefore, counters incremented using atomic operations is kept in scratch. Since scratch memory is limited, this method is not useful for supporting a large number of counters.

Depending on the data rate, the Intel XScale® core component periodically reads the value of the counter and does an atomic swap of zero into that location. The Intel XScale® core component maintains a separate 64-bit version of the same counter, which is incremented with the value of the 32-bit counter every time the atomic swap with zero is done. Control plane applications or SNMP agents running on the Intel XScale® may read this 64-bit counter.

For example, the Packet RX block keeps a Packets Received counter. At OC-48 line rates, 6.14 million packets may be received per second. This means a 32-bit counter overflows in $(4 \text{ billion} / 6.14 \text{ million}) = 651$ seconds. So every 400 (< 651) seconds the Intel XScale® core component for this block does an atomic swap with zero and update the associated 64-bit counter.

The byte counters overflow much faster. For example, at OC-48 line rates, where 6.14 million packets of 49 bytes each may be received per second. This means that a 32-bit counters overflow in $4 \text{ billion} / (6.14 \text{ million} * 49) = \text{approximately } 13$ seconds. This implies that separate timers may be needed to poll different types of counters.

Note: Not all counters need to be 64-bit, and it is a design decision which counters are 32-bit and which are 64-bit.

2.5.2 Read-Modify-Write in Critical Sections

An alternate way of implementing 64-bit counters is to update the counter in a critical section. This is useful for counters maintained in microblocks such as SRTCM, WRED etc. which implement critical sections to read, modify and write flow based data structures. In these microblocks, the counters may be stored in the flow-based data structures used to implement the SRTCM or WRED algorithm. The counters are read in with the rest of the data structure, modified and written back in the critical section. The disadvantage is that the number of instruction cycles in the critical section for these microblocks is increased.

Another method is to pass along the counter updates in a bit mask to the statistics microengine. The statistics microengine performs the counter updates based on the bit mask and the flow id stored in the packet meta data. It uses the CAM to cache flow information and implements the folding algorithm to read, modify and write the statistics data structure for a specified flow. This technique is more useful on the IXP2800, which has spare microengines.

2.6 Implementation of Dropping Packets

This section describes how packets buffers are dropped in this application. Packets that are marked to be dropped (`dl_next_block` is set to `IX_DROP`) are propagated through the pipeline until the `dl_sink[]` block is reached. If a packet consists of a single buffer, the buffer is simply enqueued on to the buffer free list (Q-Array entry 16).

If a packet consists of a chain of buffers, then it cannot be added directly to the free list. This is because the hardware does not guarantee that the enqueue of a chain of buffers is atomic with respect to dequeues and other enqueue operations. Therefore each buffer in the packet must be added separately to the free list. Since the number of buffers in a packet may be large, traversing a packet chain in the packet-processing pipeline to free the buffers in the chain can take a fairly high number of I/O operations. Therefore packet buffer chains are not freed in the packet-processing pipeline.

Instead the packet in the scratch ring is enqueued to the Queue Manager. The queue number in the enqueue request is Q-Array entry 17 which is used as the drop queue. This Q-Array entry (like the buffer free list) is permanently stored in the Q-Array hardware and is never swapped out to SRAM. Any enqueue requests to this Q-Array entry results in the packet being enqueued on this queue.

When the Queue Manager gets either a NULL enqueue request or a NULL dequeue request, it uses that free slot to dequeue a buffer from the Q-Array entry 17 and enqueue into the buffer free list (Q-Array 16). The rationale is that a large packet results in NULL slots in the pipeline, which may be used by the Queue Manager.

Note: In cell mode, since the cell count field is set in each buffer, a number of dequeue requests (until cell count goes to zero) may be required before the buffer is actually dequeued. To alleviate this, the Queue Manager sets the cell count field in the buffer handle for the SOP buffer to zero. The cell count for the remaining buffers is not modified.

2.7 Global Build Switches

Table 2-4 shows the build switches that may be applied to all projects. All these are turned off by default.

Table 2-4. Build Switches that May be Applied to all Projects

Symbol	Description
POTS	To turn on Pool of Threads support
_DEBUG_COUNTERS	To turn on debug counters
IXP_SIMULATION	To run the application on the transactor
USE_IMPORT_VAR	To run the application using XScale Core Components

System Data Structures for Packet Replication

3

This chapter describes the IXA Framework Extensions required to support packet replication. It describes system-wide data structures, as well as the system architecture required for implementing applications requiring packet replication.

Basic data structures which describe a packet are specified in [Chapter 2, “System Data Structures and Design Choices”](#). These structures include buffering architecture, packet meta data, buffer chaining, etc. In this chapter, extensions to the basic structures are described to perform efficient packet replication.

3.1 Packet Replication

Efficient packet replication is the key to supporting applications such as L2 VLAN Bridging and IPV4 Multicast Forwarding, at line rate. In such applications, multiple copies of the same packet are created and sent to multiple ports.

Efficient packet replication involves the following:

1. Packet replication must not result in copying of packet data. Copying of packet data across the DRAM buffers is an extremely costly operation.
2. Performance must scale with the number of multicast groups or number of VLANs, etc.
3. Performance must scale with the number of copies per packet.

3.1.1 Buffering Design

A packet buffer to be replicated (unicast or multicast or broadcast) consists of the following parts:

1. Data buffer which stores the packet data. This is located in DRAM.
2. Buffer descriptor for this data buffer which store the meta information. This is located in SRAM.

If a packet spans more than one buffer, then buffer chaining is employed as described in [Section 2.4](#). This packet is referred to as a “parent packet” and the buffers are referred to as “parent buffer(s)” throughout this document. The corresponding buffer handles are called “parent handle(s)” and the meta information is called “parent meta” information.

A replicated packet has three additional parts:

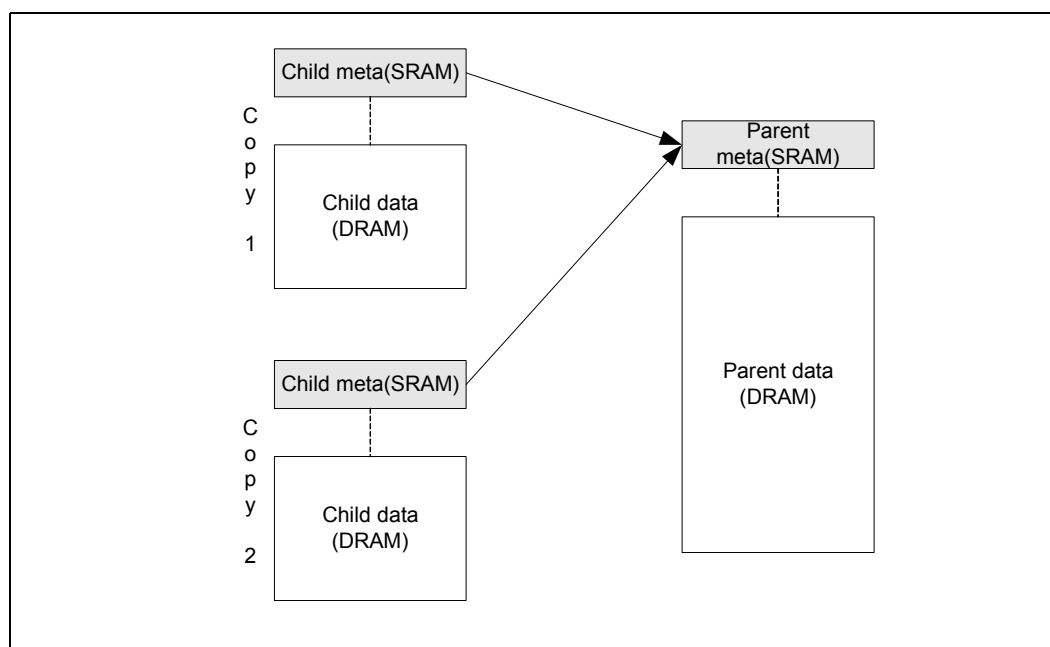
- Data buffer that stores copy-specific L2/L3 headers. This is located in DRAM.
- Buffer descriptor for this data buffer that stores the meta information. This is located in SRAM.
- In the buffer descriptor, a pointer to the parent packet.

These additional parts are collectively called the “child buffer” and the corresponding buffer handles are “child handles” throughout this document. Each child buffer consists of “child meta” and “child data” portions much like the parent buffers. However, the sizes of parent and child buffers may differ, depending on the application.

Note: Once the packet is replicated, queuing must be done using the child buffer handles only.

The packet replication functionality takes a parent packet and creates multiple child packets. Each child packet points back to the parent packet. The following figure shows an example of a parent packet with two children.

Figure 3-1. Packet Replication Example



3.1.2 Child Buffer

The child buffer is identified by a handle, called the child buffer handle. The child buffer consists of meta data in SRAM (Buffer Descriptor) and packet data area in DRAM. There is a 1 to 1 mapping between the buffer descriptor and packet data. This implies that one can be derived from the other using simple arithmetic.

The child buffer handle format is same as the regular buffer handle format, as described in [Section 2.4](#).

Currently the child buffers will be located in the same SRAM and DRAM channels as parent buffers. This will also result in minimal changes required in the dispatch loop library.

3.1.3 Cell Mode

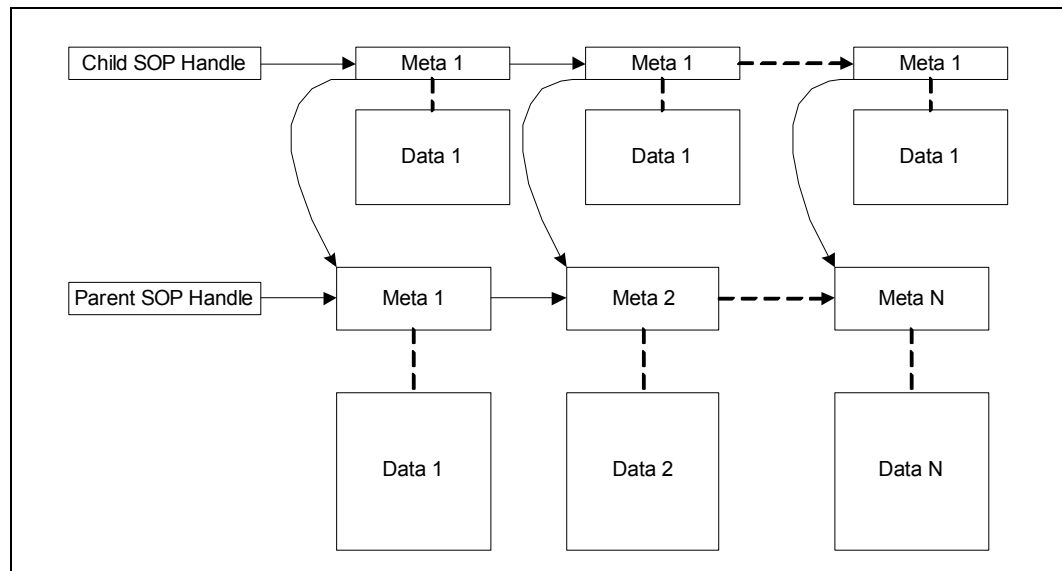
When buffers are chained in cell mode, the buffer handle encodes the cell count for the corresponding buffer. The Q-array hardware manipulates this cell count field. In this case, the Flat Queuing model is employed as described in [Section 2.4.1](#).

When a copy is created, the following things must be taken into consideration:

- Each parent buffer in the chain gets a corresponding child buffer.
- The child buffer inherits the cell count from the corresponding parent buffer.
- The cell count includes any headers in the child buffer data.
- The child buffers are also chained much like the parent buffers.
- The child buffer chain is enqueued to the Q-array instead of the parent buffer chain.

The above scheme facilitates enqueue/dequeue operations on each copy of the packet in cell mode. Figure 3-2 illustrates this scheme.

Figure 3-2. Packet Replication in Cell Mode

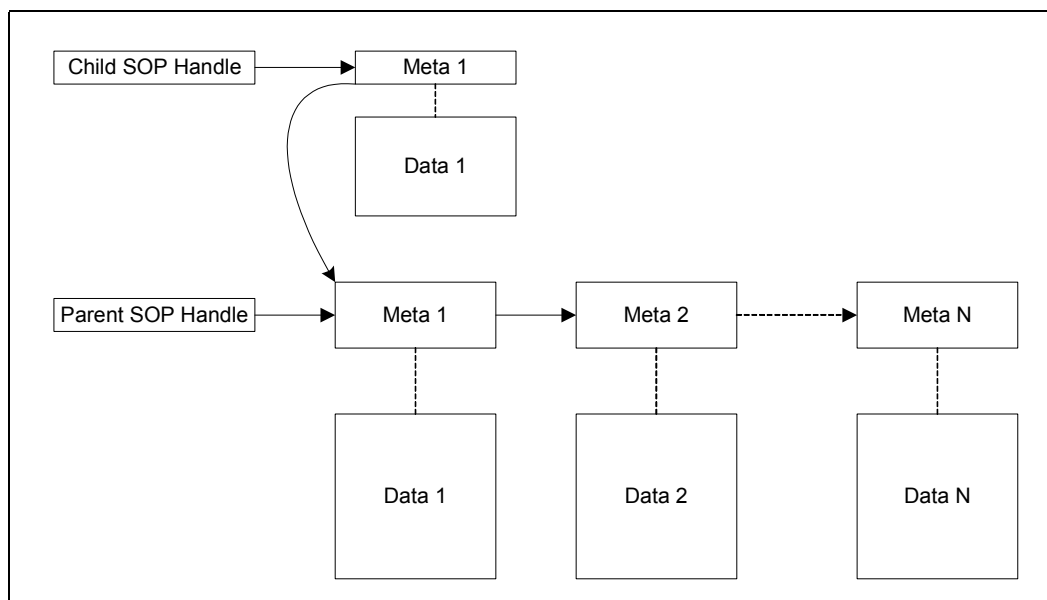


Since the packet header normally is in the first buffer, the data portions of child buffers are unused, except for the first one. This results in wastage of DRAM for these child buffers, as the buffering architecture requires a 1 to 1 mapping between the meta data and packet data to simplify calculations.

3.1.4 Packet Mode

In this mode, the Hierarchical Queuing model is employed as described in Section 2.4.2. In this mode, there is only one child buffer for the entire packet. When a parent packet spanning multiple buffers is to be enqueued, only the first buffer meta data is used for the enqueue operation. Hence when a copy of this packet is to be enqueued, we only need one child buffer. In this mode, the cell count field is ignored by the Q-array hardware. Figure 3-3 illustrates this scheme.

Figure 3-3. Packet Replication in Packet Mode



3.2 Parent and Child Meta Data (Buffer Descriptor)

This section describes the parent and child meta data formats when copies of packets are created. When a packet has no copies, the meta data of the parent buffer is as described in [Section 2.2](#). When a packet is replicated, the meta data format and size remain essentially the same, however, some fields are overloaded.

The rationale for this layout is as follows:

1. Packet Processing blocks such as IPV4 Forwarder, Diffserv blocks, etc. should not distinguish between unicast or multicast packets if they don't process them differently.
2. All changes due to packet replication should be hidden in dispatch loop functions or driver blocks. As a result, packet processing blocks need minimal changes, if any.
3. There should be no performance impact on unicast packet processing even after enabling the extensions to support packet replication.

3.2.1 Cell Mode

The following sections describe the child and parent meta data structures for replicated packets using cell mode.

3.2.1.1 Child Meta Data

Table 3-1 lists the fields of the child meta data structure when operating in cell mode.

Note: The fields shown in **bold text** are different than the standard meta data fields.

Table 3-1. Child Meta Data in Cell Mode

LW	Bits	Size	Description
0	31:0	32	Next child handle. Points to next child handle if one exists.
1	31:16	16	Parent buffer size in bytes.
1	15:0	16	Offset of start of data in the parent buffer in bytes.
2	31:16	16	Packet size. Size of the packet across all buffers including child buffers.
2	15:12	4	Freelist ID for parent buffer.
2	11:8	4	Rx status.
2	7:0	8	Type of header at offset in child buffer if data exists in child buffer. Otherwise, it is for parent buffer.
3	31:16	16	Input port number.
3	15:0	16	Output port number.
4	31:16	16	Next hop ID.
4	15:8	8	Fabric port number.
4	7:4	4	Reserved.
4	3:0	4	Next hop ID type.
5	31:24	8	Color.
5	23:0	24	Flow ID.
6	31:16	16	Class ID.
6	15:0	16	Parent Buffer ID. Given a buffer handle, this value can be figured out using a simple calculation, as follows: Buffer ID = ((parent_handle & 0xFFFFF) – META_BASE)/ (sizeof(parent meta)) The maximum number of buffers supported is 16K.
7	31:22	10	Offset of start of data in the child buffer, in bytes.
7	21:14	8	Size of data in child buffer.
7	13:4	10	Reserved.
7	3:0	4	Free list ID for child buffer.

3.2.1.2 Parent Meta Data

Table 3-2 lists the fields of the parent meta data structure when operating in cell mode.

Note: The XXX in Table 3-2 indicates that these fields are a “don’t care,” as this information is overridden by the information in the child meta data structure.

Table 3-2. Parent Meta Data in Cell Mode when Packet is Replicated

LW	Bits	Size	Description
0	31:0	32	Next handle. Points to next parent buffer if one exists.
1 thru 6	31:0	32	XXX
7	31:16	16	Reserved.
7	15:0	16	Reference count. Indicates how many children exist for this buffer.

3.2.2 Packet Mode

The following sections describe the child and parent meta data structures for replicated packets using packet mode.

3.2.2.1 Child Meta Data

Table 3-3 lists the fields of the child meta data structure when operating in packet mode.

Note: In packet mode, LW 0 is not used to link child buffer handles. The parent buffer meta data must be used to walk through the buffer chains.

Table 3-3. Child Meta Data in Packet Mode

LW	Bits	Size	Description
0	31:22	10	Offset of start of data in the child buffer in bytes
0	21:14	8	Size of data in child buffer
0	13:4	10	Reserved.
0	3:0	4	Free list ID for child buffer
1	31:16	16	Parent buffer size in bytes.
1	15:0	16	Offset of start of data in the parent buffer in bytes
2	31:16	16	Packet size. Size of the packet across all buffers including the multicast buffer.
2	15:12	4	Freelist ID for parent buffer
2	11:8	4	Rx status.
2	7:0	8	Type of header at offset in child buffer if data exists in child buffer. Otherwise, it is for parent buffer.
3	31:16	16	Input port number
3	15:0	16	Output port number
4	31:16	16	Next hop ID
4	15:8	8	Fabric port number
4	7:4	4	Reserved.
4	3:0	4	Next hop ID type
5	31:24	8	Color
5	23:0	24	Flow ID

Table 3-3. Child Meta Data in Packet Mode (Continued)

LW	Bits	Size	Description
6	31:16	16	Class ID
6	15:0	16	Parent buffer ID. Given a buffer handle, by simple calculations this value can be figured out. The calculation used is as follows. Buffer ID = ((parent_handle & 0xfffff) – META_BASE)/(sizeof(parent meta)) The maximum number of buffers supported is 16K.
7	31:0	32	Packet Next

3.2.2.2 Parent Meta Data

Table 3-4 lists the fields of the parent meta data structure when operating in packet mode.

Note: The XXX in Table 3-4 indicates that these fields are a “don’t care,” as this information is overridden by the information in the child meta data structure.

Table 3-4. Parent Meta Data in Packet Mode when Packet is Replicated

LW	Bits	Size	Description
0	31:0	32	Next handle. Points to next parent buffer if one exists.
1 thru 6	31:0	32	XXX
7	31:16	16	Reserved.
7	15:0	16	Reference count. Indicates how many children exist for this buffer.

3.3 Child Buffer Freelist

As described in the previous sections, packet replication requires the creation of child buffers and buffer chains. There are some important points to note:

1. The number of child buffers required are much more than the parent buffers.
2. The child buffers are typically used to store any copy-specific headers and are therefore typically small.
3. In certain applications, it is possible to store the copy-specific header along with the child meta data for efficiency purposes. This feature is not part of the IXA SDK release.

For these reasons, an additional free list is required to maintain the child buffers.

3.4 Packet Copier

A Packet Copier microblock is included in the IXA SDK software to perform packet replication, because the amount of processing required to perform packet replication warrants a microblock on its own. Since the processing amount varies with the number of copies required and the mode (cell vs. packet), this block cannot share the same thread as other packet processing or driver blocks.

For this reason, packets identified as multicast will be sent to this block to make the necessary copies. Then the copies may re-enter the packet processing pipeline where they are subjected to further header processing and QoS treatment such as Metering, Scheduling, etc. It is possible to run this microblock in fewer threads to cater to lower rates of multicast traffic. In this case, the same microengine can be shared by other microblocks.

The interfaces to the Packet Copier microblock are described in [Chapter 37, “Packet Copier Microblock”](#).

3.5 Dropping of Packet Copies

[Section 2.6](#) describes the standard situation for dropping packets without copies. This section describes the implementation of packet dropping for replicated packets.

3.5.1 Dropping Packet Copies in Packet Mode

In packet mode, a packet copy is dropped as follows:

1. Free the child buffer to its freelist.
2. Decrement reference count in parent SOP buffer only, because in packet mode only the SOP buffer has a reference count field.
3. If the reference count is zero, free the parent buffer to its freelist.
4. For subsequent buffers in the packet, simply free them to their freelist. There are no reference counts for these buffers.

3.5.2 Dropping Packet Copies in Cell Mode

In cell mode, a packet copy is dropped as follows:

1. Free the child buffer to its freelist.
2. Decrement the reference count in its parent buffer.
3. If the reference count is zero, then free the parent buffer to its freelist.

3.6 Differentiating Child Buffer from Parent Buffer

It is not possible to differentiate a child buffer from a parent buffer by simply looking at the buffer handle, because the type of handle cannot be encoded in the handle itself.

However, once the meta data is read, it is possible to differentiate by looking at the Rx status field in the meta data as shown in [Figure 3-4](#). The valid combinations are described in [Table 3-5](#).

Note: The parent and child buffer meta data have to be set accordingly.

Figure 3-4. RX Status Field Definition

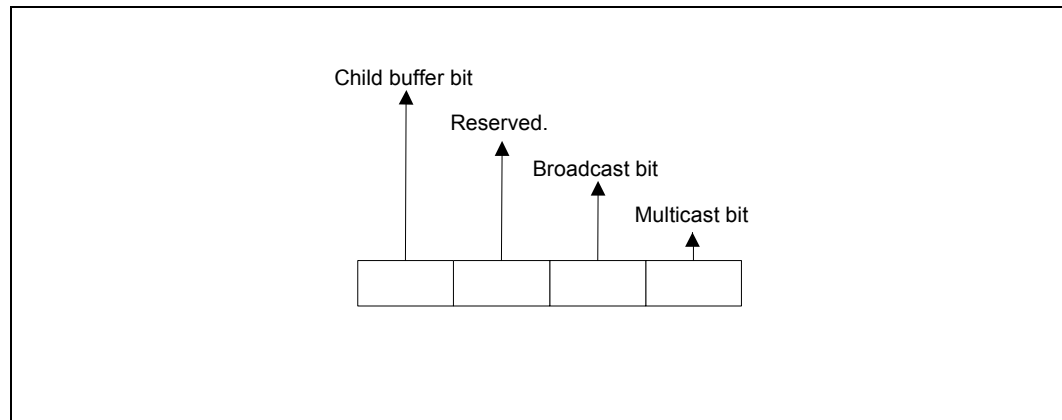


Table 3-5. Rx Status Valid Values

Field Value	Description
0x0	Parent buffer, unicast packet
0x1	Parent buffer, multicast packet
0x2	Parent buffer, broadcast packet
0x3 – 0x7	INVALID
0x8	Child buffer, unicast packet
0x9	Child buffer, multicast packet
0xa	Child buffer, broadcast packet
0xb	INVALID
0xc	Reserved.
0xd	Reserved.
0xe – 0xf	INVALID

3.7 Build Switches

The packet replication capability is turned on by defining the switch `MULTICAST_SUPPORT_ENABLE`.

Microblocks

The Intel[®] Internet Exchange Architecture Software Development Kit (Intel[®] IXA SDK) specifies how to develop modular software building blocks (microblocks) in microC or microcode on the microengines. These blocks may be combined with other software blocks provided by Intel (and possibly third parties) to build different applications such as line-cards for MPLS, DSLAMs, Multi-Service Switches and so on. Associated with each microblock is a slow path component on the Intel XScale[®] processor (core component) that configures the microblock and handles exception packets. The IXA Portability Framework also specifies how to write the Intel XScale[®] core component and how to interface it with the corresponding microblock.

This document focuses on software building blocks for

- Sending and Receiving packets over POS, Ethernet and ATM media interfaces
- IPv4, IPv6 Forwarding, MPLS and DiffServ
- Sending and Receiving packets over the CSIX switch fabric
- Scheduling and queuing packets, Traffic Management and Shaping

It also illustrates how these blocks may be combined to create simple applications, such as POS IPv4, Ethernet and ATM.

The following microblocks are discussed in this document:

- [Receive](#)
 - [Chapter 4, “Packet RX Microblock”](#)
 - [Chapter 5, “CSIX RX Microblock”](#)
 - [Chapter 6, “ATM AAL5 RX Microblock”](#)
- [Transmit](#)
 - [Chapter 7, “CSIX TX Microblock”](#)
 - [Chapter 8, “Packet TX for SPHY and MPHY-4”](#)
 - [Chapter 9, “Packet TX for MPHY-16”](#)
 - [Chapter 10, “Packet Transmit for OC-192 POS”](#)
 - [Chapter 11, “ATM AAL5 TX Microblock”](#)
 - [Chapter 12, “Packet TX–Multiports Microblock”](#)
- [Queue Manager](#)
 - [Chapter 13, “Queue Manager For OC-48 Microblock”](#)
 - [Chapter 14, “Queue Manager For OC-192 Microblock”](#)
 - [Chapter 15, “Packet Queue Manager Microblock”](#)
 - [Chapter 16, “ATM Queue Manager Microblock”](#)

- Scheduler Microblocks
 - Chapter 17, “Fabric Scheduler For OC-48”
 - Chapter 18, “Fabric Scheduler For OC-192”
 - Chapter 19, “OC-48 WRR/DRR Packet Scheduler”
 - Chapter 20, “OC-192 DRR Egress Scheduler”
 - Chapter 21, “Egress Queue Manager (DiffServ) Microblock”
 - Chapter 22, “Egress Scheduler (DiffServ) Microblock”
 - Chapter 23, “TM4.1 Shaper and Scheduler Microblock”
- Forwarder
 - Chapter 24, “IPv4 Forwarder Microblock”
 - Chapter 25, “IPv6 Forwarder Microblock”
 - Chapter 26, “IPv6 To IPv4 Tunneling Microblock”
 - Chapter 27, “IPv6 To IPv4 Translation Microblock”
- Layer 2
 - Chapter 28, “Layer-2 Decapsulation and Classify”
 - Chapter 29, “Layer-2 Encapsulation”
- DiffServ
 - Chapter 30, “6-tuple Exact Match Classifier Microblock”
 - Chapter 31, “Three Color Meter Microblock”
 - Chapter 32, “DSCP Marker Microblock”
 - Chapter 33, “DSCP Classifier Microblock”
 - Chapter 34, “Weighted Random Early Detection (WRED) Microblock”
- MPLS Microblocks
 - Chapter 35, “FTN Forwarder Microblock”
 - Chapter 36, “ILM Forwarder Microblock”
- Services Microblocks
 - Chapter 37, “Packet Copier Microblock”
 - Chapter 38, “Freelist Manager”

Receive

The Receive microblocks section includes the following chapters:

- [Chapter 4, “Packet RX Microblock”](#)
- [Chapter 5, “CSIX RX Microblock”](#)
- [Chapter 6, “ATM AAL5 RX Microblock”](#)

Receive microblocks interface with the MSF and reassemble incoming m-packets (RBUFS) into complete packets for further processing by down stream packet processing code. For each packet, the packet data is written to DRAM, the packet descriptor (meta-data) is written to SRAM and a handle to the packet is written to a scratch ring for use by the packet processing stage.

The table below summarizes the different Receive microblocks supported on the SDK:

Microblock	Description	Usage	Cycle Budget
Packet Receive	For PPP/POS and Ethernet frames.	May be run on one or two microengines depending on the data rate. For IXP2400 OC-48, 4 GBE and IXP2800 10GBE data rates, a single microengine is used. For OC-192 data rates 2 microengines in a pipeline are used	57 per microengine for OC-192 POS on IXP2800. Otherwise 97 for OC-48 POS on IXP2400 or 94 for 10 GBE on IXP2800
CSIX Fabric Receive	Reassembles cframes into packets. Supports up to 1024 VoQs	May be run on one or two microengines depending on the data rate. For IXP2400 OC-48 and 4 GBE data rates, a single microengine is used. For OC-192 and 10 GBE data rates 2 microengines in parallel are used	57 per microengine for OC-192 POS on IXP2800. Otherwise 97 for OC-48 POS on IXP2400 or 94 for 10 GBE on IXP2800
ATM AAL-5 Receive	For ATM interface. Supports AAL-5 Reassembly. May be combined with AAL-2 microblock	Supports OC-48 data rates with 2 microengines on the IXP2400 and 1 microengine with the IXP2800.	105 per microengine for OC-48 on the IXP2400 and 248 per microengine for OC-48 on the IXP2800

4.1 Overview

The Packet Receive microblock performs frame-reassembly on the mpackets coming in on a POS or Ethernet media interface. The microblock is written such that it supports up to 16 virtual ports, one or more of which may be unused. This allows the microblock to support different configurations such as Quad-OC12, 16 OC-3 or a single OC-48 port on the IXP2400 and Quad OC-48, 16 OC-12 or a single OC-192 port on the IXP2800.

The microblock moves packet data from RBUF to DRAM and also writes packet descriptor (or packet metadata) information into SRAM or into a scratch ring for the next processing stage. Since POS and Jumbo Ethernet frames may be up to 9k bytes, some large packets may be stored in multiple buffers chained together as a link-list. The buffer handles for the first and last packet in the chain are queued in the scratch ring for the next processing stage.

This microblock supports two configurations. It can run entirely on one microengine (8 threads running in parallel) or it can be run on two microengines in a context pipeline connected by a next neighbor ring. The single microengine design is used for the IXP2400 and for Gigabit Ethernet on the IXP2800. The two microengine design is used for supporting OC-192 POS data rates.

4.2 Assumptions and Dependencies

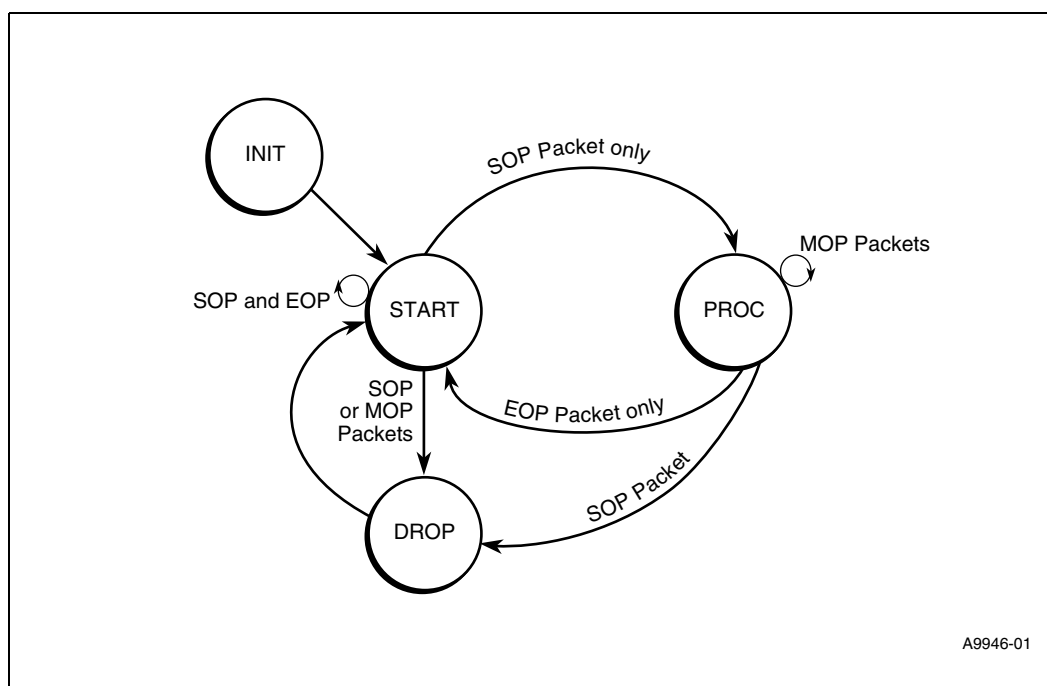
The following assumptions are made in the design and implementation of the Packet RX block:

- The Packet RX block supports up to 16 virtual ports. This implies that the Receive Reassembly Context (RXC) may be entirely stored in local memory. Finding the RXC is as simple as using the port number and RXC size to index into local memory.
- For every mpacket received, the Packet RX block moves the packet data from the RBUF into DRAM. Subsequent stages in the pipeline read the header data from DRAM when needed. They also decide the size of the read burst depending on the application requirements. Once the Packet RX block gets a signal that the data from the RBUF has been written into DRAM, it frees up the RBUF element by putting it back on the element free list. The IP header is not processed directly from the RBUF.
- The Packet RX block needs to maintain the sequence in which packets arrive. Buffer handles for packets are queued into the scratch ring in the order in which they arrive on the MSF media interface. This is achieved by making sure that the threads execute in order.
- The RBUF element size selected must be either 128 or 256 bytes. The 64-byte option is not used. This allows us more cycle budget to handle MOP and EOP only mpackets (refer to [Section 4.8, “Performance Analysis” on page 91](#)).
- The Packet RX block computes and sets up the cell count for every buffer. The cell size is a compile-time option with a default of 120 bytes. In the computation of the cell count, the Packet RX block adds, for SOP buffer only, `PREPEND_BYTES` to the buffer size. This allows the CSIX transmit block to add a per-packet header of size `PREPEND_BYTES` to every packet. The cell count computation may be turned off via a compile time defined for use in applications where the output port is not a cell based media interface.

- For the SOP buffer, the Packet RX block starts writing the packet data into DRAM at offset equal to 128—or more, see [Section 2.3, “Optimizing DRAM Bank Scheduling for Buffers” on page 59](#) for details. The first 128 bytes of the SOP buffer is always left empty to allow blocks down the pipeline to add and remove headers—note that these bytes are not included in the computation of the cell count. For the non-SOP buffers, the packet data is written into DRAM at offset equal to zero.

4.3 Packet RX State Machine

Figure 4-1. Packet RX State Machine



SOP—first mpacket of PPP Packet.

MOP—middle mpacket of PPP packet.

EOP—last mpacket of PPP packet.

There are two unstable states and two stable states in the Packet RX state machine.

- INIT**—This is an unstable state. In this state, the Packet RX micro code initializes context descriptors and immediately transfer to the START state. Only thread 0 goes through this state and other threads goes directly to the START state.
- START**—This is a stable state. In this state, the Packet RX microblock checks the incoming mpacket. If the mpacket is a SOP-only mpacket, the Packet RX microblock transfers to the PROC state. If the mpacket is an EOP or MOP mpacket, the Packet RX microblock transfers to the DROP state and drop the mpacket. If the incoming mpacket is an SOP and EOP mpacket, this mpacket contains a complete PPP or Ethernet packet. In this case, Packet RX

microblock sends the packet to the `dl_sink[]` microblock, stay at the START state and wait for the new mpacket to arrive.

- **PROC**—This is the stable processing state. In this state, Packet RX microblock checks the incoming mpacket. If the mpacket is an MOP mpacket, the Packet RX microblock writes the mpacket into DRAM and stay in this state. If the incoming mpacket is an SOP mpacket, this is a back-to-back SOP mpacket. In this case, the Packet RX micro code transfers to the DROP state, drop the current buffer/buffer chain being reassembled, transfer back to the START state and write the new incoming SOP mpacket into another DRAM buffer. If the mpacket is an EOP mpacket, the Packet RX micro code writes the mpacket into DRAM, send the reassembled packet to the `dl_sink[]` microblock and transfer to the START state.
- **DROP**—This is an unstable state. In this state, the Packet RX micro code drops the mpacket—at times, all mpackets already in the receiving buffer—and transfers to the START state immediately.

This state machine only describes the basic processing procedure. For further details refer the flow chart in [Section 4.7, “Flow Chart For Single Microengine Design”](#) on page 87.

4.4 Data Structures

4.4.1 Receive Reassembly Context (RXC)

Table 4-1 specifies the Reassembly Context (RXC) which is maintained by every thread.

Table 4-1. Receive Reassembly Context

LW	Bits	Size	Field	Description
0	31:1	31	reserved	Reserved
0	0:0	1	state	If 1 PROC state else START state
1	31:0	32	Pkt_destination	Where the pkt goes next. (IPV4, Core etc)
2	31:0	32	Sop_buf_size	Size of sop Buffer. (i.e bytes in sop buffer)
3	31:0	32	Prev_buf_size	Size of previous buffer.
4	31:0	32	Curr_data_ptr	Current DRAM pointer for current buffer
5	31:0	32	Buf_size	Size of the current buffer in bytes
6	31:0	32	Buf_offset	Offset where the data begins
7	31:0	32	Packet_size	Total packet size across buffers
8	31:0	32	Error_count	Error packets received for this port
9	31:0	32	Sop_buf	Buffer handle of SOP buffer
10	31:0	32	Cell_count	Cell count for current buffer
11	31:0	32	Byte_remaining	Bytes left in last cell ($\text{buffer_size} \% \text{cell_size}$)
12	31:0	32	Prev_buf_handle	Handle to previous buffer
13	31:0	32	Sop_buf_offset	Offset where data begins in SOP buffer. (this vary between 128 and 512 because of dram bank scheduling)
14	31:0	32	Cur_buf	Current buffer.

Table 4-1. Receive Reassembly Context

LW	Bits	Size	Field	Description
15	31:24	8	Port header type	Header type of data coming in on this port
15	23:16	8	Reserved	Reserved
15	15...0	16	Max_buf_size	Port first buffer maximum data size

4.4.2 Statistics

The Packet RX block maintains the following statistics on a per port basis. The microblock maintains these as 32 bit counters and the Intel XScale® core component keeps a 64-bit version of these counters. Refer to [Section 2.5, “Statistics and Handling of 64-bit Counters”](#) on page 63.

Table 4-2. Statistics Counter Offsets from Base for Port

Counter	Offset
Packets Received	0x0
Bytes Received	0x4
Packets dropped	0x8
Packets to XScale core	0xc

4.4.3 Jump Table

The Packet RX microcode uses a jump table based on the current state and the SOP/EOP bit for the current mpacket. For more details refer the flow chart in [Section 4.7](#). [Table 4-3](#) explains each entry in the jump table.

Table 4-3. Jump Table Entries

Entry Name	Description	Valid/Error
SESP (Expect SOP, Get SOP/EOP)	Expected SOP mpacket, received a mpacket with both SOP and EOP bits set	Valid
SEPP (Expect MOP/EOP, Get SOP/EOP)	Expected either a MOP or a EOP mpacket, received a mpacket with both SOP and EOP bits set	Error
MSP (Expect SOP, Get MOP)	Expected SOP mpacket, received MOP mpacket	Error
ESP (Expect SOP, Get EOP)	Expected SOP mpacket, received EOP mpacket	Error
SPP (Expect MOP/EOP, Get SOP)	Expected either a MOP or EOP mpacket, received a SOP mpacket	Error
SSP (Expect SOP, Get SOP)	Expected a SOP mpacket, received a SOP mpacket	Valid
MPP (Expect MOP/EOP, Get MOP)	Expected either a MOP or EOP mpacket, received a MOP mpacket	Valid
EPP (Expect MOP/EOP, Get EOP)	Expected either a MOP or EOP mpacket, received a EOP mpacket	Valid

4.5 Build Switches

Table 4-4 shows the compile time build switches are relevant to the Packet Receive Microblock.

Table 4-4. Compile Time Build Switches Relevant to the Packet Receive Microblock

Symbol	Description
TWO_ME_PACKET_RX	To run Packet RX as a two microengine pipeline. Default is to run it on a single microengine
FIRST_PACKET_RX_ME	The first microengine in the pipeline
SECOND_PACKET_RX_ME	The second microengine in the pipeline
RX_PHY_MODE	Relevant only to the IXP2400. May be set to SPHY_1_32, (Single port)MPHY_4 (4 ports, 2 threads per port)SPHY_4_8 (4 ports, 2 threads per port)MPHY_16 (16 ports)
NO_CELL_COUNT	In this mode, the Packet RX block does not add cell count to the buffer handle
DISABLE_MAC_FILTERING	Used for Ethernet. In this mode, the Ethernet header is not written to DRAM.
PACKET_RX_COUNTERS	Turn on Counters. Note that there is no need to turn on counter handling in the microblock. The XScale Core also maintains counters for Packet RX by querying the media device.

4.6 Algorithm

This section describes the high level algorithm for the Packet Receive block.

4.6.1 Single Microengine Design

Phase 1

- The thread is woken up from the thread free list by the MSF.
- The thread looks into the Receive Status Word (RSW) to find the port #, RBUF element number and size.
- Based on the port #, the thread finds the Receive Context and determines which code path to take (SESP, SSP, MPP etc. as specified in the above table).
- The thread copies the RBUF into DRAM.
- For SOP, the metadata is written to SRAM
- For EOP m-packets and in the case of a new buffer being allocated, the first two words of the buffer descriptor are flushed to SRAM.
- For SOP and SOP/EOP—that is, if a buffer is used, another is pre-fetched from the free list.
- The thread waits for previous thread signal and all I/O initiated.

Phase 2

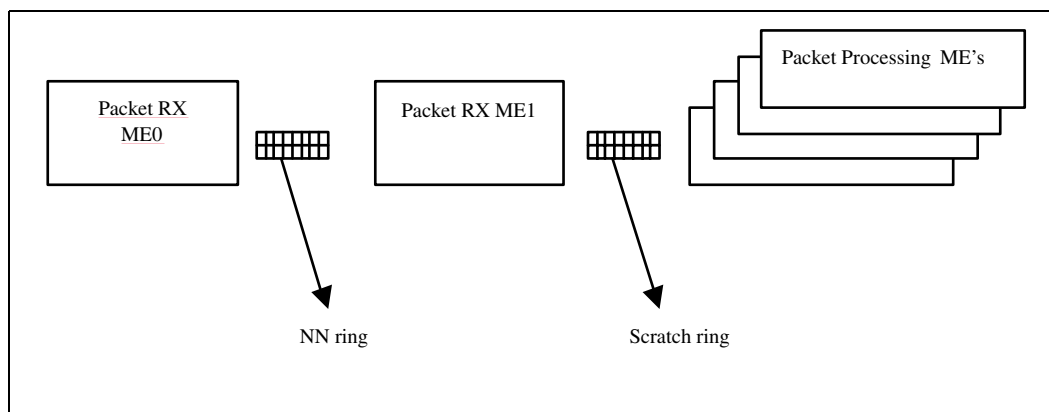
- The RBUF element is freed and the thread is written back to the free list.

- If EOP, the thread writes the packer handle and descriptor information into the scratch ring for processing in the next stage.
- The thread sends a signal to the next thread to enter phase 2.
- The thread waits for the scratch write to complete and for the MSF Receive event signal.

4.6.2 Two Microengine Design

Figure 4-2 illustrates the Packet Receive on two microengines.

Figure 4-2. Packet Receive on Two Microengines



Figures 4-3 and 4-4 shows the design for the two microengine version.

Figure 4-3. Design for the First Receive Microengine

```

First receive microengine
Phase 1
  • Read packet status from Receive Status Word. Do error checks and check type
    of m-packet (SOP/EOP, SOP only etc.)
  • Get the Reassembly Context (RXC) local memory pointer using the port number
    in the receive status word
  • Update RXC in local memory
  • if (buffer is consumed) {
    Write buffer meta data to SRAM
    Pre-fetch another buffer for the thread
  }
  • Wait (SRAM meta write and buffer pre-fetch if any AND previous thread
    signal)
Phase 2
  • Signal next thread
  • Write message to NN ring of next receive microengine
  • Put thread on free list
  • Wait (receive event)
  
```

Figure 4-4. Design for the First Receive Microengine

```
Second receive Microengine
Phase 1
· Read from NN ring
· Write RBUF element to DRAM buffer
· Wait (DRAM write, previous thread signal)
Phase 2
· Signal next thread
· Free RBUF element
· if (EOP)
{
    Sink block writes to scratch ring
}
· Wait (scratch put, previous thread signal)
```

4.7 Flow Chart For Single Microengine Design

The following figures in this sub-section are a three-page flowchart for the POS RX functionality.

Figure 4-5. POS Receive Flowchart: Page 1 of 3

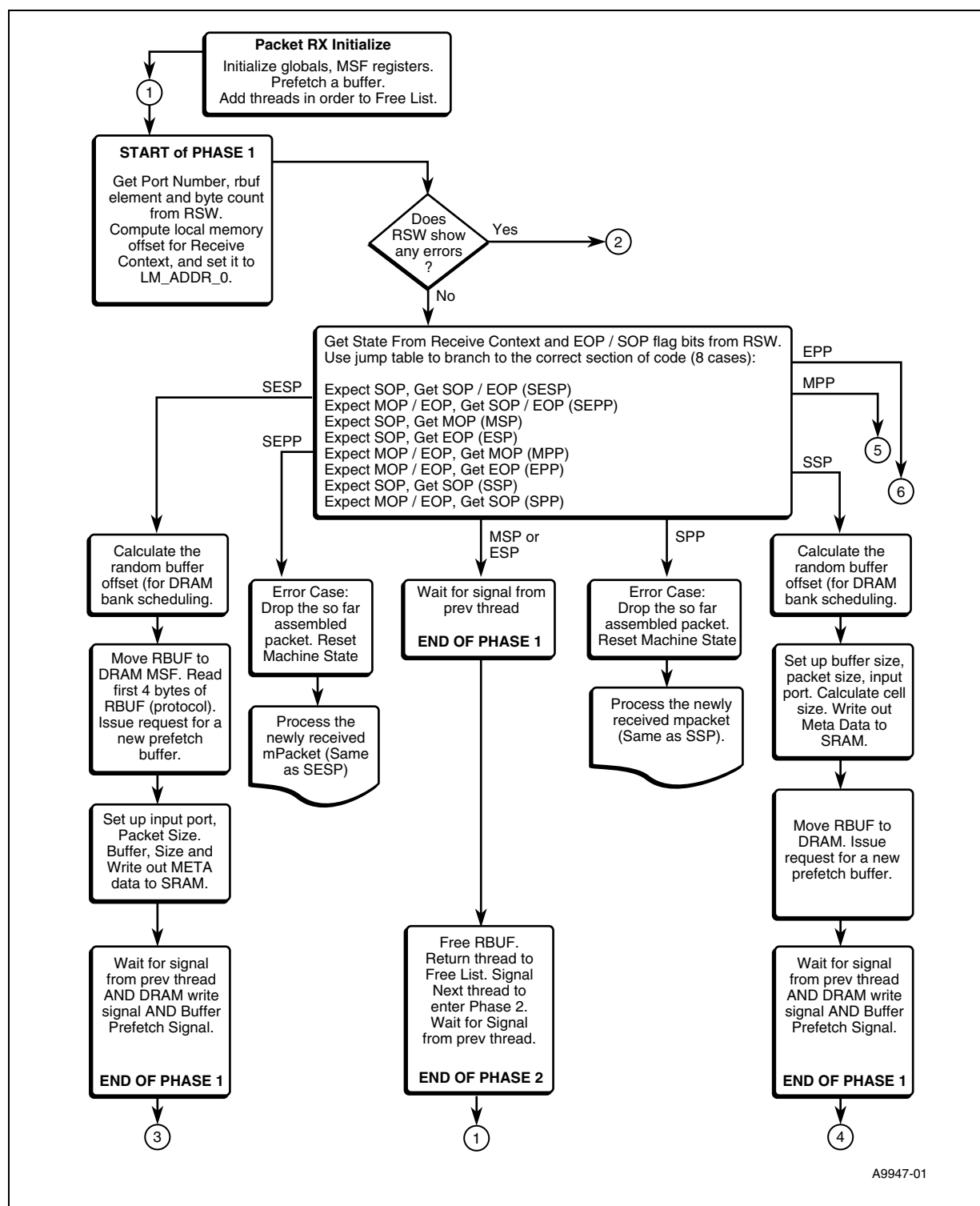


Figure 4-6. POS Receive Flowchart: Page 2 of 3

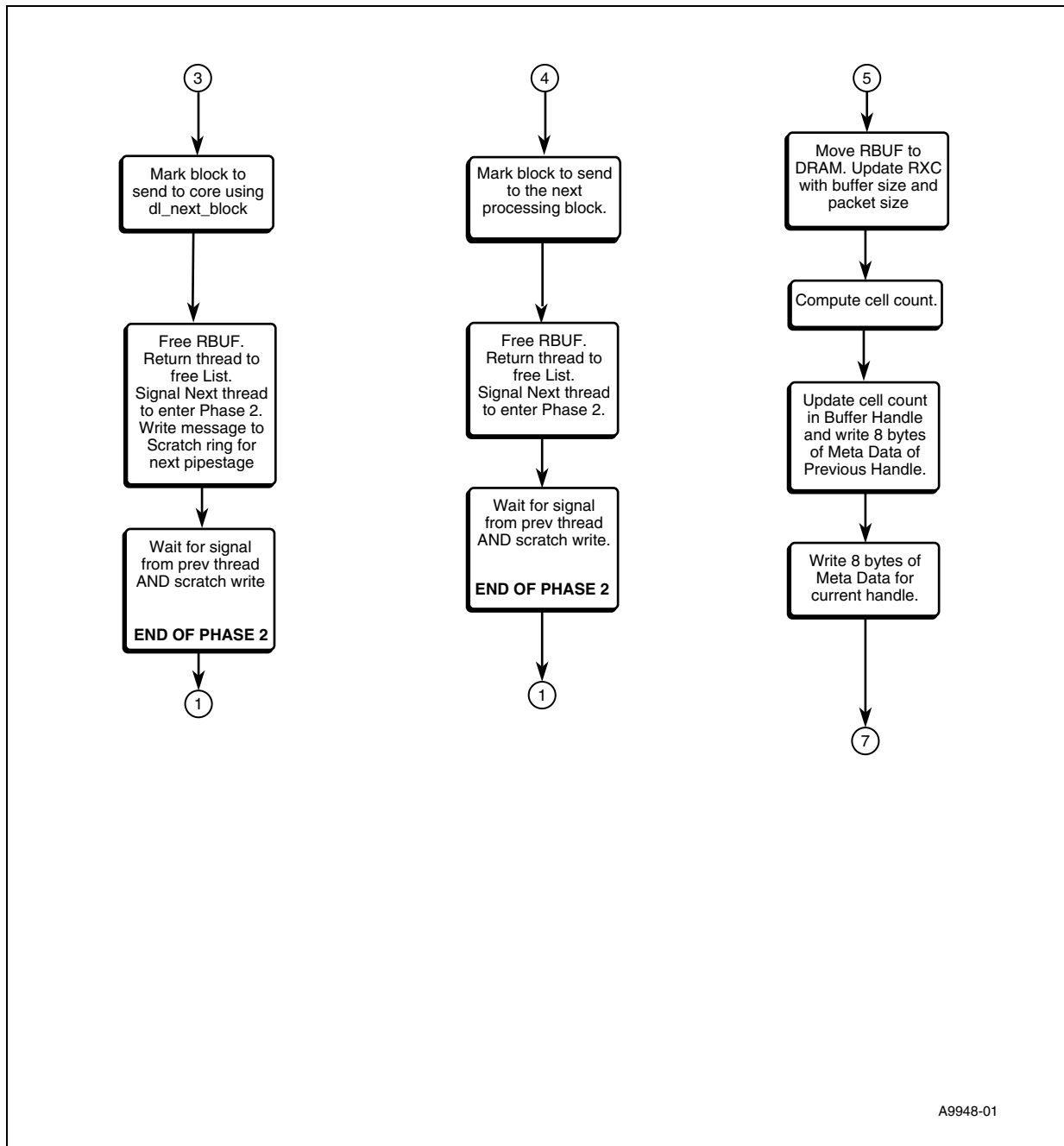
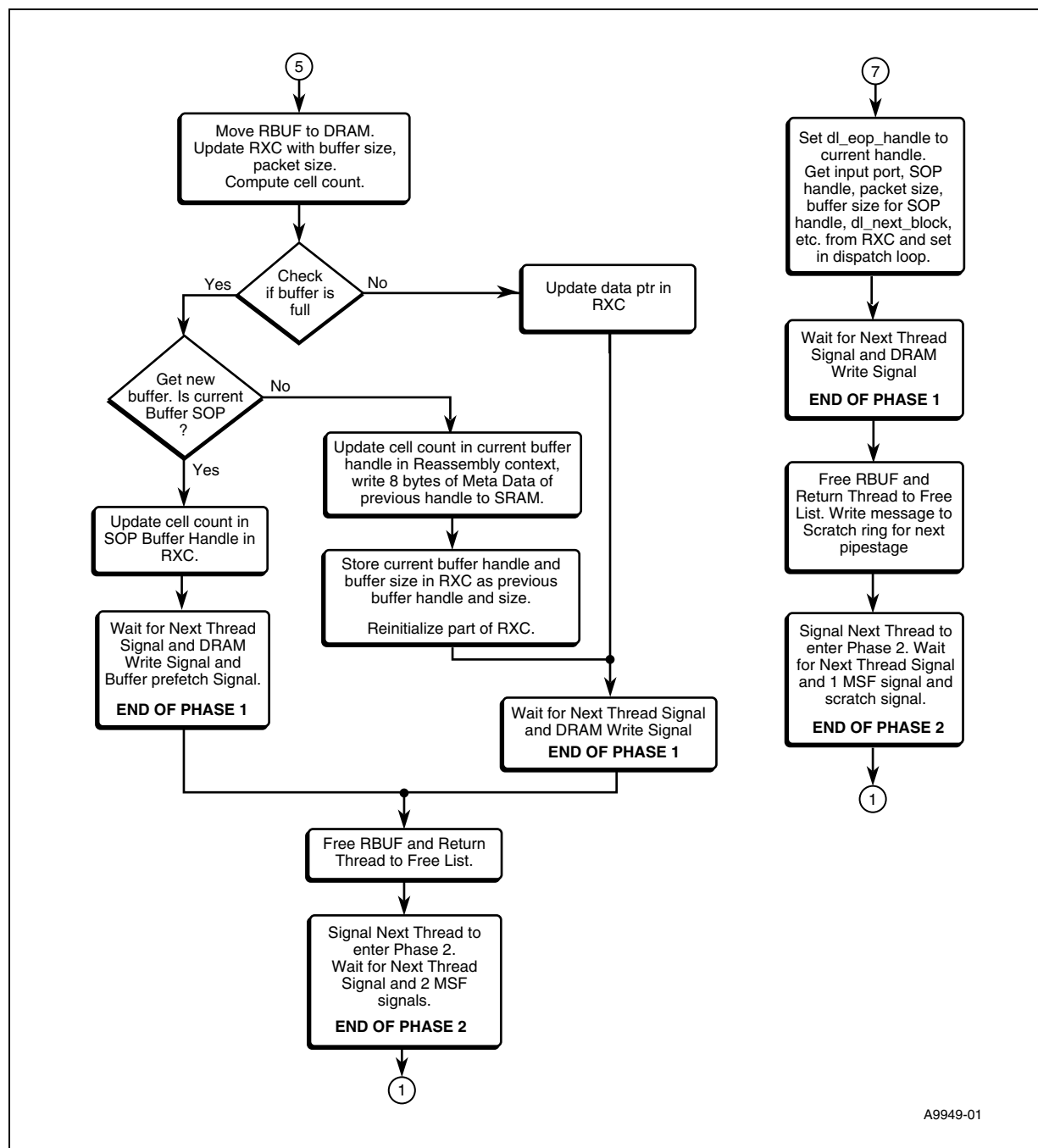


Figure 4-7. POS Receive Flowchart: Page 3 of 3



4.8 Performance Analysis

Assuming 128 byte RBUF elements, the worst case is for min size packets. Table 4-5 shows the performance of the block with respect to various data rates and media types.

Table 4-5. Performance Analysis of the Block

Item	600 MHz IXP2400 min POS	600 MHz IXP2400 4G min Ethernet	1.4 GHz IXP2800 min OC-192	1.4 GHz IXP2800 10G min Ethernet
Available cyclecount	97	101	57 (+ 57 for 2 microengine design)	94
Actual cycle count	81	86	94 for both microengines included	86

For example, the IXP2400 operates at 600 MHz. For a minimum POS packet of 49B, the packet inter-arrival time at OC-48 line rate is 97 ME cycles. In order to maintain line rate for minimum packets, each stage of the pipeline cannot exceed this budget. In other words, each stage of the pipeline needs to retire a packet every 97 cycles. Since the Packet RX microblock is a context pipe-stage it must handle a minimum POS packet in 97 cycles. Also since eight threads are executing in parallel, the I/O latency available to handle a minimum POS packet is $97 * 8$ cycles.

For a packet that spans two mpackets, the critical case is for a packet of size 136 bytes—assuming 128 byte RBUFs. In this case, assuming the IXF6048 strips seven bytes of the PPP header—two bytes for the address and control fields, four bytes for the FCS and one byte for the flag, the first RBUF contains 128 bytes and the next RBUF contains one byte of data. The available cycle count for min packet POS is $((136/49) * 97) / 2 = 134$ cycles per RBUF.

Note: A 256-byte RBUF size would give us extra cycles for the multiple mpacket case. However, the number of available RBUFs would be reduced in half. For a 64-byte RBUF size, a packet spanning two RBUF's (72 bytes) would get a cycle budget of only $((72/49) * 97) / 2 = 71$ cycles per RBUF. This is not enough to process an RBUF and therefore the 64-byte RBUF option is not used.

For Four-Gigabit Ethernet, assuming minimum Ethernet packet size of 64 bytes and inter-packet gap of 20 bytes, the packet inter-arrival time is 101 microengine cycles.

Table 4-6 summarize the performance analysis for the Packet RX block.

Table 4-6. Packet RX Block Performance Analysis: I/O Operations for min Packet

I/O Operations for min Packet	Phase
MSF read of four bytes	1
RBUF to DRAM xfer of 40 bytes	1
SRAM dequeue	1
MSF write to return RBUF to free list	2
MSF write to return thread to free list	2
Scratch write of four words	2

4.8.1 Characterization Data

Table 4-7. Packet RX Microblock Characterization Data

Data	Value
General:	
Microblock Name	Packet_Rx
Microblock Version Number	1.0
Implementation Language	microcode
Configuration Options use to gather this set of data	
Measurement Environment (tool settings)	N/A
Performance: (cycle counts, latencies, bandwidths)	
Instruction cycle count per packet (common case packet / worst case cycle count)	IXP2400 min POS 49B (81) IXP2400 min 4G Ether 64B (86) IXP2800 min OC-192 49B (94) IXP2800 min 10G Ether 64B (86)
Common-case packet/path assumptions to be documented here	N/A
Scratch Memory	
# of longwords read (for bandwidth calculations)	
# of longwords written (for bandwidth calculations)	4
# and type of each atomic operation performed (for bandwidth calculations)	
SRAM	
# of longwords read	1
# of longwords written	4
# and type of each atomic operation performed (for bandwidth calculations)	
DRAM	
# of quadwords read	
# of quadwords written	
Other I/O instructions (i.e. MSF, PCI, CAP, and Hash)	dramRbufRd (up to 16 LWs) msfFastWr (2 times)
List of dependent I/O accesses in the longest latency path	Data write (dram rbuf_rd <=16) Prefetch a buffer (sram deq 1) Write Meta data (sram wr 4) Free RBUF (msf fast wr) Return thread to free list (msf fast wr) dl_sink (scr wr 4)
Per-Microengine Resources:	

Table 4-7. Packet RX Microblock Characterization Data (Continued)

Data	Value
Control-Store Usage (# of instructions used)	700
Local Memory Footprint (# of long words used)	256 (+ 128 if STRIP_4BYTES_CRC is defined)
Local Memory Configuration (shared, or per-context pointer)	256 (+ 128 if STRIP_4BYTES_CRC is defined)
Local Memory - # of LM pointers used	1 (2 if STRIP_4BYTES_CRC is defined)
GPR Usage – minimum, static usage (absolutes, static, globals)	
Transfer Reg. Usage – minimum, static usage	
Next Neighbor Reg. Usage – minimum, static usage	NN ring for 2 ME Packet Rx design (used for OC-129 POS, 10G Ether)
Signal Usage – minimum, static usage	
CAM used? (yes or no)	No

Global Resources:

Scratch footprint (# of longwords used) – constant or formula ...	1024 (Scratch Ring to the next microblock)
SRAM footprint (# of longwords used) – constant or formula ...	N/A
DRAM footprint (# of quadwords used) – constant or formula ...	N/A
Q-Array usage - # of queues used and if they need to be cached	N/A
CRC Unit used?	N/A
Hash Unit used? (yes or no)	N/A

MSF Usage Information:

Media Bus Configuration	POS3 (1 ME Packet Rx) SPI4 (2 ME Packet Rx)
RBUF, TBUF usage	8 KB Total RBUF (128B RBUF)
CBus signals	

Other Information:

Critical Section Length (compute cycles + memory accesses)	
# of phases	2
Packet Metadata - fields read	
Packet Metadata - fields written	buffer_next, offset, bufferSize, hdr_type, rx_stat, free_list, packet_size, output_port, input_port
Header - fields read	N/A
Header - fields written	N/A

Table 4-7. Packet RX Microblock Characterization Data (Continued)

Data	Value
Documentation:	
Thread Ordering Requirements	Hyper Task Chaining
OS dependencies	N/A
Chip/hardware dependencies (i.e. Crypto Unit in 2850)	
Tested on which SDK Release(s)	IXA SDK 3.5
Tested on hardware? Which hardware configuration?	OC-48 POS, OC-192 POS, 4G Ether, 10G Ether
Tested in which applications (not an all inclusive list)	OC-48 POS, OC-192 POS, 4G Ether, 10G Ether
Possible Configuration Options	One ME Packet Rx (For OC-48 POS, 4Gig Ether) Two ME Packet Rx (For OC-192 POS, 10Gig Ether)
Major limitations (i.e. Rx/Tx microblocks written to use entire ME)	N/A
Packet Sequencing Issues (esp. in POT applications)	N/A
Core Component or Interface requirements or dependencies	N/A

5.1 Overview

The CSIX RX microblock receives c-frames from a CSIX fabric and reassembles them into packets. A key difference between the CSIX receive and the Packet receive microblock is that while the Packet RX block supports only 16 virtual ports, the CSIX RX block supports any number of VOQs. The ingress blade id and the QoS class are used to uniquely identify a VOQ.

Since the packets being reassembled may be up to 9k bytes, some large packets may be stored in multiple buffers chained together as a link-list. The buffer handles for the first and last packet in the chain are queued in the scratch ring. The CSIX RX microblock also sets up packet meta information (offset, size etc.) which are passed down the pipeline either in SRAM or in the scratch ring itself.

This microblock supports two configurations. It can run entirely on one microengine (8 threads) or it can be run on two microengines (16 threads). The single microengine design is used for the IXP2400 and for Gigabit Ethernet on the IXP2800. The two microengine design is used for supporting OC-192 POS data rates.

5.2 Assumptions and Dependencies

The following assumptions are made in the design and implementation of the CSIX RX block:

- The one microengine CSIX RX design supports an unlimited number of VOQ's. Each VOQ has a context that is maintained in SRAM, so the number of VOQ's is limited only by the size of SRAM available. Up to 16 contexts are cached in the microengine local memory using the folding technique. The two microengine CSIX RX design supports 256 VOQ's. Each VOQ has a context that is maintained in the microengines' local memory, limiting the number of VOQ's to 256.
- A VOQ is uniquely identified by the ingress blade id (in the per c-frame prepend header) and the class id (taken from either the extension header or the QoS class id field in the prepend header using a compile time constant). The one microengine CSIX RX design assumes 64 blades and 16 classes per blade (both are compile time constants and may be changed). The <blade id, class id> pair (10 bits; 6-bit blade id and 4-bit class id) is used as a key to access the VOQ context. The two microengine CSIX RX design assumes 16 blades and 16 classes per blade (both are compile time constants and may be changed). The <blade id, class id> pair (8 bits; 4-bit blade id and 4-bit class id) is used as a key to access the VOQ context. Both the ingress blade id and class cannot be 0 since the combination of the two is used as the cam lookup key.
- For every c-frame received, the CSIX RX block moves the packet data from the RBUF into DRAM. Subsequent stages in the pipeline read the header data from DRAM when needed. Once the CSIX RX block gets a signal that the data from the RBUF has been written into DRAM, frees up the RBUF element by putting it back on the element free list. We do not attempt to process the header directly from the RBUF.
- For a specific VOQ, The CSIX RX block needs to maintain the sequence in which packets arrive. Buffer handles for packets are queued into the scratch ring in the order in which they

arrive on the MSF media interface. This is achieved by making sure that the threads execute in order.

- The RBUF element size selected must be either 128 or 256 bytes. The 64-byte option is not used. This allows us more cycle budget to handle MOP and EOP only mpackets.
- For the SOP buffer, the CSIX RX block starts writing the packet data into DRAM at an offset at least equal to 128-bytes. The first 128-bytes are left empty to allow blocks down the pipeline to add/remove headers. For the non-SOP buffers, the packet data is written into DRAM at offset equal to 0. To improve the bank utilization of DRAM, the offset for the SOP buffer is rotated between either of 128, 256, 384 or 512; for more information, refer to [Section 5.4.2, “Optimizing the DRAM Bank Utilization” on page 98](#).

5.3 Basic Algorithm

We first describe the algorithm for the single microengine design.

The CSIX RX block has 4 phases, shown in [Table 5-1](#), and an initialization phase.

Initialization Phase: Each thread initializes certain thread local variables and adds itself to the MSF thread free list in strict order. In addition thread 0 does some global configuration by initializing local memory, the CAM, the SRAM contexts and the RBUF free list. It also configures the MSF to work in CSIX mode by setting the *MSF_rx_control* register. The next thread signals protecting phase 2, 3 and 4 are pre-sent to thread 0 to get the pipeline running for the first time.

Phase 1: The MSF auto pushes status and wakes up a thread when a c-frame arrives. The thread checks the Receive Status Word (RSW) for errors. If there are no errors, it issues a read of the RBUF prepend header to get the ingress blade id. It then waits for the read complete signal and the signal from the previous thread to enter phase 2.

Phase 2: This phase implements the folding algorithm. The entry to this phase is ordered (order is ensured by the inter-thread signal) since this is a critical section. The blade id and class id (from the CSIX extension header) are used to compute a 10-bit key, which is used to do a CAM lookup for the RXC. If the CAM lookup is a hit, the thread signals the next thread to enter phase 2, proceeds to the start of phase 3 and waits for the previous thread signal. In the case of a CAM miss, the thread evicts the LRU (Least Recently Used) RXC from local memory by writing it to SRAM and issues a read from SRAM of the new context. The thread then signals the next thread to enter phase 2 and proceeds to the start of phase 3 where it waits for the I/O operations to finish and a signal from the previous thread.

Phase 3: In the case of a CAM miss, the new context is moved from transfer registers to local memory. In the case of a CAM hit, it is assumed that the context has already been moved to local memory by a previous thread. The cframe is processed and the context is updated if necessary with information from the prepend header. The cell sequence number from the prepend header of the received cframe is checked against the expected cell sequence number in the RXC. If it does not match the cframe is dropped. The partially received packet is also dropped. If the cell sequence number matches the expected sequence number in the RXC, the cframe data is moved from RBUF to DRAM. If an EOP is detected, then the packet meta data is written to SRAM. The thread then signals the next thread to enter phase 3 and proceeds to the start of phase 4 and waits for the I/O operations to finish and for a signal from the previous thread.

Phase 4: In this phase, the RBUF element is freed. If an EOP is detected, then the packet handle is sent to the sink block in the dispatch loop for queuing to the next stage in the pipeline via a scratch ring. The thread then signals the next thread to enter phase 4 and puts itself back on the thread free list. [Table 5-1](#) illustrates the basic algorithm for the single microengine design.

Table 5-1. CSIX Receive Algorithm for a Single Microengine Design

<p>Phase 1</p> <ol style="list-style-type: none"> 1. MSF read of the prepend header. 2. Wait (MSF read, previous thread signal) <p>Phase 2</p> <ol style="list-style-type: none"> 1. Signal next thread. 2. If (SOP/EOP) { RBUF --> DRAM flush meta data } 3. CAM lookup 4. If (miss) { evict LRU RXC by writing to SRAM Read in new RXC } 5. Wait (SRAM RXC read complete, SRAM RXC write complete, previous thread signal) <p>Phase 3</p> <ol style="list-style-type: none"> 1. Signal next thread. 2. If (CAM miss) { move RXC to local memory } 3. If (Not SOP/EOP)) { RUF --> DRAM if (EOP) flush meta } 4. Wait for DRAM write to complete, previous thread signal, and SRAM write complete if EOP. <p>Phase 4</p> <ol style="list-style-type: none"> 1. Signal next thread. 2. Free RBUF element. 3. If EOP, sink block writes to scratch ring. 4. Put thread on free list. 5. Wait (receive event, scratch I/O done)
--

5.4 Optimizations to the Basic Algorithm

5.4.1 Optimizations for the Single C-Frame Packet

This section describes optimizations to the overall algorithm for the case where the entire packet fits into a single c-frame.

This case is detected when both the SOP and the EOP bits are set in the cframe prepend header. For this case, the RXC is not required to initiate the RBUF to DRAM write and the SRAM write of the packet meta data. Therefore these I/O operations are moved up from phase 3 to phase 2. This gives more time for them to finish since the thread stills waits for these I/O operations at the end of phase 3.

The RXC is still looked up and obtained to drop any other packet that was currently being reassembled. However apart from the expected sequence number field, the rest of the RXC is not updated with any of the information from the prepend header. Therefore only 3 of the 8 long words of the RXC are read in from SRAM.

5.4.2 Optimizing the DRAM Bank Utilization

The IXP2400 has a single DRAM channel with four banks. The DRAM memory banks are interleaved to improve concurrency and bandwidth utilization. Bits 7..8 of the command address are used as the bank select bits. This means that every 128 bytes of DRAM falls into a different bank.

In order to spread the memory accesses uniformly across banks, the following scheme is used. For the first cframe of every packet, the starting offset into the DRAM buffer at which the packet data is written is rotated among four values—128, 256, 384 and 512. This implies that for a stream of minimum size packets, the first packet is written into a buffer at offset 128, the second packet into another buffer at offset 256 and so on.

5.5 Extending the Design to Run on Two Microengines

This section describes how the code was modified to run on two microengines. This is essential to meet the cycle budget for OC-192 data rates.

The CSIX RX code as a compile time option can be executed on two microengines (16 threads). Each microengine does a portion of the work. The work is split between the two microengines in a manner that allows them to execute simultaneously.

The algorithm used to execute CSIX RX on two microengines is shown in [Table 5-2](#).

Table 5-2. CSIX RX Algorithm Running on Two Microengines

First Microengine

Phase 1

1. Wakeup when a CSIX cell arrives in an RBUF element.
2. Read CSIX cell prepend header from RBUF.

Phase 2

1. Prefetch a buffer to store the cell payload in DRAM.
2. Write some buffer meta-data fields in SRAM.
3. For non-min packet cells, update cell context information in local memory.
4. Send message on NN ring to second microengine to continue processing.
5. Wait for next cell arrival signal.

Second Microengine

Phase 1

1. Wait for an NN ring message from first microengine.
2. Read message from NN ring.
3. Write cell payload to DRAM.
4. For non-min packet cells, update cell context information in local memory.
5. Write remaining meta-data fields in SRAM.

Phase 2

1. Free the RBUF element.
2. Write message on scratch ring for next microblock.

5.6 Data Structures

5.6.1 Receive Reassembly Context (RXC) for One Microengine Design

Table 5-3 specifies the Reassembly Context (RXC). Each RXC is 8 words or 32 bytes. Therefore the total SRAM required to support 1024 contexts is 32k bytes. 16 of these contexts are cached in local memory. Therefore this block utilizes $32 * 16 = 512$ bytes or 128 words of local memory.

Table 5-3. CSIX Receive Reassembly Context for one Microengine

LW	Bits	Size	Field	Description
0	31:16	16	packet_size	Size of the packet
0	15:0	16	cell_seq_number	Expected cell sequence number
1	31:0	32	sop_buffer_handle	Current DRAM pointer for current buffer
2	31:0	32	current_buffer_handle	Size of the current buffer in bytes
3	31:0	32	current_data_handle	Buffer Handle for EOP buffer descriptor
4	31:16	16	current_buffer_size	Size of the current buffer
4	15:0	16	sop_buffer_offset	SOP buffer offset
5	31:24	8	reserved	Reserved
5	23:16	8	header_type	Header type
5	15:0	16	output_port	Output port
6	31:0	32	flow_id	Flow id
7	31:24	8	remainder	Used to compute cell count
7	23:0	23	prv_buf_handle	Previous buffer handle

5.6.2 Receive Reassembly Context (RXC) for Two Microengine Design

Table 5-4 and Table 5-5 specify the Reassembly Contexts (RXC). In the first microengine, each RXC is 8 bytes. In the second microengine, each RXC is 8 bytes. In both microengines, the entire RXC is stored in local memory. For 256 VOQ's this requires $256 * 8 = 2048$ bytes of local memory per microengine.

Table 5-4. CSIX Receive Reassembly Context for First Microengine

LW	Bits	Size	Field	Description
0	31:16	16	sop_buffer_offset	SOP buffer offset in bytes
0	15:0	16	packet_size	Size of the packet in bytes
1	31:16	16	current_buffer_offset	Buffer offset of current buffer in bytes
1	15:8	8	header_type	Header type
1	7:0	8	cell_seq_number	Expected cell sequence number

Table 5-5. CSIX Receive Reassembly Context for Second Microengine

LW	Bits	Size	Field	Description
1	31:0	32	sop_buffer_handle	Current DRAM pointer for current buffer
2	31:0	32	current_buffer_handle	Size of the current buffer in bytes

5.6.3 Lookup Key

The lookup key is a 32-bit value composed as shown in [Table 5-6](#) and [Table 5-7](#).

Table 5-6. CSIX Receive Lookup Key for One Microengine Design

Bits	Contents
31:10	All zeros
9:4	Blade number
3:0	Class number

Table 5-7. CSIX Receive Lookup Key for Two Microengine Design

Bits	Contents
31:10	All zeros
7:4	Blade number
3:0	Class number

5.6.4 Statistics

The CSIX RX block maintains the following statistics on a per VOQ basis in debug mode only. The microblock maintains these as 32-bit counters. The XScale core component keeps a 64-bit version of these counters; for more information, refer to [Section 2.5, “Statistics and Handling of 64-bit Counters”](#) on page 63.

Table 5-8. CSIX Receive: Statistics Counter Offsets from Base for VOQ

Counter	Offset
Packets received	0x0
Bytes received	0x4
Cframes received	0x8
Packets dropped	0xc

5.7 Build Switches

Table 5-9 describes the compile time options used in the CSIX Receive microblock.

Table 5-9. Compile Time Options Used in the One Microengine CSIX Receive Microblock

Symbol	Description
VERTICAL_PARITY_CHECK	Turn on vertical parity checking
CHECK_ZERO_BUFFER_HANDLE	Turn check for invalid buffer handles
CELL_MODE	Turn on for use of this microblock with a cell mode transmit.
COUNTERS	Turn on counters

Table 5-10. Compile Time Options Used in the Two Microengine CSIX Receive Microblock

Symbol	Description
MULTICAST_SUPPORT_ENABLE	Enables additional processing required for multicast applications.
COUNTERS	Turn on counters.

5.8 Performance Analysis

Tables 5-11, 5-12, and 5-13 summarize the performance analysis for the worst case: a 56-byte cframe containing a minimum size POS packet for which a CAM miss occurs.

Table 5-11 shows the budget and actual cycle count for the single microengine design for different configurations.

Table 5-11. CSIX RX—Budget and Cycle Count for One Microengine Design

Description	Instruction Cycle Count
Worst Case Cycle count	89
600 MHz IXP2400 budget for POS min packet at OC-48 rates	97
600 MHz IXP2400 budget for Ethernet min packet at 4 Gbps rates	101
1.4 GHz IXP2800 budget for Ethernet min packet at 10 Gbps rates	94

Table 5-12 shows the budget and actual cycle count for the two microengine design used in the OC-192 POS configuration.

Table 5-12. CSIX RX—Budget and Cycle Count for Two Microengine Design

Description	Instruction Cycle Count
Worst Case Cycle count	110
1.4 GHz IXP2800 budget for OC-192 POS min packet	$57 \times 2 = 114$

Table 5-13 shows the I/O operations across the different phases.

Table 5-13. CSIX RX Performance Analysis—I/O Operations for Min Packet Worst Case

I/O Operations for min packet Worst Case	Phase (Single ME Design)	Phase (2 ME Design)
MSF read of 16 bytes	1	ME 1 Phase 1
RBUF to DRAM xfer of 40 bytes	2	ME 2 Phase 1
SRAM dequeue to get next buffer handle	2	ME 1 Phase 2
Write to SRAM of meta data of 24 bytes	2	ME 1 Phase 2 ME 2 Phase 1
Write to SRAM of LRU RXC (28 bytes)	2	N/A
Read from SRAM of new RXC (12 bytes)	2	N/A
Write current TXC to SRAM	N/A	N/A
Dropping partially assembled buffer (1 SRAM enqueue or 1 scratch write of 3 words)	4	ME 2 Phase 1
MSF fast write to return RBUF to free list	4	ME 2 Phase 2
MSF fast write to return thread to free list	4	ME 1 Phase 1
Scratch write of 3 words	4	ME 2 Phase 2

5.8.1 Characterization Data

Table 5-14. CSIX Rx Microblock Characterization Data

Data	Value
General:	
Microblock Name	csix rx
Microblock Version Number	1.0
Implementation Language	microcode
Configuration Options use to gather this set of data	1. USE_IMPORT_VAR 2. VERTICAL_PARITY_CHECK 3. COUNTERS 4. CELL_MODE 5. IS_IXPTYPE(__IXP28XX) 6. CHECK_ZERO_BUFFER_HANDLE 7. USE_EXT_CLASS_ID 8. MULTICAST_SUPPORT_ENABLE 9. TWO_ME_CSIX_RX 10. CSIX_RX_FIRST_MICROENGINE
Measurement Environment (tool settings)	SDK 3.5
Performance: (cycle counts, latencies, bandwidths)	
Instruction cycle count per packet (common case packet / worst case cycle count)	One microengine: 89 Two microengines: 110

Table 5-14. CSIX Rx Microblock Characterization Data (Continued)

Data	Value
Common-case packet/path assumptions to be documented here	
Scratch Memory	
# of longwords read (for bandwidth calculations)	0
# of longwords written (for bandwidth calculations)	3
# and type of each atomic operation performed (for bandwidth calculations)	0
SRAM	
# of longwords read	One microengine: 7, Two microengines: 0
# of longwords written	One microengine: 13, Two microengines: 6
# and type of each atomic operation performed (for bandwidth calculations)	0
DRAM	
# of quadwords read	5
# of quadwords written	0
Other I/O instructions (i.e. MSF, PCI, CAP, and Hash)	msf[read] of 16 bytes, 2 msf fast wr, 1 sram[deq]
List of dependent I/O accesses in the longest latency path	

Per-Microengine Resources:	
Control-Store Usage (# of instructions used)	One microengine: 1019, Two microengines: 828
Local Memory Footprint (# of long words used)	64
Local Memory Configuration (shared, or per-context pointer)	per-context
Local Memory - # of LM pointers used	1
GPR Usage – minimum, static usage (absolute, static, globals)	
Transfer Reg. Usage – minimum, static usage	
Next Neighbor Reg. Usage – minimum, static usage	
Signal Usage – minimum, static usage	
CAM used? (yes or no)	Yes

Global Resources:	
Scratch footprint (# of longwords used) – constant or formula ...	0
SRAM footprint (# of longwords used) – constant or formula ...	One microengine: CSIX_RX_CTX_TOTAL*CSIX_RX_CTX_SIZE
DRAM footprint (# of quadwords used) – constant or formula ...	None
Q-Array usage - # of queues used and if they need to be cached	None

Table 5-14. CSIX Rx Microblock Characterization Data (Continued)

Data	Value
CRC Unit used?	no
Hash Unit used? (yes or no)	no
MSF Usage Information:	
Media Bus Configuration	N/A
RBUF, TBUF usage	128-byte RBUF element
CBus signals	
Other Information:	
Critical Section Length (compute cycles + memory accesses)	
# of phases	One microengine: 4, Two microengines: 2, 2
Packet Metadata - fields read	
Packet Metadata - fields written	
Header - fields read	
Header - fields written	
Documentation:	
Thread Ordering Requirements	Yes
OS dependencies	VxWorks, Linux
Chip/hardware dependencies (i.e. Crypto Unit in 2850)	IXP2400, IXP2800
Tested on which SDK Release(s)	IXA SDK 3.5
Tested on hardware? Which hardware configuration?	POS, ETH, ATM
Tested in which applications (not an all inclusive list)	Almost all Egress apps, e.g. OC48 POS, OC192 POS
Possible Configuration Options	
Major limitations (i.e. Rx/Tx microblocks written to use entire ME)	
Packet Sequencing Issues (esp. in POT applications)	Packets received must be in sequence
Core Component or Interface requirements or dependencies	

6.1 Overview

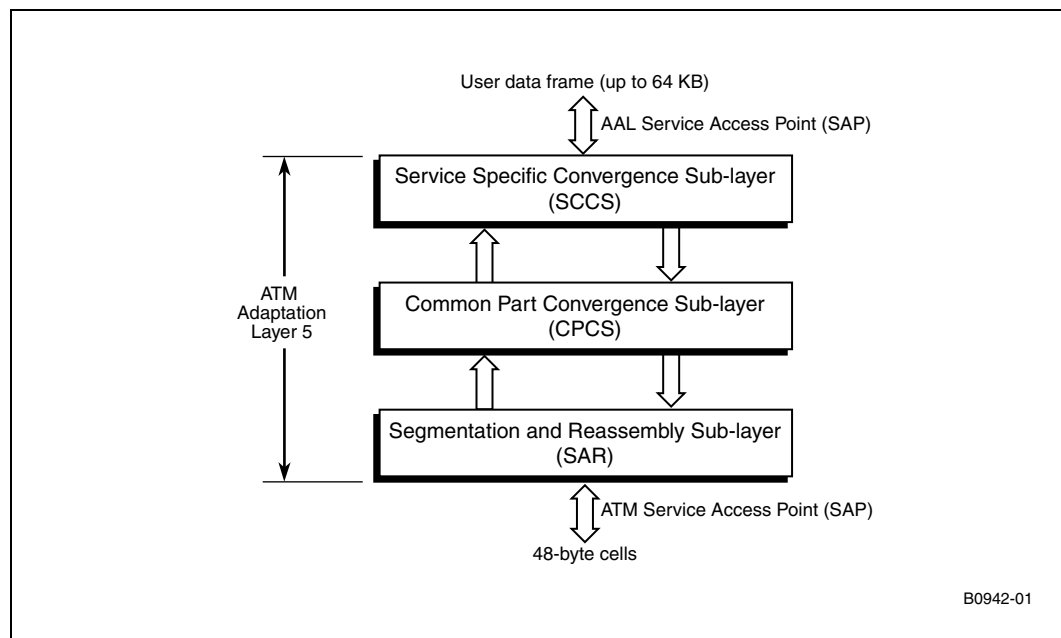
The **ATM Adaptation Layer (AAL) 5** Receive (RX) microblock performs ATM layer, AAL5-Segmentation and Reassembly Sublayer (SAR) and AAL5-Common Part Convergence Sublayer (CPCS) functions. The AAL5-Service Specific Convergence Sublayer (SCCS) layer is unused.

6.1.1 ATM Terminology

This section introduces common ATM terminology. Readers familiar with ATM terminology may skip this section with no loss of continuity.

ATM Adaptation Layer (AAL) 5 has 3 sub-layers—Service Specific Convergence Sublayer (SCCS), Common Part Convergence Sublayer (CPCS) and the Segmentation and Reassembly Sublayer (SAR). [Figure 6-1](#) shows the structure of these sublayers.

Figure 6-1. Structure of Sub-layers in AAL5



The SAR layer receives a stream of 48-byte cells from the media, reassembles them into AAL5 frames and passes them to the CPCS sub-layer.

The CPCS layer receives frames from the SAR layer and strips off any frame headers and the AAL5 trailer.

SCCS receives IP packet from CPCS (up to 64KB in length) and performs service specific functions. This layer is often unused.

Service Data Unit (SDU)—This is data as it appears at the interface between a particular sub-layer and the sub-layer immediately below. For example, data coming into the CPCS sub-layer is referred to as CPCS-SDU.

Protocol Data Unit (PDU)—This is data as it appears at the interface between a particular sub-layer and the sub-layer immediately above. For example data going out of the CPCS sub-layer is referred to as CPCS-PDU. CPCS-PDU becomes SCCS-SDU at the SCCS sub-layer.

For more generic definitions refer to [I.363.5] and [I.361]

6.1.2 ATM Layer Functions

The AAL5 RX microblock receives a 52-byte ATM PDU from the ATM framer. The framer strips off the 1-byte HEC from ATM header. The ATM layer in the RX microblock strips off the 4-byte cell header from each ATM PDU (52-byte) received to get a 48-byte ATM SDU, which is sent to the SAR layer.

6.1.3 AAL5-SAR Functions

The AAL5 RX microblock implements the AAL5 SAR layer and reassembles all 48-byte SAR PDUs for one AAL5 frame into a SAR layer SDU with maximum size of up to 655,35 bytes (including padding and a 8-byte trailer). The last SAR PDU of the frame has the ATM user-to-user (AUU) bit set in its cell header, which is used to separate one frame from another.

6.1.4 CPCS Functions

The AAL5 RX microblock also implements the CPCS layer and strips off the 8-byte trailer and padding (if any) to form a CPCS SDU (that is, a user data frame). This frame is passed to the packet-processing pipeline for further handling.

6.1.5 Design Overview

This section describes the design for an ATM Receive microblock for OC-48 SPHY 1x32 UTOPIA mode. The design for OC-48 MPHY-4 and SPHY 4x8 is virtually identical except that the threads are divided into four per port. Since only four threads are used, no inter-ME signaling is required and some optimization is possible.

The ATM RX microblock supports up to 64K virtual circuits. For each of the 64K VCs, there is an associated Reassembly Context (RXC) of 9 long words stored in SRAM, which helps in the frame reassembly task spread across multiple threads. Local memory is used as a cache for these reassembly contexts and the CAM is used for lookups. The threads execute in strict order to maintain the order of ATM cells received from the media interface. Since AAL-5 frames may be up to 64KB, some large packets may be stored in multiple buffers chained together as a link-list. The buffer handles for the first and last buffer of the packet, along with other meta data information are passed to the next microblock in the functional pipeline via the scratch ring.

For each ATM cell received from the framer, the MSF auto-pushes a Receive Status Word (RSW) to the transfer registers of a microengine thread. The thread does a hash lookup based on the VPI, VCI and input port of the received cell to get the location in SRAM of the corresponding

Reassembly Context (RXC) and VC based statistic counters. (The hash lookup may be compiled out at build time and replaced with a simple index lookup using the 16-bit VCI.) The block then reads the RXC from SRAM and also copies the entire ATM cell into the microengine registers.

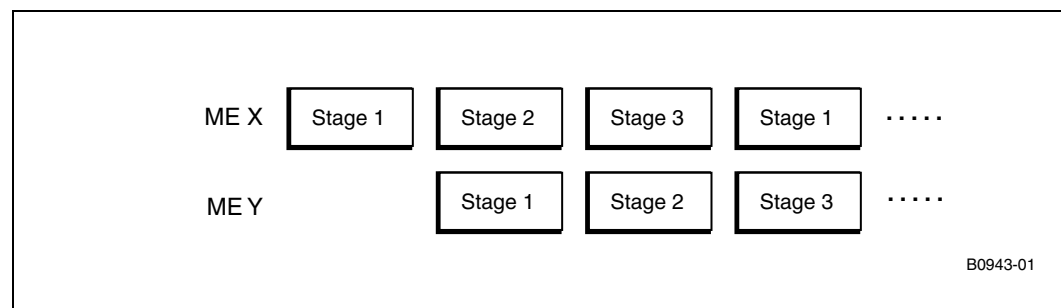
The ATM header is stripped off and CRC-32 is computed over the remaining 48 bytes. The payload of the cell is written to DRAM and the RXC is updated accordingly. On EOP, the CRC is validated against the CRC in the trailer of the AAL5 frame. The buffer handles for the first and last buffer of the packet, along with other meta data information are queued on to the scratch ring to be handled by the next stage of the packet processing pipeline.

6.1.5.1 OC-48 (SPHY_1_32)

We use the UTOPIA protocol in SPHY_1_32 mode. Two microengines running in a three stage functional pipeline are used to achieve OC-48 line rate.

This requires the assembler switch of TWO_ME_RX_FP.

Figure 6-2. Three Stage Functional Pipeline for Two Microengine AAL5 RX Design



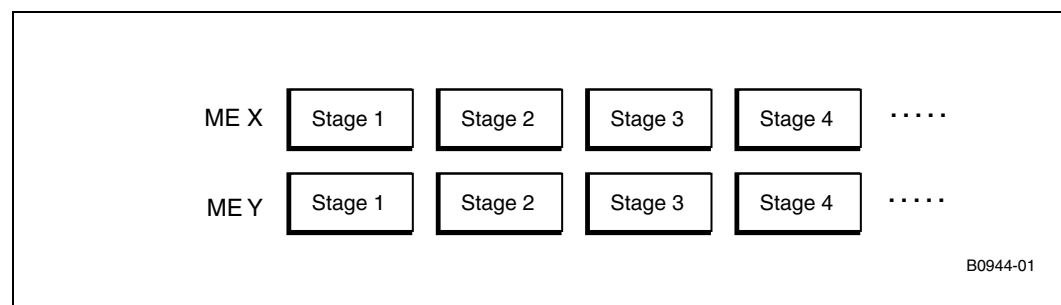
6.1.5.2 Quad OC-12 (SPHY_4_8) or OC-24/OC-12 (SPHY_1_32)

In this configuration, two micro-engines running independently (set of 4 threads per single OC-12 port) are used to achieve Quad OC-12 performance in SPHY_4_8 mode.

A single micro-engine can be used to achieve OC-24 or OC-12 rates in SPHY_1_32 mode.

Both these configurations require the assembler switch of ONE_ME_RX_FP.

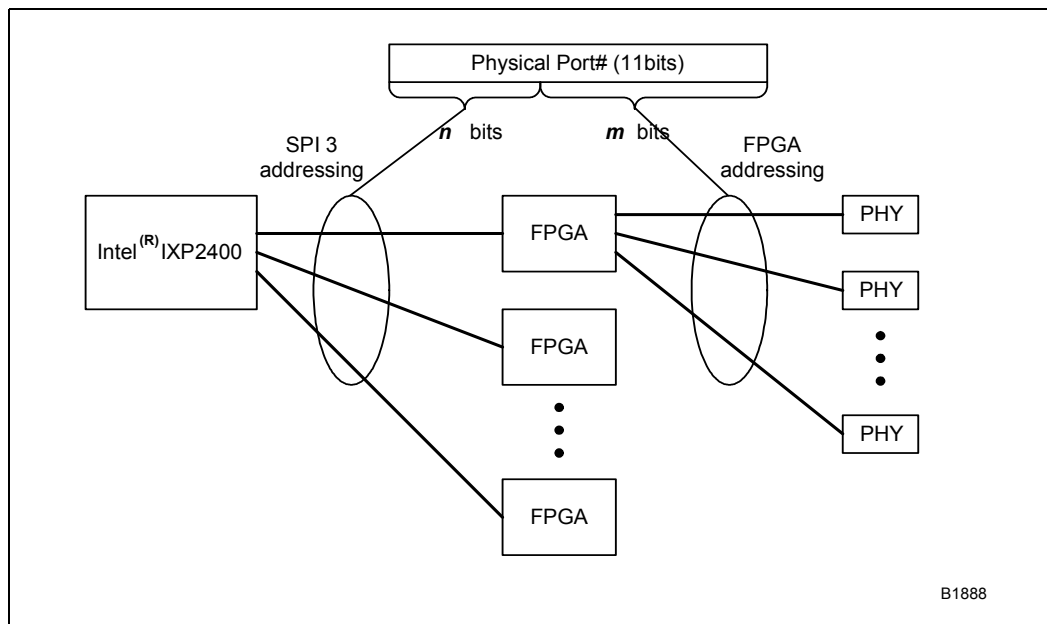
Figure 6-3. Four Stages Running on Two Microengines Independently



6.1.5.3 A Maximum of 2048 Ports Addressing

Figure 6-4 illustrates the addressing scheme used to support a maximum of 2048 ports.

Figure 6-4. A Maximum of 2048 Ports Addressing Scheme



The n most significant bits from the port number are used for device addressing on the SPI-3 bus. The number of these bits depends on the number of output ports defined in an implementation.

The m least significant bits are used for the address of the output PHY connected to the FPGA device. Table 6-1 contains an example of split port number bits.

Table 6-1. An Example of Split Port Number Bits

Number of SPI-3 ports	Number of bits used for SPI-3 addressing - n	Number of bits used for FPGA addressing - m	Number of FPGA ports
1	0	11	2048
4	2	9	512

The port number split (SPI-3 and FPGA addressing) is defined at compile time.

6.2 Configuration/Switches

Table 6-2 shows the AAL5 RX microblock supported pre-processor switches.:

Table 6-2. AAL5 RX Microblock Pre-Processor Switches

Switch	Description
RX_PHY_MODE (in dl_system.h)	Set this switch to SPHY_1_32 when running the microblock in SPHY 1x32 mode. In this mode, we need 2 microengines running AAL5 RX as a functional pipeline to achieve OC-48 line rate or 1 micro-engine running AAL5 RX to achieve OC-24/OC-12 line rates. Set this switch to SPHY_4_8 when running the microblock in SPHY 4x8 mode. When running in this mode, we need 2 microengines running AAL5 RX (4 threads per OC-12 port) to achieve Quad OC-12 line rates.
TWO_ME_RX_FP	Set this switch to have AAL5 RX running on two microengines as a functional pipeline to support SPHY_1_32 (OC-48 rates).
ONE_ME_RX_FP	Set this switch to have AAL5 RX running on 4 threads per port to support one or more OC-12 ports.
SPHY_4_8_VC_INIT	Use this switch to initialize VCs and hash tables for SPHY_4_8 (Quad OC-12) in simulation.
HASH_LOOKUP	Set this switch to perform hash based lookup to get the VC Info (vc_id)
SIMPLE_LOOKUP	Set this switch to do skip the hash lookup to get VC Info (vc_id). In this case a direct mapping is assumed between VCI and vc_id's. This requires that the hash tables and RXC tables be initialized accordingly.
FIRST_AAL5_RX_ME	Use this switch to identify the first ME to start processing or the 1st of 2 ME is used as a context pipeline: <ul style="list-style-type: none"> When RX_PHY_MODE is equal to SPHY_1_32, this switch identifies the first ME to start processing. When RX_PHY_MODE is equal to SPHY_4_8, this switch identifies the 1st of 2 ME's used as a context pipeline.
NEXT_AAL5_RX_ME (used only with TWO_ME_RX_FP):	Used this for inter-me signaling—identifies the next ME in the functional pipeline relative to this ME.
AAL5_RX_COUNTERS:	Define to enable counters (as explained earlier).
FPGA_HEADER	Define the switch to support a maximum of 2048 ports.
NON_AAL5_TX, CSIX_TX	This says it has TX block other than AAL5_TX. And the TX is CSIX TX. This is used to count the number of c-frames in a buffer. This is required to queue the packet for transmitting.

6.3 Assumptions, Dependencies and Risks

The following assumptions are made in the design and implementation of the AAL5 RX block:

- The design assumes that it receives cells belonging to an AAL5 frame in order from the MSF.
- The AAL5 RX block has no way of detecting if cells were dropped by the MSF. If a cell is dropped, this error is detected only by the CRC and packet length check on the AAL5 frame in which case the entire frame is dropped.
- The AAL5 RX block supports up to 64K virtual circuits. This implies that the AAL5 RX block has to store VC information (includes RXC and VC statistic counters) in SRAM and local memory may only be used as a cache. The current design uses the folding algorithm to optimize the read of the RXC from SRAM.
- The current design supports both hash lookup and simple lookup (using 16-bit VCI) to find the pointer to the VC information in SRAM. The user is given a build option to select at compile time either the simple lookup or the hash lookup.
- For every ATM cell received, the AAL5 RX block reads the cell from the RBUF into transfer registers, processes the cell and then writes the payload into DRAM. Subsequent stages in the pipeline reads the packet data from DRAM as needed. The AAL5 RX block does not process either the LLC SNAP or the IP header.
- The AAL5 RX block assumes that it receives a 52-byte cell from the framer, that is, it assumes that the framer strips off the HEC. This implies that no byte alignment is required to process the ATM cell.
- The RBUF element size used is 64 bytes, but the design would work for a 128-byte size as well.
- The AAL5 RX block assumes code running on the XScale core sets up the appropriate fields in the RXC.
- For the first buffer of every packet, the AAL5 RX block leaves some headroom to allow the blocks down the pipeline to add or remove headers.

Note: These bytes are not included in the computation of the cell count. To optimize dram bank scheduling this headroom is rotated across packets to be 128, 256, 384 or 512 bytes. For the rest of the buffers in the packet, the packet data is written into DRAM at offset equal to 0.

- The AAL5 RX block computes and sets the cell count (the `seg_count` field) for every buffer depending on the transmit media. For CSIX transmit, the size of the cell would be equal to c-frame size after taking into account the per-packet and per c-frame prepend headers. If the transmit media is POS or Ethernet, then the cell count field is not used.
- The base address in SRAM for the VC information (`AAL5_RX_VC_INFO_SRAM_BASE`) should not be zero, as we are using the absolute address of the RXC as a tag to access the CAM and zero is not a valid tag.

6.4 Data Structures

6.4.1 Reassembly Context (RXC) or VC Info table

The ATM RX block needs to reassemble ATM cells (for the same VC) into AAL5 frames. The Reassembly Context (RXC) is the data structure used to maintain state information pertaining to the reassembly process. Each VC (virtual circuit uniquely identified by VPI, VCI and input port number) has an associated RXC, which is reset at the end of processing a complete AAL5 frame. The RXC information is stored in SRAM and local memory is used as a cache. The CAM is used to manage tags for this cache and the folding algorithm is used to optimize the read of the RXC information.

The current design supports up to 64K VCs, hence there may be up to 64K RXC structures, each with nine long words. To minimize the size of the critical section for the two microengine design, the RXC is split into two parts - RXC1 and RXC2. RXC1 (LW0 and LW1) is read, modified and written by pipestage 1 of the functional pipeline, while RXC2 (LW2 to LW8) is read, modified and written by pipestage 2 of the pipeline. Each microengine flushes its RXC back to SRAM before signaling the next microengine to start that stage. This maximizes parallel execution on the two microengines without loss of data integrity.

For a single microengine design, there is no need to divide the RXC context because only one microengine can modify the RXC at one time. [Table 6-3](#) specifies the RXC data structure.

Table 6-3. RXC Data Structure

LW	Bit	Len	Name	Description
0	31:31	1	exp_SOP	When EOP is hit, exp_SOP bit is set for next thread to know that it has a SOP cell. It is initialized to 1 by control plane and then is set/reset by ATM RX.
	30:30	1	vc_allocated	Whether this VC is unallocated/allocated (allocated = 0). This bit is used by control plane to delete the VC entry from the hash table, if it is unallocated. It is set by control plane for its job of maintaining tables, and is not used by ATM RX.
	29:24	6	Reserved	Reserved
	23:23	1	vc_state	State of VC (valid(0) or not valid(1). It is set by control plane and is used by ATM RX.
	22:20	3	aal_type	AAL Type supported by this VC000-AAL5, 001-AAL3/4, 010-AAL4, 011-AAL3, 100-AAL2. It is set by control plane and used by ATM RX.
	19:16	4	free_list_id	Freelist ID from where the buffer handle would be fetched (used only for non AAL5, ATM RX is setting BD:Freelist ID to 0 for AAL5). It is set by control plane and used by ATM RX.
	15:0	16	Reserved	Reserved
1	31:0	32	crc_residue	Cumulative CRC of the current packet. It is initialized to 0xffff ffff by control plane and ATM RX then updates it.

Table 6-3. RXC Data Structure (Continued)

LW	Bit	Len	Name	Description
2	31:31	1	Eop	EOP bit for the current buffer
	30:30	1	Sop	SOP bit for the current buffer
	29:24	6	seg_cnt	Seg_cnt for the current buffer
	23:0	24	ptr_to_curr_buf	pointer to the current buffer
3	31:31	1	Eop	EOP bit for the SOP buffer
	30:30	1	Sop	SOP bit for the SOP buffer
	29:24	6	seg_cnt	Seg_cnt for the SOP buffer
	23:0	24	ptr_to_SOP_buf	pointer to the SOP buffer
4	31:16	16	Sop_buff_size	Size of SOP buffer
	15:0	16	Offset	offset of the start of data in the buffer in bytes (for SOP Buffer)
	31:16	16	buf_offset	Current DRAM offset into the current buffer, at which the received cell would be stored.
5	15:0	16	curr_pkt_size	number of bytes, received till now for the current packet
6	31:31	1	Drop_bit	Bit set to discard the packet assembled till now
	30:30	1	Exception_bit	Indicates that frames received on this VC are used for signaling. It is set by control plane and is used by ATM RX.
	29:28	2	Reserved	Reserved
	27:24	4	Header_type	Type of header carried by the frame on this VC. It is set by control plane and is used by ATM RX.
	23:0	24	Prev_prev_buf_add	Address of previous buffer to the previous buffer
7	31:16	16	Prev_buf_size	Size of previous buffer
	15:8	8	Bytes_remaining	Number of bytes in the current c-frame. It is used to keep track of the number of c-frames in the current buffer.
	7:0	8	Cell_cnt	Number of c-frames in the current buffer.
8	31:0	32	Prev_buf_handle	Pointer to the previous buffer
9	31:0	32	Optional_info	A value assigned by the service user that has a meaning to that user only. The value is passed with every packet received on the VC and should be used to eliminate a need for a lookup in the next (after the ATM RX) block.

Table 6-4 shows the valid combinations.

Table 6-4. Valid Combinations

VC_alloc	VC_valid	
0	0	VC is valid, allocated

Table 6-4. Valid Combinations

VC_alloc	VC_valid	
0	1	VC is invalid temporarily
1	0	Invalid combination
1	1	VC can be removed from hash and VC Information tables

6.4.2 AAL5 RX Counters

Tables 6-5 and 6-6 shows the port based and VC based counters currently supported. Each counter is 32-bit and is stored in a specific location in SRAM. The XScale core reads the 32-bit counter periodically and may maintain a 64-bit version of it.

Table 6-5. Port Based Counters

AAL5_RX_PKT_COUNT	Counts the number of valid packets received by AAL5 RX
AAL5_RX_PKT_DROP_COUNT	Counts the number of packets dropped by AAL5 RX (bad_CRC, bad_Len, null_buffer)
AAL5_RX_HW_ERROR	Counts the number of hardware RX errors (Cell_size error, RSW_Par_Err)
AAL5_RX_VC_ERROR	Counts the number of hash_miss encountered during VC_lookup

Table 6-6. VC Based Counters

AAL5_RX_CELL_CNT	Counts the number of valid cells received by AAL5 RX (AAL5 cells, signalling cells)
AAL5_RX_MGT_CELL_COUNT	Counts the number of mgt cells received (RM and OAM cells) by AAL5 RX
AAL5_RX_AAL5_PDU_COUNT	Counts the number of valid AAL5 frames reassembled by AAL5 RX
AAL5_RX_AAL5_PDU_CRC_BAD	Counts the number of AAL5 frames with bad CRC
AAL5_RX_AAL5_PDU_LEN_BAD	Counts the number of AAL5 frames with bad Length

6.4.3 Hash Table

The hash table is used to get a pointer to the VC information in SRAM. Two hash tables—primary and secondary are both maintained in SRAM. Each entry in the primary hash table maintains two VC keys and if required a pointer into the secondary table. The secondary table is used to store overflows from the primary table. Each entry in the secondary table stores up to four VC keys and a pointer to the next hash bucket in the link list.

Table 6-7 shows the CRC32 from two LW (as init value of CRC32 count we assume zero) used to find a hash table.

Table 6-7. CRC32 From Two LW Used to Find a Hash Table

LW	Bits	Size	Field	Description
0	31	1	Not used	
	30:24	8	Reserved	Reserved
	23:16	8	VPI	Virtual Path Identifier
	15:0	16	VCI	Virtual Channel Identifier
1	31:11	21	Reserved	Reserved
	10:0	11	Input_port	Input port maintained from SPI4 port, and FPGA port

Table 6-8 shows the buckets in the primary hash table.

Table 6-8. Buckets Structure in the Primary Hash Table

LW	Bits	Size	Field	Description
0	31	1	Valid_bkt	Set by cp if bucket is valid
	30:0	31	VC_key1	First VC_key
1	31:16	16	VC_ptr2	Index of the RXC (for VC_key2) in VC table, range[0, 64K-1]
	15:0	16	VC_ptr1	Index of the RXC (for VC_key1) in VC table, range[0, 64K-1]
2	31:0	32	VC_key2	Second VC_key
3	31:28	4	VC_key1_a	VC_key1 extension
	27:24	4	VC_key2_a	VC_key2 extension
	23:16	8	Reserved	Reserved
	15:0	16	Next_bkt_index	Index of the next secondary bucket in the chain

Table 6-9 shows the buckets in the secondary hash table.

Table 6-9. Buckets Structure in the Secondary Hash Table

LW	Bits	Size	Field	Description
0	31	1	Valid_bkt	Set by cp if bucket is valid
	30:0	32	VC_key1	First VC_key
1	31:16	16	VC_ptr2	Index of the RXC (for VC_key2) in VC table, range[0, 64K-1]
	15:0	16	VC_ptr1	Index of the RXC (for VC_key1) in VC table, range[0, 64K-1]
2	31:0	32	VC_key2	Second VC_key
3	31:0	32	VC_key3	Third VC_key
4	31:16	16	VC_ptr4	Index of the RXC (for VC_key4) in VC table, range[0, 64K-1]
	15:0	16	VC_ptr3	Index of the RXC (for VC_key3) in VC table, range[0, 64K-1]

Table 6-9. Buckets Structure in the Secondary Hash Table

LW	Bits	Size	Field	Description
5	31:0	32	VC_key4	Fourth VC_key
6	31:0	32	Reserved	Reserved
7	31:28	4	VC_key1_a	VC_key2 extension
	27:24	4	VC_key2_a	VC_key3 extension
	23:20	4	VC_key3_a	VC_key4 extension
	19:16	4	VC_key4_a	VC_key4 extension
	15:0	16	Next_bkt_index	Index of the next secondary bucket in the chain

Table 6-10 shows the VC_key#

Table 6-10. VC_key#

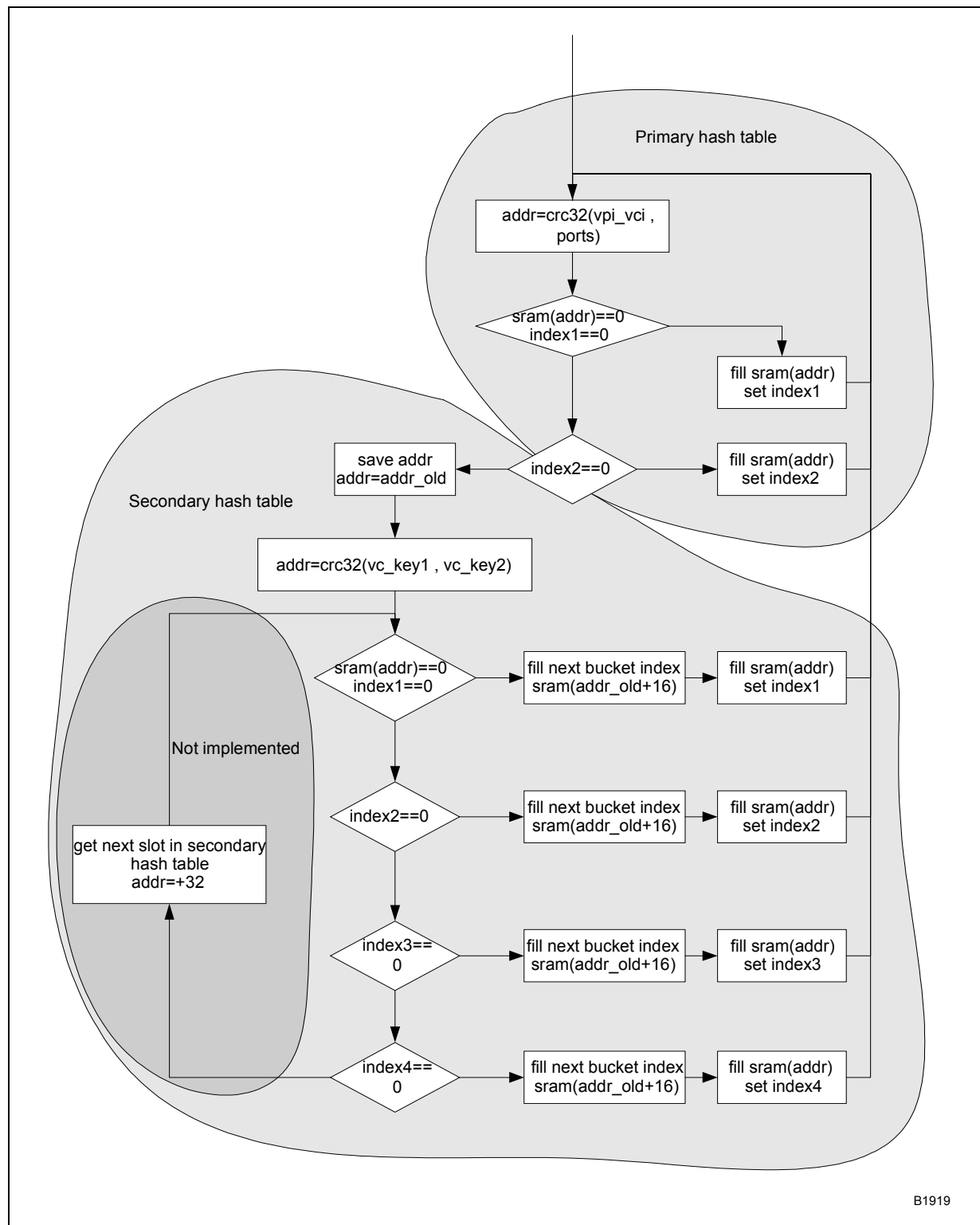
Bits	Size	Field	Description
31	1	Reserved	Reserved
30:24	7	Input_port	7 LSB of input port
23:16	8	VPI	Virtual Path Identifier
15:0	16	VCI	Virtual Channel Identifier

VC_key#_a include 4 MSB of input port.

6.4.3.1 Collisions Resolutions

Figure 6-5 illustrates collisions resolutions in the hash table.

Figure 6-5. Collisions Resolutions



B1919

6.5 Algorithm and Pseudocode

6.5.1 Issues and Challenges in the Implementation

- The AAL5 RX block needs to maintain the sequence in which cells arrive. Also reassembled packets are passed to the next pipestage in the order in which they arrive on the MSF media interface. This is achieved by making sure that the threads execute in order.
- Neither the 64-bit RSW (Receive Status Word for each RBUF element) in the UTOPIA mode nor the ATM cell supports an SOP bit. Therefore to determine when the thread has received an SOP cell, it depends on the expect_SOP bit of RXC. This bit is set when the EOP cell of the last frame for this VC is received.
- The ATM RX microblock supports up to 64K virtual circuits. This implies that the ATM RX block has to store RXC data structures in SRAM and use local memory as a cache. In the two microengine design, a reference count per RXC is kept in local memory, which is incremented by one every time a thread accesses the RXC for the first time and decremented when it is done using the RXC. The RXC is written back to SRAM from local memory when the reference count is zero.
- The ATM RX has been designed to have separate code paths for SOP_EOP, EOP_only, MOP_only and SOP_only, to optimize the overhead of checking for EOP and RXC—exp_SOP bits in the critical section.
- To minimize the memory latency associated with the allocation of a new buffer, each thread in the ATM RX block pre-fetches a buffer from the free list. During initialization, each thread allocates a buffer and stores the handle in a general-purpose register. On SOP or when the current buffer is full, the pre-fetched buffer is used and a request is made to allocate another buffer. Thus at any given time, the thread always has a buffer allocated.
- Cell count calculation for SOP_EOP case.
For CSIX TX, the CSIX prepend header is added for every packet, in addition to the c-frame header for every c-frame. This header is included in the cell count calculation. This per packet prepend header is added on the fly and so its not counted in buffer size/packet_size.
The CSIX cell is 120B. For the first CSIX cell of the packet there is a packet prepend header of 8B. So the first CSIX cell is 112B.
Therefore for eop_sop, the assumption is that the cell count is 0 (actual 1 cell)
- Cell count calculation for SOP_ONLY.
For CSIX TX,

$$\text{byte_remainder} = \text{cell_size (i.e. bytes_received)} + \text{per_pkt_prepend_bytes}$$

$$\text{cell_count} = 0.$$
 And no need to further compute the cell_count.
This is only applicable for first CSIX frame (SOP_only case).
- Cell count calculation for extra_cell, considering TX_CELL_SIZE = 120B(CSIX TX)
When an extra cell is received with the AAL5 trailer only.
For CSIX TX there is no change in the cell_count computed from the previous iteration as there is an extra_cell.

6.5.2 Algorithm

6.5.2.1 Two Microengine Design

In this case, the ATM RX microblock is implemented as a three stage functional pipeline running on two microengines. The function of each stage and its phases are summarized below. Each pipe-stage is a critical section and a microengine must wait for an inter-ME signal before entering any stage.

Phase1 (Wake up from the thread freelist)

```
Read RSW register
    Check for rsw errors (rsw_parity_err, idle_request, cell_sz_err)

Issue MSF read of RBUF into xfer reg

Get RXC ptr (hash_lookup - at least 1 wait[sram_read_sig])
    Check for hash_miss

If ctx 0 {Wait [CS1_inter_me_sig, rbuf_read_sig, next_thread_sig]
        cam_clear}
Else Wait[rbuf_read_sig, next_thread_sig]
```

Pipestage1 (Critical section1):

Phase2:

```
Free up the rbuf element

Signal next thread to enter phase 2

Get RXC1 from SRAM (Use cam -with folding)
    Wait [rx_c_in_sig, next_thread_sig] (cam_miss)
    or Wait [next_thread_sig] (cam_hit)
```

Pipestage1 cntd (Critical section1):

Phase3:

```
Decrement the ref_cnt by 1

Check for AAL type, VC_state (valid/ invalid), management cell
(Here its AAL5, VC_valid, user_cell)

Signal next thread to enter phase 3
Check for EOP, SOP, MOP, EOP_SOP types (jump table)

Reassembly macro:
    Compute CRC
    Update RXC1
    If(ref_cnt == 0) {Write back RXC1 to SRAM}
        If(ctx == 7) {Signal next ME CS1_inter_me_sig - cap_wr cam_clear}
    If (ctx = 0) Wait [CS2_inter_me_sig, next_thread_sig, rxc_write_sig]
        if (rx_c_write)]
    Else Wait [next_thread_sig, rxc_write_sig (if rx_c_write)]
```

```

Pipestage2 (Critical section2):
Phase 4:

Reassembly macro (cntd):

Signal next thread to enter phase 4

Get RXC2 from SRAM
    Wait [rxc_in_sig, next_thread_sig] (cam_miss)
    Or Wait [next_thread_sig] (cam_hit)

Pipestage2 cntd (Critical section2):
Phase 5:

Reassembly macro (cntd):

Decrement ref_cnt by 1
Signal next thread to enter phase 5

Update RXC2

Write Meta data to SRAM

If(ref_cnt == 0) {Write back RXC2 to SRAM - not reqd when EOP}
    If(ctx ==7) {Signal next ME CS2_inter_me_sig - cap_wr}

Write data to DRAM

Prefetch a buffer (if SOP cell/ Buffer overflow)

Set some meta data fields using dispatch loop macros
Update dispatch loop variables
    (dl_next_block, dl_eop_buf_handle, dl_buf_handle)

If (ctx = 0)
    Wait[CS3_inter_me_sig, meta_wr_sig, data_wr_sig,
        bd_prefetch_sig,
        next_thread_sig, rxc_write_sig(if rxc_wr)]
Else
    Wait [meta_wr_sig, data_wr_sig, bd_prefetch_sig,
        next_thread_sig,
        rxc_write_sig (if rxc_wr)]

```

```

Pipestage3 (Critical Section3):
Phase 6:

Signal next thread to enter phase 6
Put onto the thread free list

Dl_Sink
    If (dl_next_block = ATM_RX_NEXT1)
        Write on to scratch
        If(ctx ==7) {Signal next ME CS3_inter_me_sig - cap_wr}

Wait [rsw_rcvd_sig, scr_put(if any)]

Loop back to Phase 1

```

6.5.2.2 Single Microengine Design

In this case, ATM RX is implemented with following four phases to run on one microengine.

Phase1:

```
Read RSW
  Check for rsw errors (rsw_parity_err, idle_request,
    cell_sz_err)

Read RBUF into xfer reg

Get RXC ptr (hash_lookup - atleast 1 wait[sram_read_sig])
  Check for hash_miss

Wait [rbuf_read_sig, next_thread_sig]
```

Phase2:

```
Signal next thread to enter phase 2

Free up the rbuf ele (fast_wr)

Get RXC from SRAM (folding)
  Wait [rxc_in_sig, rxc_out_sig, next_thread_sig] (cam_miss)
  Wait [next_thread_sig] (cam_hit)
```

Phase3:

```
Check for AAL type, VC_state (valid/in_valid), mgt_cell
(Here its AAL5, VC_valid, user_cell)

Signal next thread to enter Phase 3
Check for EOP, SOP, MOP, EOP_SOP types (jump table)
Reassembly macro:
  Compute CRC

  Write Meta data to SRAM

  Update RXC

  Write data to DRAM

  Prefetch a buffer (if SOP cell / Buffer overflow)

Set some meta data fields using dispatch loop macros
Update dispatch loop variables
  (dl_next_block, dl_eop_buf_handle,
    dl_buf_handle)

Wait [meta_wr_sig, data_wr_sig, bd_prefetch_sig,
  next_thread_sig]
```

```
Phase4:
Signal next thread to enter Phase 4
Put onto the thread freelist to receive next cell

Dl_sink
(dl_next_block = ATM_RX_NEXT1) Write on to scratch
Wait [rsw_rcvd_sig, scratch_put_sig (if any)]

Loop back to phase 1
```

6.6 Flow Chart (Two Microengine Design)

Figure 6-6. IXP2000 ATM RX Flow Chart

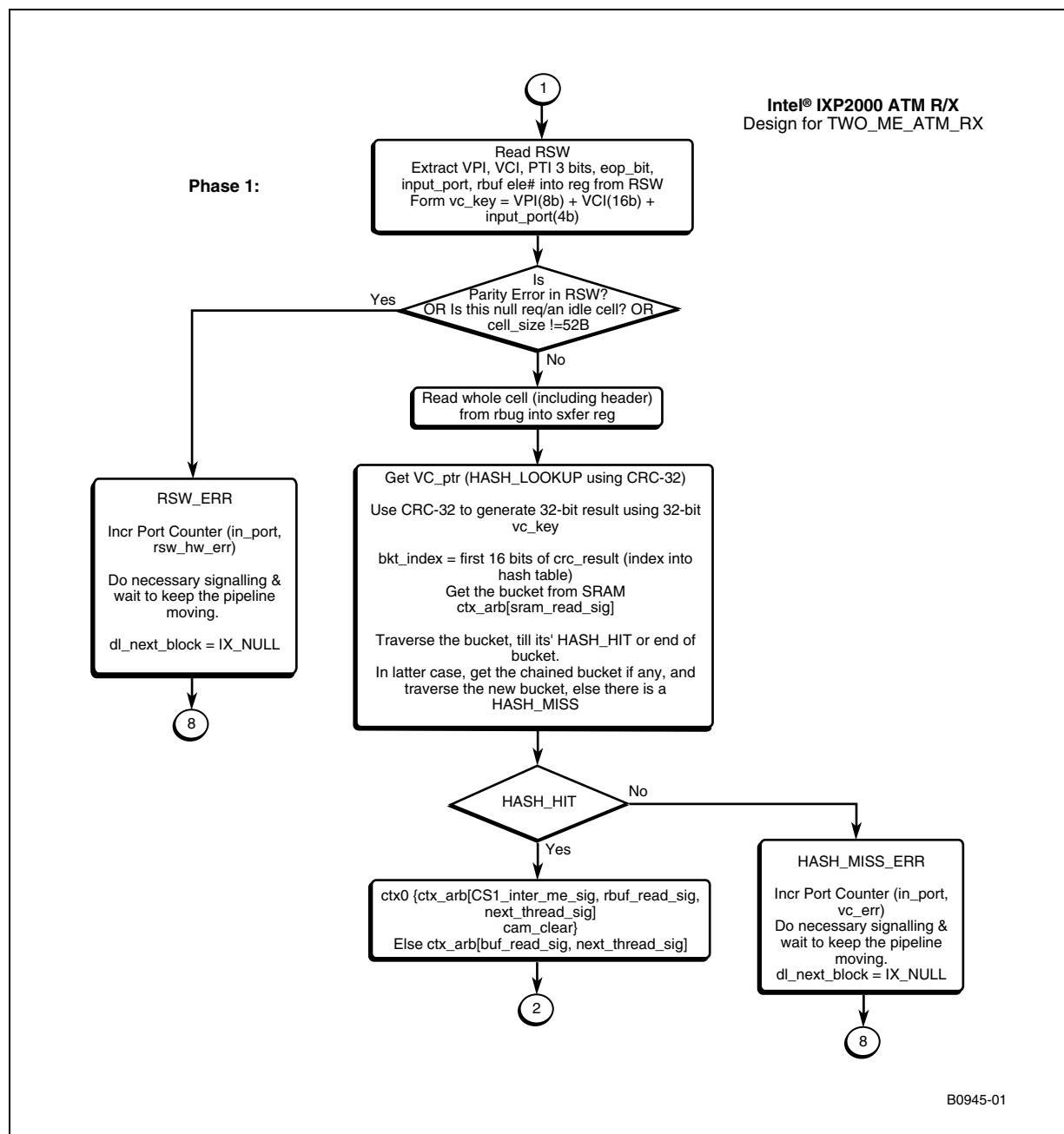


Figure 6-7. Pipe Stage 1 Start: Phase 2 Flow Chart

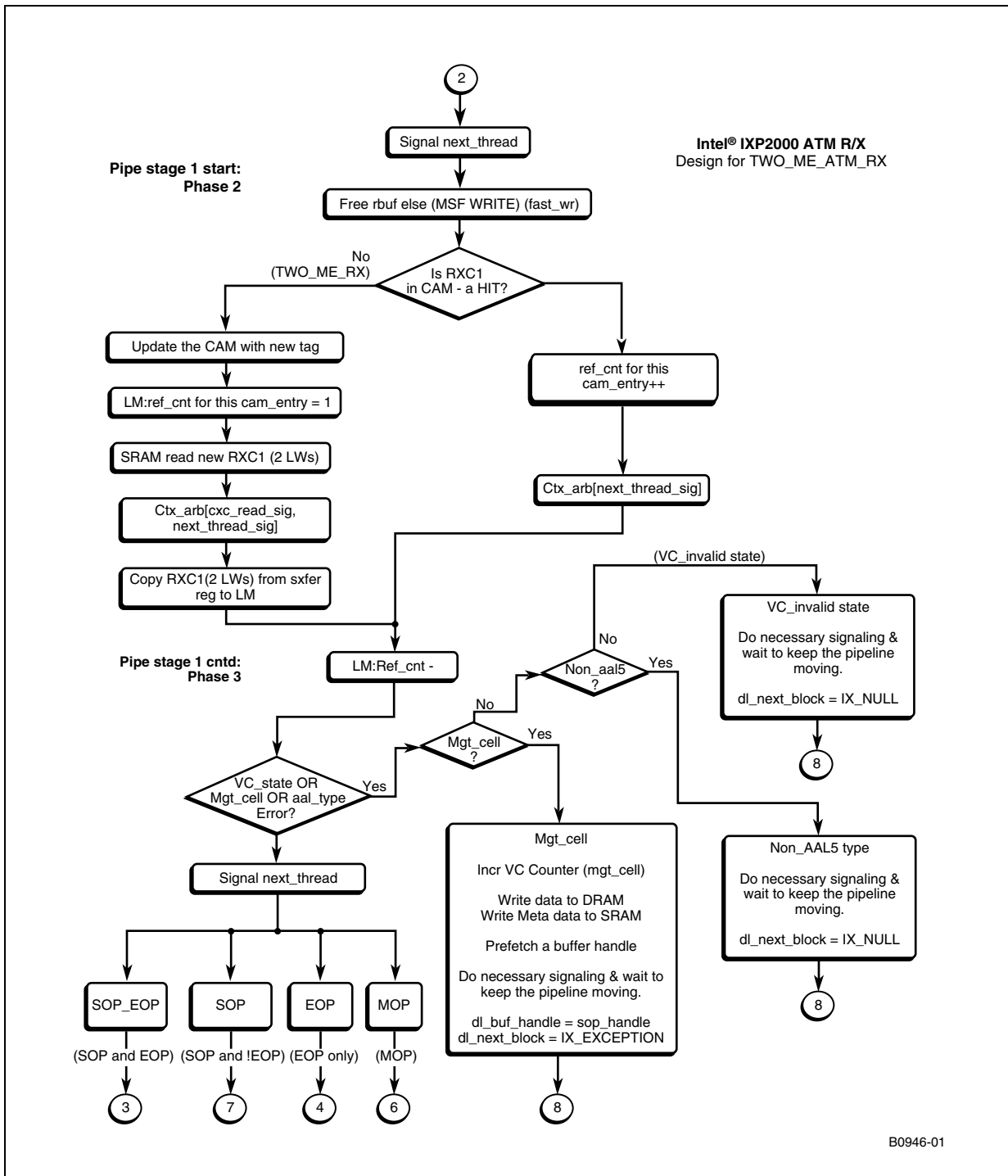


Figure 6-8. SOP and EOP Flow Chart

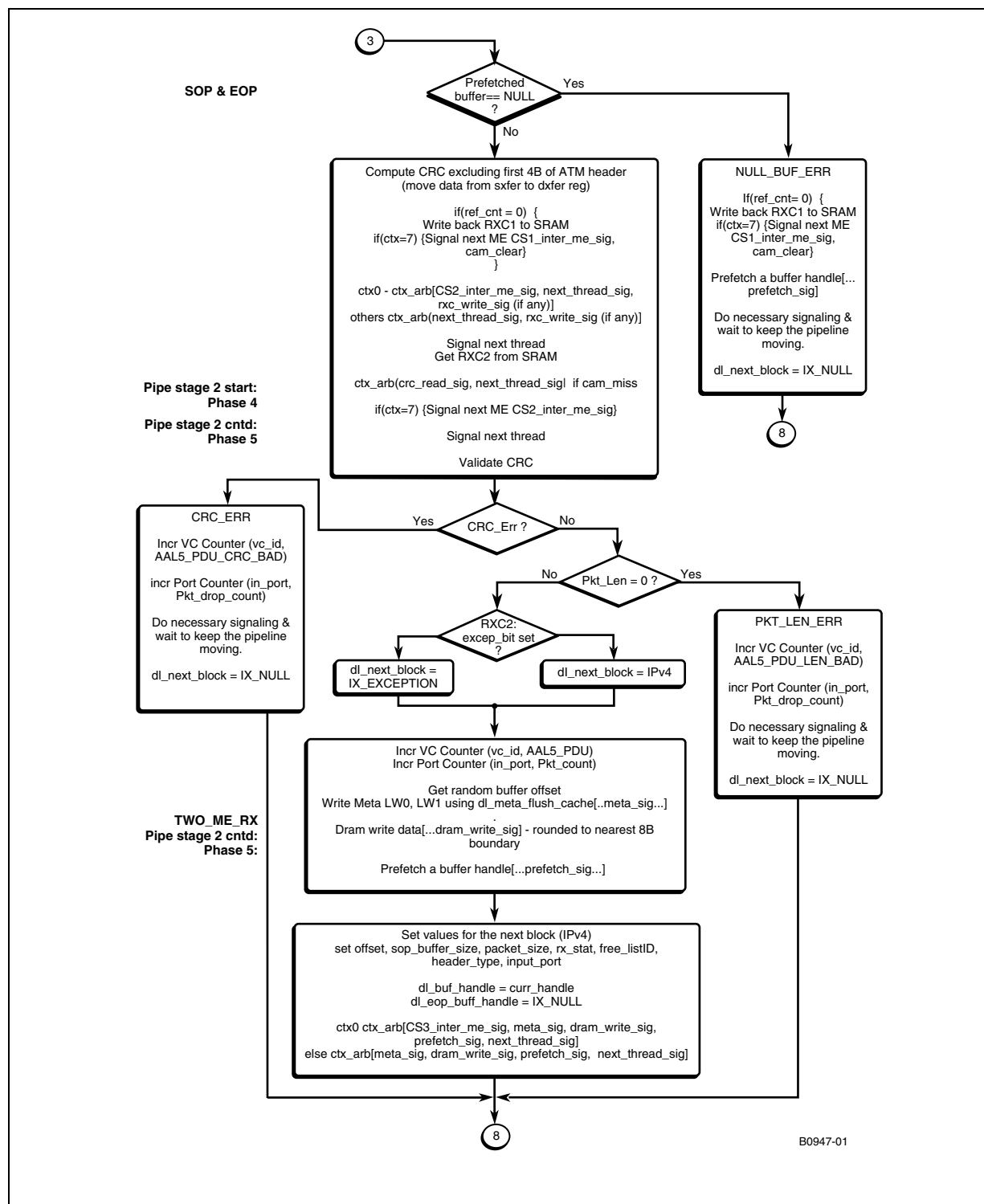


Figure 6-9. EOP Only Flow Chart (Page 1 of 2)

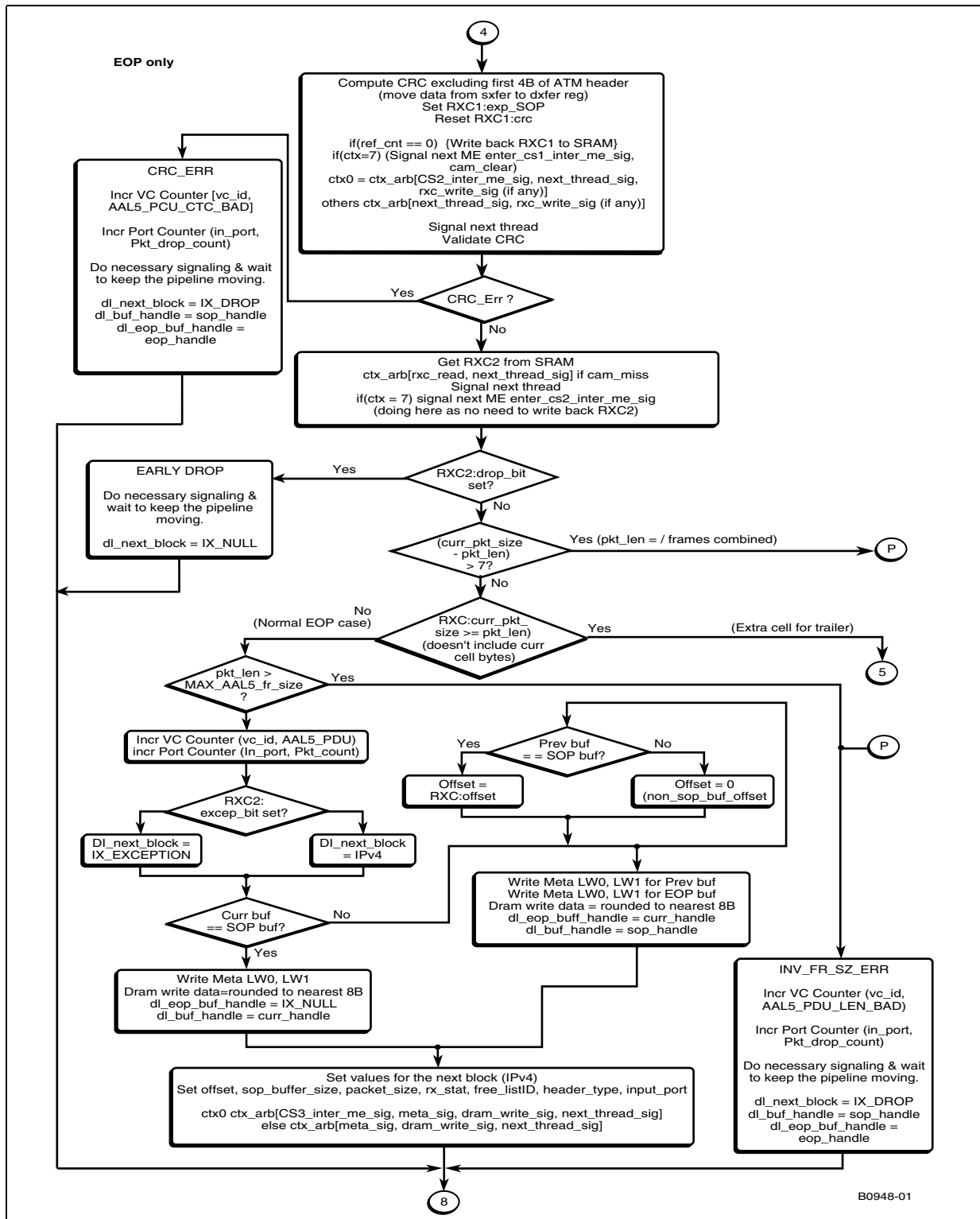


Figure 6-10. EOP Only Flow Chart (Page 2 of 2)

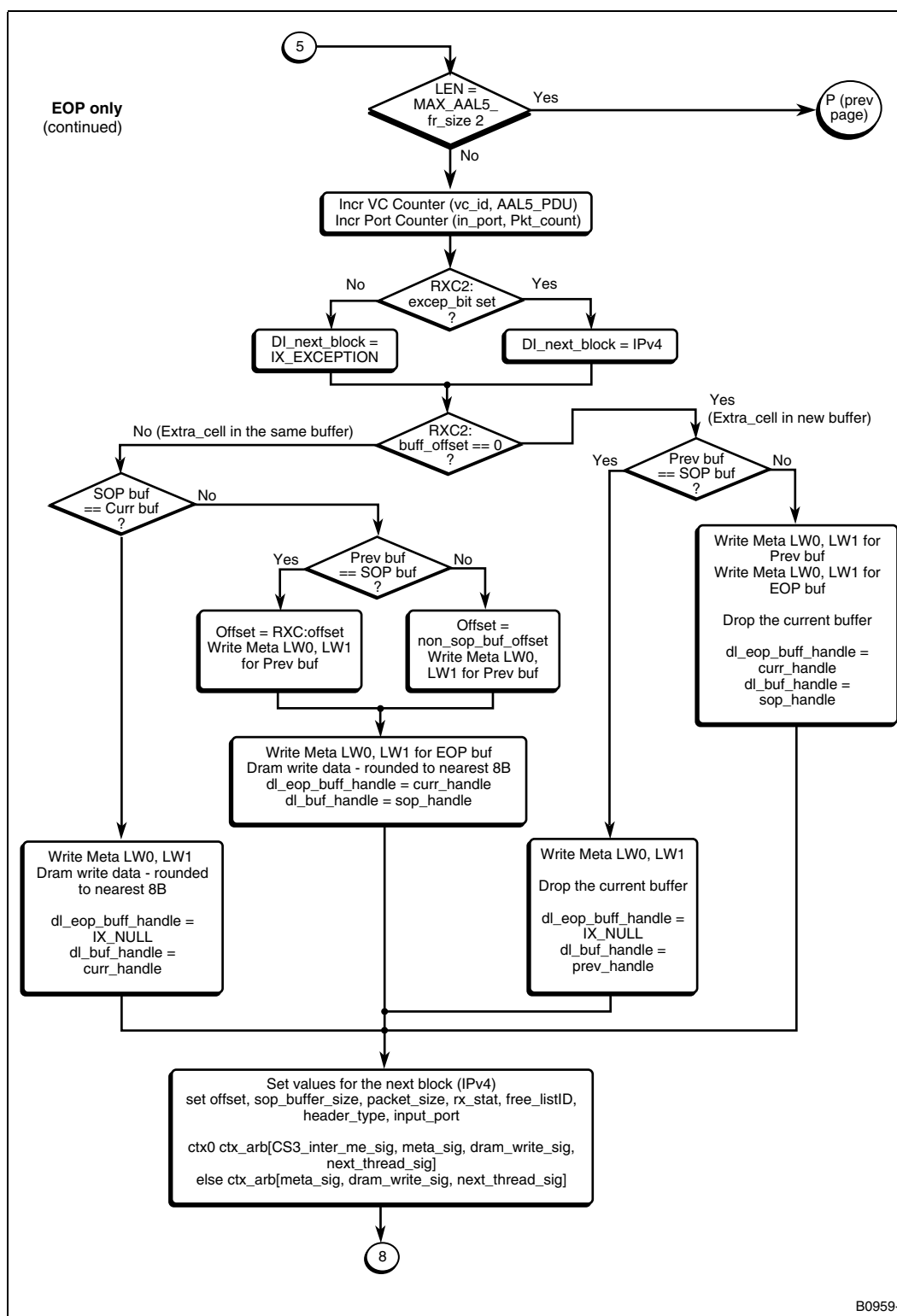


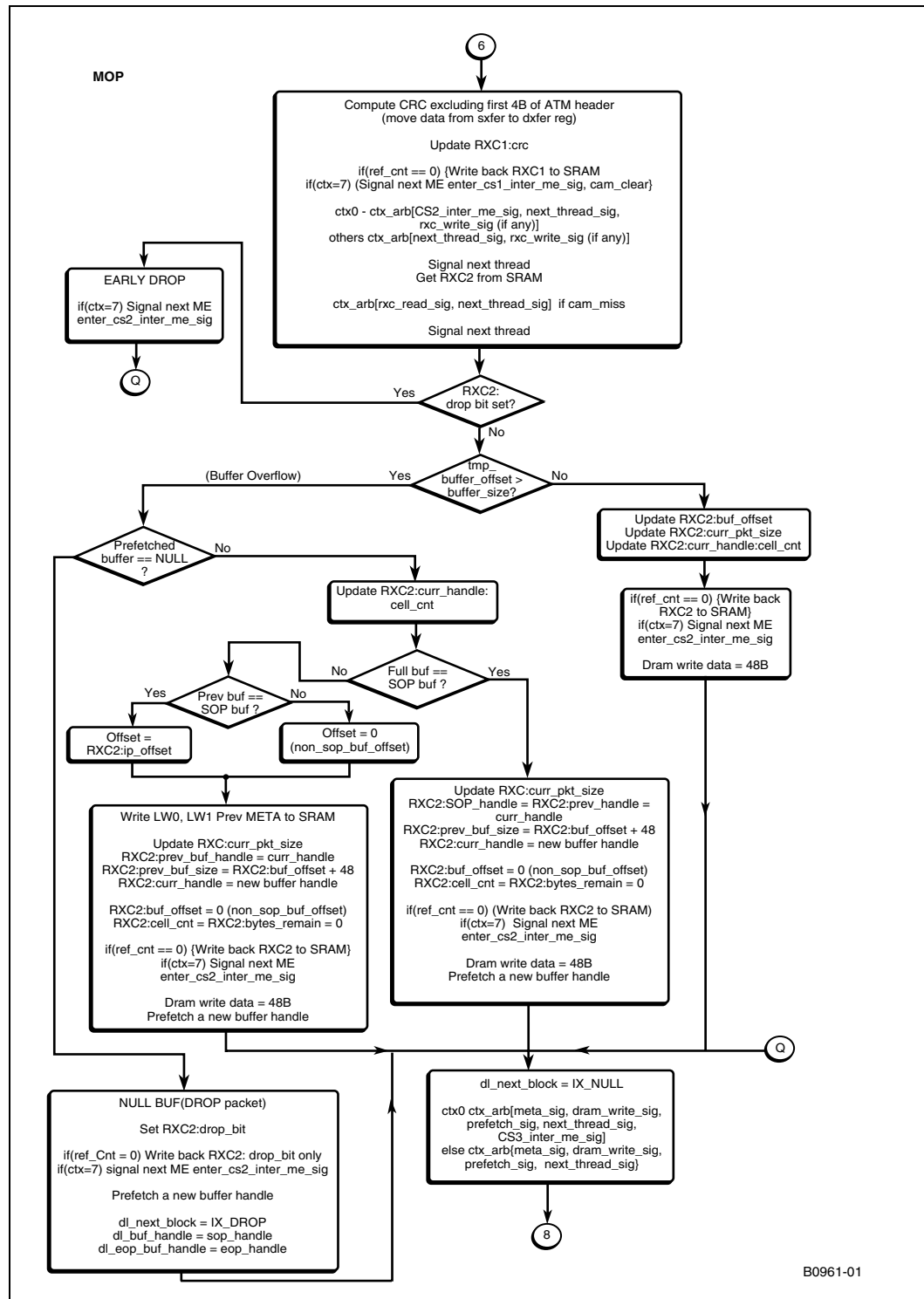
Figure 6-11. MOP Flow Chart


Figure 6-12. SOP Only Flow Chart

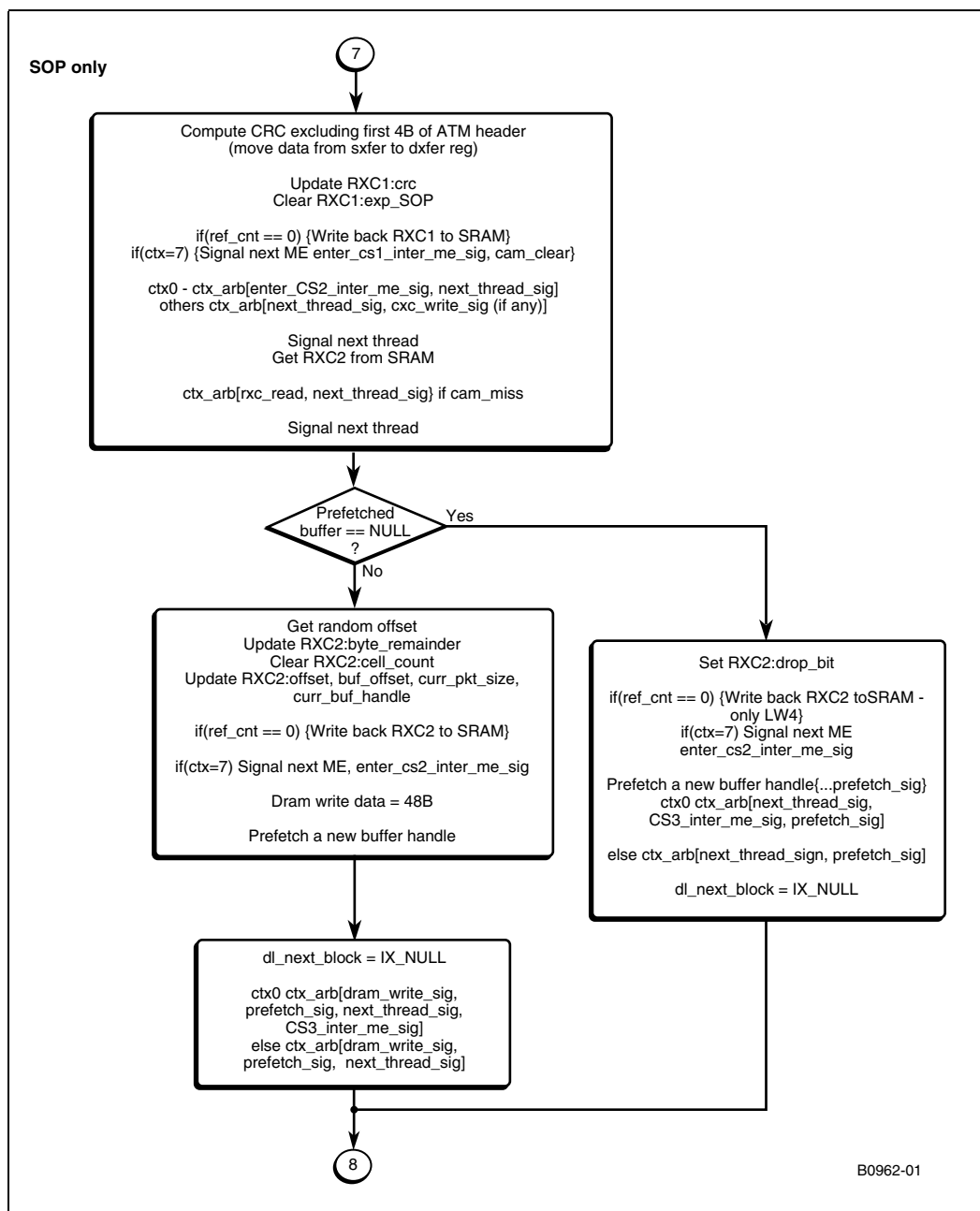
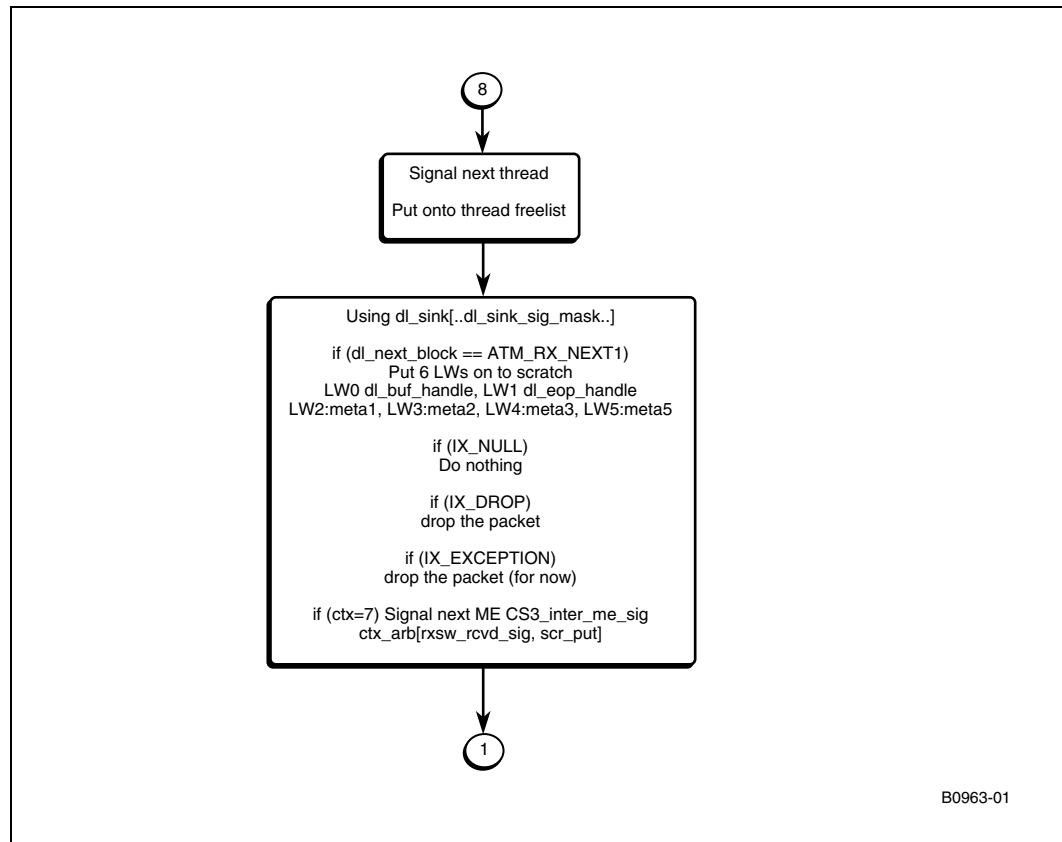


Figure 6-13. Pipe Stage 3: Phase 6 Flow Chart



6.7 Performance Analysis

This section includes the performance estimates for the ATM RX microblock for the critical path or the longest path of the design. ATM RX runs on two microengines in the functional pipeline to achieve OC48 performance. Same code may be used with the ONE_ME_AAL5_RX compiler to run on one ME and achieve OC-12 performance.

6.7.1 Two Microengine Design

Tables 6-11 and 6-12 show the instruction estimates and break up of the critical path cycles for the worst case scenario for ATM RX microblock two microengine design.

Table 6-11. Instruction Estimates—Two Microengine Design

Block	EOP_SOP	SOP	MOP	EOP	Max allowed
Critical Path	32	32	32	32	
Critical Section 1 (Pipe stage 1)	81	80	77	79	100 cycles
Critical Section 2 (Pipe stage 2)	62 (signals next ME after 22 instr)	54 (signals next ME after 43 instr)	Worst case: 89 (signals next ME after 75 instr)	Worst case: 95 (signals next ME after 21 instr)	100 cycles
Critical Section 3 (Pipe stage 3)	23	23	23	23	100 cycles
Total cycles	198	189	Worst case: 221	Worst case: 229	212 cycles
Avg I/O latency	1600				1696 cycles
Control store usage	1300				4096 instr

Table 6-12. Break up of the Critical Path Cycles for the Worst Case Scenario

Phases	EOP_SOP	SOP	MOP	EOP	Maximum Allowed
Critical Path (Phase1)	32	32	32	32	
Pipe stage 1 (Phase 2)	15	15	15	15	
Pipe stage 1 (Phase 3)	17 + 49	17 + 48	17 + 45	17 + 45	
Pipe stage 2 (Phase 4)	16	16	16	16	

Table 6-12. Break up of the Critical Path Cycles for the Worst Case Scenario (Continued)

Phases	EOP_SOP	SOP	MOP	EOP	Maximum Allowed
Pipe stage 2 (Phase 5)	46	38(3 -cell_cnt ^a)	Buf_not_full: 52 (10-cell_cnt ^a) buf_full_SOP: 67 (13-cell_cnt ^a) buf_full_!SOP: 73 (13 - cell_cnt ^a)	Pkt_in_one_buf:70 (11-cell_cnt ^a) PKT_in_one+_buf:7 9 (11-cell_cnt ^a)	
Pipe stage 3 (Phase 6)	23	23	23	23	
Total Cycles	198	189	200, 215, 221	220, 229	212 cycles

a. Cell_cnt—indicates the number of cycles incurred in counting the number of csix frames per buffer. If POS TX/ETHER TX is used in place of CSIX TX, the total instruction count would reduce by Cell_cnt number.

Note: This estimate includes instructions for counters update (updates three counters for EOP case—9 instructions)

6.7.2 Single Microengine Design (4 threads per OC-12 port)

Table 6-13 shows the instruction estimates and I/O usage for ATM RX microblock.

Table 6-13. Instruction Estimates and I/O Usage for ATM RX Microblock

Phases	EOP_SOP	SOP	MOP	EOP	Maximum Allowed
Phase1	43	43	43	43	
Phase 2	34	34	34	34	
Phase 3	14 + 76	14 + 55	14 + Buf_not_full: 66 Buf_full_SOP: 84 Buf_full_!SOP: 90	14 + Pkt_in_one_buf:99 PKT_in_one+_buf:108	
Phase 4	22	22	22	22	
Total Cycles	189	168	179, 197, 203	212, 221	424 cycles
Control store usage	1250				4096 instr
Avg I/O latency	1300				106*4*8 = 3392 cycles

Note: This estimate includes instructions for counters update (updates three counters for EOP case—9 instructions)

6.8 Resource Usage

Tables 6-18 and 6-19 show the SRAM and local memory usage for ATM RX microblock.

Table 6-14. SRAM Memory

SRAM Channel	Data structure	Total No.	Unit Size (Bytes)	Total Bytes Used	Location	Max Memory Available
Channel0						8MB
	Port statistic counters	16 Ports	8LW * 4	512 B	@ 2.5 MB	
	VC Info Table	64K bkts	16LW * 4	4 MB	@ 4.0 MB	
	I. RXC Table		9LW * 4	2304 KB		
	II. VC statistic counters		7LW * 4	1792 KB		
Channel1						8MB
	Primary Hash table	64K bkts	4LW * 4	1 MB	@ 5MB	
	Secondary Hash table	1K bkts (can be changed to 4K-8K)	8LW * 4	32 KB	@ 6MB	

Table 6-15. Local Memory

Data Structure	Total No.	Unit Size (LWs)	Total LWs Used	Max LWs Available
RXC	16	16	256	
Constants	16	1	16	
			272	640

6.8.1 Control store

Control store usage = 1300 instructions (Maximum allowed is 4K)

6.8.2 Registers and Signals

Tables 6-16 and 6-17 show the registers and signals for two and single microengine design

Table 6-16. Registers and Signals—Two Microengine Design

Parameter	Used ^a	Budget	Unit
GP registers	25	32	
SRAM registers	10	16	
DRAM registers	15	16	
Signals	11	15	

- a. This is the maximum usage during an execution path. It does not imply that all these registers/signals are used all the time.

Table 6-17. Registers and Signals—Single Microengine Design

Parameter	Used ^a	Budget	Unit
GP registers	24	32	
SRAM registers	10	16	
DRAM registers	13	16	
Signals	9	15	

- a. This is maximum usage during an execution path. It does not imply that all these registers/signals are used all the time.

6.8.3 I/O Commands

Tables 6-18 and 6-19 show the I/O commands for two and single microengine design.

Table 6-18. I/O Commands—Two Microengine Design

Phase no	Description	Which command	Data size
1	Read RBUF into sxfer reg	Rbuf read	13 LWs
	Get VC ptr (hash Lookup)	SRAM read (primary table)	4LWs
		SRAM read (secondary table), if required	8LWs
2	Free up RBUF element	Msf fast write	
	Get RXC1 from SRAM (if cam_miss)	SRAM read	2LWs
3	Write back RXC1 to SRAM (if ref_cnt =0)	SRAM write	2 LWs
4	Get RXC2 from SRAM(if cam_miss)	SRAM read	8LWs
5	Write Meta data to SRAM(EOP_SOP) - 1 sram write(SOP) - none(MOP) - max 1 sram write(EOP) - always 2 sram write	SRAM write	2LWs
	Write back RXC2 to SRAM(- if ref_cnt=0, - not reqd on EOP)	SRAM write	7 LWs
	Data write(on ex_cell (EOP cell with just trailer) no DRAM wr)	DRAM write(on EOP rounded to nearest 8B boundary)	12 LWs
	Prefetch a buffer(on SOP or buffer overflow)	SRAM dequeue	1LW
	Dropping an unused buffer (on ex_cell in new buff)	SRAM enqueue	1LW
6	Write to scratch_ring -dl_sink[]	SCRATCH write	6 LWs
	Put on to thread freelist	Msf fast write	

Table 6-19. I/O Commands—Single Microengine Design

Phase no	Description	Which command	Data size
1	Read RBUF into sxfer reg	Rbuf read	13 LWs
	Get VC ptr (hash Lookup)	SRAM read (primary table)	4LWs
		SRAM read (secondary table), if required	8LWs
2	Free up RBUF element	Msf fast write	
	Evict RXC to SRAM (if cam_miss)	SRAM write	9LWs
	Get RXC from SRAM (if cam_miss)	SRAM read	9LWs
3	Write Meta data to SRAM (EOP_SOP) - 1 sram write (SOP) - none (MOP) - max 1 sram write (EOP) - always 2 sram write	SRAM write	2LWs
	Data write (on ex_cell (EOP cell with a trailer only) no DRAM wr)	DRAM write (on EOP rounded to nearest 8B boundary)	12 LWs
	Prefetch a buffer(on SOP or buffer overflow)	SRAM dequeue	1LW
	Dropping an unused buffer (on ex_cell in new buff)	SRAM enqueue	1LW
4	Write to scratch_ring -dl_sink[]	SCRATCH write	6 LWs
	Put on to thread freelist	Msf fast write	

6.8.4 Characterization Data

Table 6-20. ATM AAL5 RX Microblock Characterization Data

Data	Value
General:	
Microblock Name	AAL5_Rx
Microblock Version Number	1
Implementation Language	microcode

Table 6-20. ATM AAL5 RX Microblock Characterization Data (Continued)

Data	Value
Configuration Options use to gather this set of data	## OC-48 AAL5 Rx ## 1. RX_PHY_MODE = SPHY_1_32 2. TWO_ME_RX_FP 3. META_CACHE_SIZE=6 4. USE_HASH_LOOKUP 5. NON_AAL5_TX 6. CSIX_TX 7. NUM_OF_INPUTPORTS_16 8. METADATA_2LW
	## 4 OC-12 AAL5 Rx ## 1. RX_PHY_MODE = SPHY_4_8 2. ONE_ME_RX_FP 3. META_CACHE_SIZE=6 4. USE_HASH_LOOKUP 5. NON_AAL5_TX 6. CSIX_TX 7. NUM_OF_INPUTPORTS_16 8. METADATA_2LW
Measurement Environment (tool settings)	N/A

Performance: (cycle counts, latencies, bandwidths)	
Instruction cycle count per packet (common case packet / worst case cycle count)	## OC-48 AAL5- Rx ## (SOP_EOP min pkt – 2 cells) 198 (SOP case) 189 (MOP case) 221 (EOP case) 229
	## 4 OC-12 AAL5 Rx ## SOP_EOP min pkt – 2 cells) 173 (SOP case) 154 (MOP case) 188 (EOP case) 203 *(Worst case; includes aborted cycles due to branches; disabled statistics)
Common-case packet/path assumptions to be documented here	1. Min Pkt consists of two ATM cells. 2. Performance measured with hash lookup hit at first entry in the bucket. 3. Receives 52-byte ATM cell after stripping off 1-byte of HEC.
Scratch Memory	
# of longwords read (for bandwidth calculations)	
# of longwords written (for bandwidth calculations)	6
# and type of each atomic operation performed (for bandwidth calculations)	
SRAM	
# of longwords read	14

Table 6-20. ATM AAL5 RX Microblock Characterization Data (Continued)

Data	Value
# of longwords written	11
# and type of each atomic operation performed (for bandwidth calculations)	
Co-processors	
bytes read (per channel)	
bytes written (per channel)	
DRAM	
# of quadwords read	
# of quadwords written	12
Other I/O instructions (i.e. MSF, PCI, CAP, and Hash)	msfRd msfFastWr (2 times) OC-48 AAL5 Rx: Read RBUF (msf rd 13) Hash lookup (sram rd 4) Free RBUF element (msffastwr) Get RXC1 (sram rd 2) Flush RXC1 (sram wr 2) Get RXC2 (sram rd 7) Write Meta (sram wr 2) Flush RXC2 (sram wr 7) Data write (dram wr 12) Prefetch a buffer(sram deq 1)
List of dependent I/O accesses in the longest latency path	dl_sink (scr wr 6) Put on to the freelist (msffastwr) OC-12 AAL5 Rx : Read RBUF (msf rd 13) Hash lookup (sram rd 4) Free up RBUF element (msffastwr) Flush RXC (sram wr 9) Get RXC (sram rd 9) Write Meta (sram wr 2) Data write (dram wr 12) Prefetch a buffer (sram deq 1) dl_sink (scr wr 6) Put on to the freelist (msffastwr)
Per-Microengine Resources:	
Control-Store Usage (# of instructions used)	1300
Local Memory Footprint (# of long words used)	272
Local Memory Configuration (shared, or per-context pointer)	272 (shared)
Local Memory - # of LM pointers used	2

Table 6-20. ATM AAL5 RX Microblock Characterization Data (Continued)

Data	Value
GPR Usage – minimum, static usage (absolute, static, globals)	
Transfer Reg. Usage – minimum, static usage	
Next Neighbor Reg. Usage – minimum, static usage	N/A
Signal Usage – minimum, static usage	11
CAM used? (yes or no)	YES

Global Resources:

Scratch footprint (# of longwords used) – constant or formula ...	1024 (Scratch Ring to the next microblock)
	Port stats 16 * 8
SRAM footprint (# of longwords used) – constant or formula ...	VC InfoTable(RXC+Stats) 64K * 16
	Primary Hash table (64K * 4)
	Sec Hash table (1K * 8)
DRAM footprint (# of quadwords used) – constant or formula ...	
Q-Array usage - # of queues used and if they need to be cached	N/A
CRC Unit used?	YES (CRC-32)
Hash Unit used? (yes or no)	N/A

MSF Usage Information:

Media Bus Configuration	UTOPIA Level3
RBUF, TBUF usage	8 KB Total RBUF (64B RBUF)
CBus signals	

Other Information:

Critical Section Length (compute cycles + memory accesses)	OC-48 AAL5 Rx Critical section instr (worst case): 79 + 95 + 23 4 OC -12 AAL5 Rx Phases instr (worst case): 32 + 34 + 118 + 20
# of phases	OC-48AAL5Rx:3criticalsections,(6phases) 4 OC-12 AAL5 Rx: 4 phases
Packet Metadata - fields read	
Packet Metadata - fields written	Metadata Wr - buffer_next, buffer_size, offset Meta data on scratchring - sop_buf_size, sop_buf_offset, pkt_size, free_list_id, rx_stat, header_type, input_port, vc_id
Header - fields read	N/A
Header - fields written	N/A
Entry and Exit Setup (assumptions on ME context)	

Table 6-20. ATM AAL5 RX Microblock Characterization Data (Continued)

Data	Value
# of MEs required to run a single instance of microblock	
Data Rates	
Memory Map of Key Data Structures	
Performance Considerations - ANY issues that have a direct impact on performance	
Documentation:	
Thread Ordering Requirements	Hyper Task Chaining
OS dependencies	N/A
Chip/hardware dependencies (i.e. Crypto Unit in 2850)	
Tested on which SDK Release(s)	IXA SDK 3.5
Tested on hardware? Which hardware configuration?	YES, 4 OC-12 AAL5 Rx
Tested in which applications (not an all inclusive list)	4OC-12AAL5Ingress(simulation&hardware), OC-48 AAL5 Ingress(simulation)
Possible Configuration Options	OC-48 AAL5 Rx (2 ME functional pipeline) with 16 input ports. 4 OC-12 AAL5 Rx (2 MEs - 4 threads per port) with 16 input ports.
Major limitations (i.e. Rx/Tx microblocks written to use entire ME)	The same code can be used for OC48 AAL5 Rx (2 ME functional pipeline) and 4OC12 AAL5 Rx (2 MEs - 4 threads per port) by changing the build option.
Packet Sequencing Issues (esp. in POT applications)	AAL5 Rx expect RBUFs to come in order from MSF. The CRC/packet length check determines the cell loss if any.
Core Component or Interface requirements or dependencies	

Transmit

The Transmit microblocks section includes the following chapters:

- [Chapter 7, “CSIX TX Microblock”](#)
- [Chapter 8, “Packet TX for SPHY and MPHY-4”](#)
- [Chapter 10, “Packet Transmit for OC-192 POS”](#)
- [Chapter 11, “ATM AAL5 TX Microblock”](#)
- [Chapter 12, “Packet TX–Multiports Microblock”](#)

Transmit microblocks segment a packet into many TBUF elements and transmits one TBUF element at a time. For each packet, packet data is read from DRAM and packet meta data is read from SRAM. Transmit microblocks vary by data rate (OC-48, OC-192), number of ports handled (1xOC-48, 16xOC-3, 10x1 GigE) and whether they handle packets (POS, Ethernet) or cells (c-frames, AAL5, AAL2).

The table below summarizes the different Transmit microblocks supported on the SDK:

Microblock	Description	Usage	Cycle Budget
sphy_mphy4	For PPP/POS and Ethernet frames.	Runs on one microengine. Can be used for 1xOC-48 (SPHY 1x32) or 4xOC-12 (SPHY 4x8 or MPHY 4)	97 per microengine.
mphy16	For PPP/POS and Ethernet frames	Runs on one microengine. Supports 16 ports.	97 per microengine.
Sphy_mphy4_2800	For PPP/POS and Ethernet frames	Supports 1xOC-192 and 1x10 GigE port. Runs on two microengine in a pipeline.	57 per microengine for OC-192 and 94 for 10 GBE.
Packet_tx_16ports	For Ethernet Frames	Supports 10x1 GigE. Max up to 16 ports. Runs on two microengines in parallel.	94 per microengine.
CSIX_TX	Segments a packet in to c-frames and transmits into fabric.	May be run on one or two microengines depending on the data rate. For IXP2400 OC-48, 4 GBE and IXP2800 10 GBE data rates, a single microengine is used. For OC-192 data rates 2 microengines in a pipeline are used.	57 per microengine for OC-192 POS on IXP2800. Otherwise 97 for OC-48 POS on IXP2400 or 94 for 10 GBE on IXP2800
AAL5_TX	For ATM interface. Supports AAL-5 segmentation. May be combined with AAL-2 microblock	Supports OC-48 data rates with 2 microengines on the IXP2400 and 1 microengine with the IXP2800.	105 per microengine for OC-48 on the IXP2400 and 248 per microengine for OC-48 on the IXP2800

7.1 Overview

The CSIX transmit microblock transmit cells (c-frames) into the CSIX fabric. It is designed such that via a compile time option, it can run on a single microengine or to run on two microengines in a context pipeline connected by a Next Neighbor ring. This allows it to support both OC-48 on the IXP2400 and OC-192 POS data rates on the IXP2800.

It receives transmit messages from the queue manager. With each transmit request, the microblock moves a cframe into a TBUF, which is then transmitted into the fabric by the MSF Transmit State Machine.

Every request has an associated packet, which is being segmented into cframes. The associated segmentation state (Transmit Context or TXC) for the packet and the packet metadata is cached in local memory. The TXC for the packet is looked up via the CAM using the queue id as the key. If the TXC is not cached in local memory, the LRU TXC is evicted to SRAM and the new TXC is read into local memory.

The TX microblock uses the TXC to compute the offset into the packet in DRAM to read data from. It allocates a TBUF element and moves data from DRAM to the TBUF. It adds the CSIX interface prepend header (Traffic Manager Header) on to the cframe along with the packet data. Along with the per-frame header, an optional per-packet header may also be added to the first c-frame of a packet as it is sent across the fabric. The headers carry reassembly information for the CSIX RX block on the egress IXP, sequence numbers for error detection and classification information (next hop ID, flow id, class id etc).

The TX microblock then validates the TBUF element and uses the MSF to send the data to the fabric. When all the data in a buffer is transmitted, it frees the buffer by placing it onto a buffer free list.

The CSIX TX block runs eight threads in parallel. The threads maintain coherence by using inter-thread signaling to run in strict order. The CAM is used to maintain a software cache of transmit contexts and the folding technique is used to optimize the read-modify write operations. In this implementation, each thread interleaves multiple transmit requests to cover the DRAM latency for moving the data from DRAM to TBUF.

7.2 Assumptions and Dependencies

- The CSIX Transmit block assumes that the cell count in each buffer is set up to include the per-packet prepend header (currently 8 bytes). In this design, this is done by the CSIX RX microblock.
- Currently the CSIX Transmit block adds an 8-byte per-cframe header and an 8-byte per packet header. It is possible to compress this information into the 8-byte per cframe header provided some assumptions are made:
 - The link level layer-2 header is at least 4 bytes. In the case of PPP, since this could be 2 to 4 bytes, this does not hold true. Hence we need to add a per-packet header. Since DRAM

is only accessible in 8-byte or quad word boundaries, the per-packet header must be a multiple of 8 bytes. But for Ethernet for example, the per-packet header is not needed.

- If we are not doing QoS, then the class ID, flow ID, etc. are not required. In this case also, the per-packet prepend is not needed.

7.3 Data Structures

This section describes the data structures used for the CSIX transmit microblock.

7.3.1 Transmit Context (TxC)

Table 7-1 describes the Transmit Context cached in local memory. At any time, 16 transmit contexts are cached in local memory and looked up using the CAM.

Table 7-1. CSIX Transmit Context

LW	Bits	Size	Field	Description
0	31:17	15	reserved	
	16:16	1	bd_valid	Thread meta data is valid
	15:0	16	bytes_sent_already	Bytes already sent in this buffer
1	31:24	8	align_port	offset for quad word alignment
	23:8	16	seq_numb	Sequence number of cell for current packet
	7:7	1	Eop_bit	Eop for multiple cframe case
	6:0	7	Free_list_id	Free list ID
2	31:16	16	next_hop_id	Next hop ID
	15:0	16	class_id	Class ID
3	31:16	16	total bytes from dram to tbuf	Buffer size in bytes plus the remainder of offset divided by 8
	15:0	8	offset	Offset of data in the buffer in bytes
4	31:0	32	reserved	
5	31:0	32	reserved	
6	31:0	32	reserved	
7	31:24	8	reserved	
	23:0	24	dl_buf_handle	Last dl_buf_handle for this queue number

7.3.2 Transmit Control Word (TCW)

Table 7-2 describes the Transmit Control Word (TCW) which is set up by the microblock. These are associated with each TBUF and are used by the MSF to create the CSIX base and extension headers. The first long word has information for the CSIX base header and the second long word has information for the CSIX extension header a unicast CSIX frame.

Table 7-2. CSIX Transmit Control Word

LW	Bits	Size	Field
0	31:24	8	Payload length
	23:21	3	Prepend offset
	20:16	5	Prepend length
	15:13	3	Payload offset
	12:10	3	Reserved
	9:9	1	CSIX reserved
	8:8	1	Private
	7:4	4	Reserved
	3:0	4	CSIX frame type
1	31:24	8	Class
	23:22	2	Private
	21:12	6	CSIX reserved
	11:0	12	Destination

7.3.3 TM Header—per C-Frame)

Table 7-3 describes the Traffic Manager (TM) header added per c-frame.

Table 7-3. CSIX Traffic Manager Header Added for Each c-frame

LW	Bits	Size	Field	Description
0	31:24	8	align_offset	offset into first quad word of payload
	23:8	16	seq_numb	Sequence number of cell for current packet
	7:7	1	eop	Last cell in a packet
	6:6	1	sop	First cell in a packet
	5:0	6	ingress_src_blade	Ingress source blade # (imported)
1	31:16	16	next_hop_id	Next hop IP address
	15:0	16	class_id	Class ID

7.3.4 Packet Header (added only to first c-frame)

Table 7-4 describes the packet header added to the first c-frame of every packet.

Table 7-4. CSIX Traffic Manager Header Added for First c-frame

LW	Bits	Size	Field
0	31:24	8	Reserved
	23:16	8	Header type
	15:0	16	Output port
1	31:0	32	Flow ID (QoS flow ID or MPLS label/flow ID)

7.3.5 Reference CSIX Base Header—One Shortword

Table 7-5 describes the format of the CSIX base header which is 16 bits in length.

Table 7-5. Reference CSIX Base Header - 1 Shortword

LW	Bits	Size	Field
0	15:14	2	Ready
	13:10	4	CSIX frame type
	9:9	1	CSIX reserved
	8:8	1	Private
	7:0	8	Payload length

7.3.6 Reference CSIX Unicast Extension Header

Table 7-6 shows the format of the CSIX unicast extension header.

Table 7-6. Reference CSIX Unicast Extension Header

LW	Bits	Size	Field
0	31:24	8	Class
	23:22	2	Private
	21:12	10	CSIX Reserved
	11:0	12	Destination

7.3.7 Statistics

The CSIX TX block maintains the following statistics on a per VOQ basis in debug mode only. The microblock maintains these as 32-bit counters. The XScale core component keeps a 64-bit version of these counters; for more information, refer to [Section 2.5, “Statistics and Handling of 64-bit Counters”](#) on page 61.

Table 7-7. CSIX Transmit Statistics

Counter	Offset from base for VOQ
Packets Transmitted	0x0
Bytes Transmitted	0x4
Cframes Transmitted	0x8
Reserved	0xc

7.4 Design

The CSIX transmit microblock receives transmit messages from the queue manager. Each transmit request contains the queue number and the buffer handle for the packet being transmitted (refer 3.9.5). For each queue, there is an associated segmentation state (Transmit Context or TXC). The TXC for different queues are stored in SRAM. 16 TXC are cached in local memory and the CAM is used to look them up. If the TXC for the current queue is not in local memory, then the LRU TXC is evicted to SRAM and the new TXC is read into local memory. The folding algorithm is used to optimize the caching of the TXCs.

The TX microblock uses the TXC to compute the offset into the packet in DRAM to read data from. It allocates a TBUF element and moves data from DRAM to the TBUF. It adds the CSIX interface prepend header (Traffic Manager Header) on to the cframe along with the packet data. Along with the per-frame header, an optional per-packet header may also be added to the first c-frame of a packet as it is sent across the fabric. The headers carry reassembly information for the CSIX RX block on the Egress IXP2400, sequence numbers for error detection and classification information (next hop ID, flow id, class id etc). The TX microblock then validates the TBUF element and uses the MSF to send the data to the fabric. When all the data in a buffer is transmitted, it frees the buffer by placing it onto a buffer free list.

The design of the CSIX TX block is logically composed of 4 phases, each of which is executed in strict order, which is ensured by inter-thread signaling. [Table 7-8](#) describes the 4 phases.

Table 7-8. Logical Phases of CSIX TX Block

Phase	Description
1	Issue read for the transmit request and wait for it to complete. Also wait for previous thread signal
2	Do a CAM lookup to check if the TXC is cached in local memory. If not, then evict the LRU TXC and read in the new TXC and metadata. If the TXC is cached, then check if the meta data is already read in, else read in the meta data. Wait for the I/O to complete and previous thread signal
3	Allocate a TBUF, transfer data from DRAM to TBUF. Add the prepend headers. Wait for I/O to complete and previous thread signal
4	Validate the TBUF for transmit. If all cframes in the buffer have been transmitted then free the buffer. Wait for I/O to complete and previous thread signal.

7.4.1 Interleaving Multiple Requests

To cover the DRAM latency, the CSIX TX block interleaves the handling of multiple requests. The four phases presented before can be combined to only two phases shown in [Table 7-9](#).

Table 7-9. Logical Phases of CSIX TX Block Interleaving for Multiple Requests

Phase	Description
1	Validate the TBUF for transmit request n-1. If cell count goes to 0, free the buffer. Request n-1 is complete Move data from DRAM to TBUF for transmit request n. Add prepend for request n Issue read for transmit request n+1 Wait for TBUF validate, prepend write and transmit request read to complete and signal from previous thread
2	Do CAM lookup for TXC associated with transmit request n+1. If lookup fails evict LRU TXC and read in new TXC from SRAM and meta data for this request. If lookup succeeds read the meta data if it was not already valid Wait for DRAM to TBUF for request n and TXC I/O for request n+1 to complete and signal from previous thread

One thing to note in [Table 7-9](#) is that the DRAM operation spans the two phases. It is issued in phase 1, but we only wait for it to complete at the end of phase 2.

7.4.2 Optimizing for the Minimum Packet Case

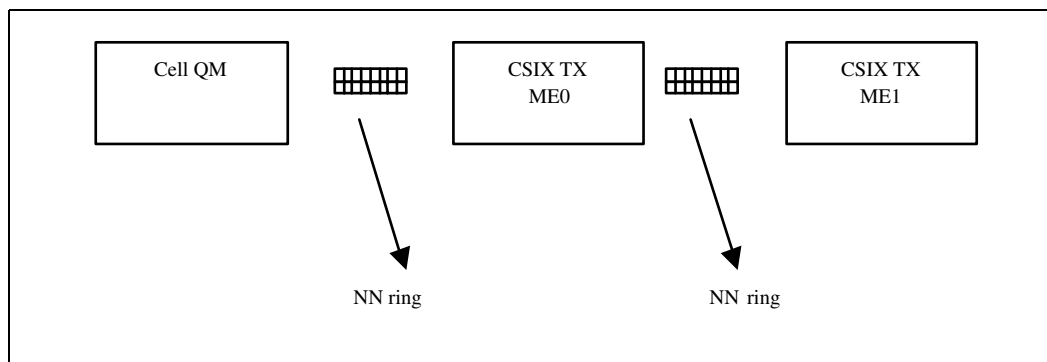
In the case where the entire packet fits into a single cframe, certain special optimizations are possible

- There is no need to read/update the TXC since no segmentation is required. This eliminates the need for a CAM lookup, read of TXC from SRAM etc.
- The buffer always needs to be freed (so checking for cell count being zero is not required)

This case is detected by checking the SOP, EOP bits (bit 30 and 31) in the buffer handle present in the transmit request.

7.4.3 Running the Microblock on Two Microengines

To support OC-192 POS data rates, the CSIX Transmit block can be conditionally compiled to run on two microengines in a context pipeline connected by a Next Neighbor ring. [Figure 7-1](#) CSIX TX microblock running on two microengines.

Figure 7-1. CSIX TX Microblock Running on Two Microengines

The first microengine:

- Reads the transmit request from the Next Neighbor ring.
- Reads the metadata of the buffer (containing this cell) from SRAM.
- Reads the transmit context from SRAM for the packet that this cell is a part of
- Calculates the prepend header for the outgoing CSIX cell.
- Calculates the buffer DRAM address for the cell.
- Writes a request to the next microengine that contains the buffer handle, VOQ#, prepend header, cell count remaining, DRAM address and SOP/EOP bit.

The second microengine:

- Reads the request from the next neighbor ring.
- Gets the next available TBUF element.
- Writes the cell prepend header to MSF.
- Launches the I/O request to read the cell from DRAM to TBUF.
- Validates the TBUF element written to.
- Frees the buffer if it is done processing the last cell in the buffer.

7.5 Flow Chart for Single Microengine Design

Figure 7-2 and Figure 7-3 show flowcharts for the two-phase CSIX Transmit.

Figure 7-2. CSIX Transmit: Flowchart for Phase 1 of Two-Phase Scheme

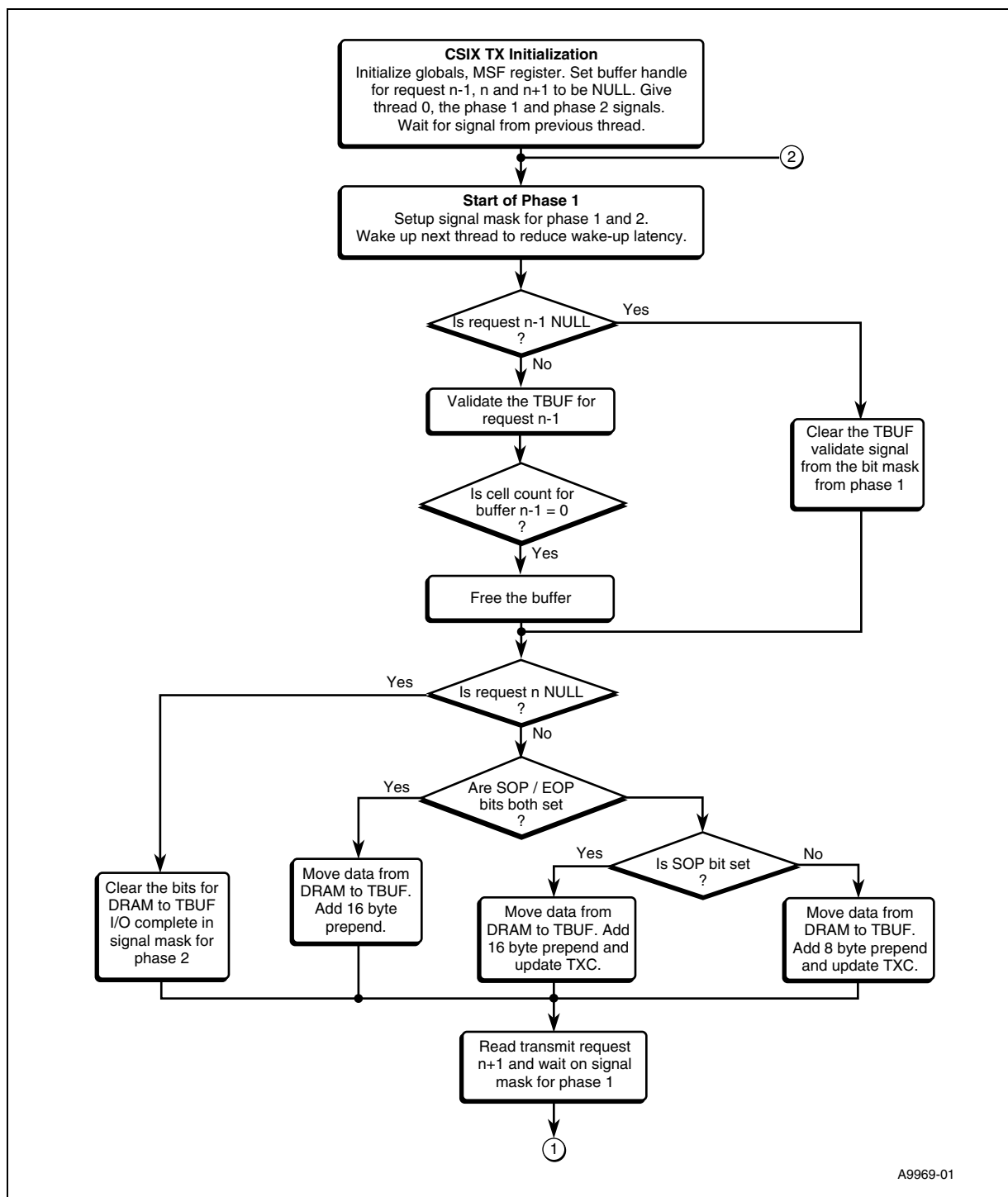
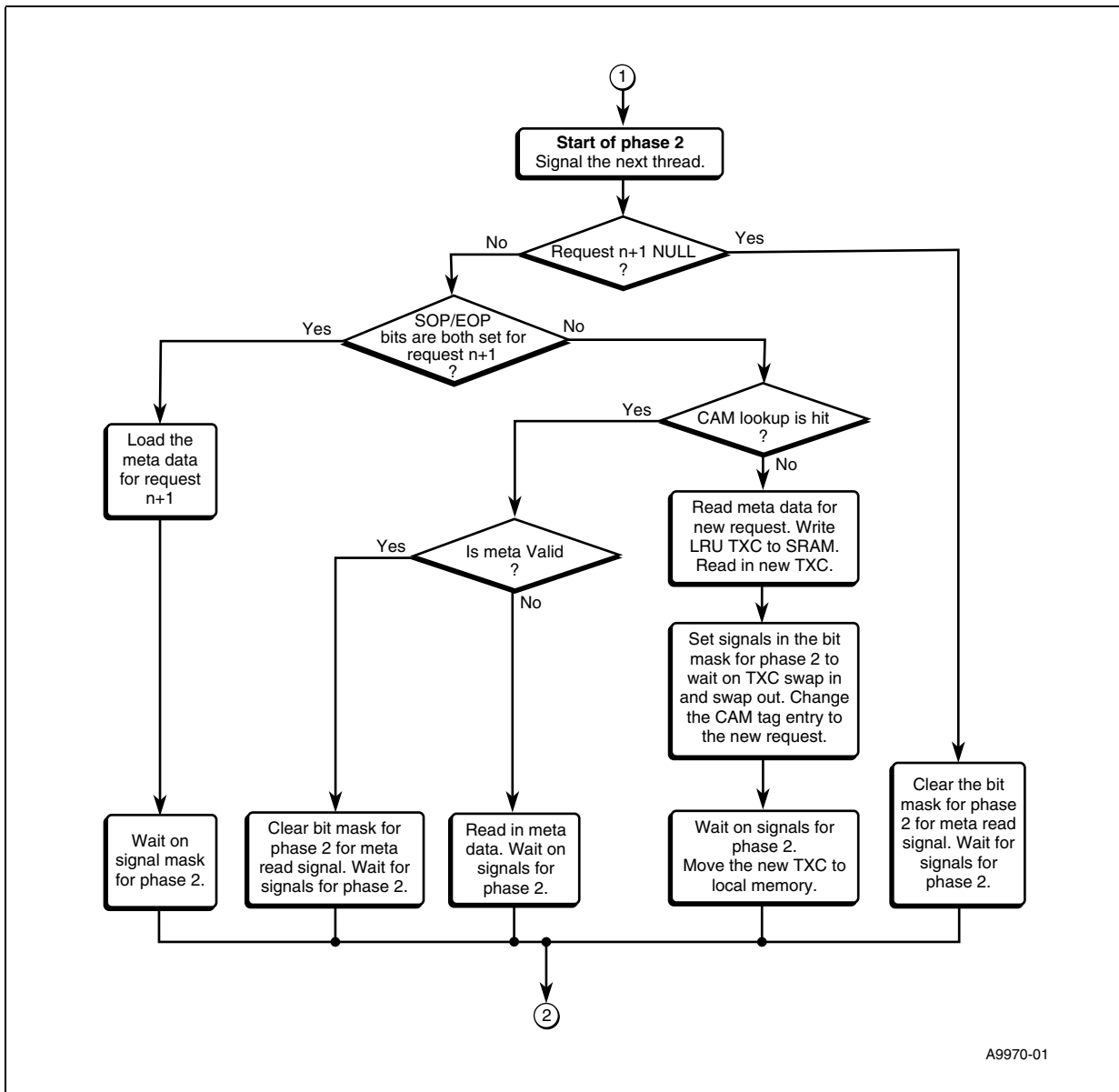


Figure 7-3. CSIX Transmit: Flowchart for Phase 2 of Two-Phase Scheme



7.6 Performance Analysis

Table 7-10 shows the cycle count in the critical path (min 49 byte POS packet) for the single microengine design.

Table 7-10. CSIX Transmit Performance Analysis: Worst Case Cycle Count

Worst Case Cycle count	IXP2400 budget for POS min packet at OC-48 rates	IXP2400 budget for Ethernet min packet at 4 Gbps rates	IXP2800 budget for Ethernet min packet at 10 Gbps rates
75	97	101	94

Table 7-10 shows that the CSIX TX microblock using the single microengine design can handle POS min packets at OC-48 data rates on the IXP2400, Ethernet min packets at 4 Gbps data rates on the IXP2400, and Ethernet min packets at 10 Gbps data rates on the IXP2800.

Table 7-11 shows the instruction cycle counts in the critical path (min 49 byte POS packet) for the two microengine design which is used in the OC-192 POS application. The table shows that the microblock meets the performance requirements for OC-192 POS with the two microengine design.

Table 7-11. Instruction Cycle Counts in the Critical Path

Microengine	Worst Case Cycle count	IXP2800 budget for POS min packet at OC-192 rates
Microengine 1	54	57
Microengine 2	53	57

Since the available cycle count for the min packet case is 97 cycles, the CSIX TX microblock meets the performance requirements in terms of cycle count.

Table 7-12 shows the I/O operations performed in the different phases of the CSIX TX microblock. Each thread has an available I/O latency of 97×8 cycles to handle a minimum POS packet.

Table 7-12. CSIX Transmit Performance Analysis: I/O Operations

I/O Operations	Phase
Scratch read of 2 words	1
Write LRU TXC to SRAM – 4 words	2
Read new TXC from SRAM – 4 words	2
Read meta data from SRAM – 6 words	2
Write from DRAM to TBUF – 40 bytes for minimum packet	1
MSF write into TBUF of prepend – up to 16 bytes for minimum packet	1
MSF write to validate TBUF and start transmit	1
Enqueue to free buffer	1

7.6.1 Characterization Data

Table 7-13. CSIX Tx Microblock Characterization Data

Data	Value
General:	
Microblock Name	csix tx
Microblock Version Number	1.0
Implementation Language	microcode
Configuration Options use to gather this set of data	1. WAIT_FOR_COMMON_RESOURCE_INITIALIZATION 2. INTERLEAVING_PHASES 3. MULTICAST_SUPPORT_ENABLE 4. FREELIST_MANAGER 5. _HARDWARE_DEBUG_ 6. IS_IXPTYPE(__IXP28XX) 7. COUNTERS
Measurement Environment (tool settings)	SDK 3.5
Performance: (cycle counts, latencies, bandwidths)	
Instruction cycle count per packet (common case packet / worst case cycle count)	One microengine: 75 Two microengines: 107
Common-case packet/path assumptions to be documented here	
Scratch Memory	
# of longwords read (for bandwidth calculations)	2
# of longwords written (for bandwidth calculations)	0
# and type of each atomic operation performed (for bandwidth calculations)	0
SRAM	

Table 7-13. CSIX Tx Microblock Characterization Data (Continued)

Data	Value
# of longwords read	10
# of longwords written	4
# and type of each atomic operation performed (for bandwidth calculations)	0
DRAM	
# of quadwords read	5
# of quadwords written	0
Other I/O instructions (i.e. MSF, PCI, CAP, and Hash)	msf[write], 1 sram[enq]
List of dependent I/O accesses in the longest latency path	

Per-Microengine Resources:

Control-Store Usage (# of instructions used)	One microengine: 281, Two microengines: 350
Local Memory Footprint (# of long words used)	64
Local Memory Configuration (shared, or per-context pointer)	shared and per-context
Local Memory - # of LM pointers used	2
GPR Usage – minimum, static usage (absolute, static, globals)	
Transfer Reg. Usage – minimum, static usage	
Next Neighbor Reg. Usage – minimum, static usage	
Signal Usage – minimum, static usage	
CAM used? (yes or no)	Yes

Global Resources:

Scratch footprint (# of longwords used) – constant or formula ...	0
SRAM footprint (# of longwords used) – constant or formula ...	CSIX_TX_CTX_SIZE*CSIX_TX_CTX_TOTAL
DRAM footprint (# of quadwords used) – constant or formula ...	None
Q-Array usage - # of queues used and if they need to be cached	None
CRC Unit used?	no
Hash Unit used? (yes or no)	no

MSF Usage Information:

Media Bus Configuration	N/A
RBUF, TBUF usage	128-byte TBUF element
CBus signals	

Table 7-13. CSIX Tx Microblock Characterization Data (Continued)

Data	Value
Other Information:	
Critical Section Length (compute cycles + memory accesses)	
# of phases	One microengine: 4, Two microengines: 2, 2
Packet Metadata - fields read	
Packet Metadata - fields written	
Header - fields read	
Header - fields written	
Documentation:	
Thread Ordering Requirements	Yes
OS dependencies	VxWorks, Linux
Chip/hardware dependencies (i.e. Crypto Unit in 2850)	IXP2400, IXP2800
Tested on which SDK Release(s)	IXA SDK 3.5
Tested on hardware? Which hardware configuration?	POS, ETH, ATM
Tested in which applications (not an all inclusive list)	Almost all Ingress apps, e.g. OC48 POS, OC192 POS
Possible Configuration Options	
Major limitations (i.e. Rx/Tx microblocks written to use entire ME)	
Packet Sequencing Issues (esp. in POT applications)	Packets received must be in sequence
Core Component or Interface requirements or dependencies	

This chapter describes the low-level design and implementation of the Packet Transmit microblock for the following modes—SPHY 1x32, SPHY 4x8, and MPHY-4.

8.1 Overview

The IXP2400 offers three modes for POS transmit—SPHY, MPHY-4, and MPHY-16. In the SPHY mode the 32-bit channel may be divided into four 8-bit channels—for example to implement a quad OC-12 configuration.

Table 8-1 lists the possible solutions based on these three modes:

Table 8-1. Possible Solutions Based on Three Packet Transmit Modes

MSF mode	Type	Ports supported	ME	Head of line blocking (HOLB) issue
SPHY 1x32	1xOC48	1 port	1 ME	No
SPHY 4x8	4xOC12	4 ports	1 or 2 ME's (2 or 4 threads per port)	No
MPHY-4	4xOC12	4 ports	1 or 2 ME's (2 or 4 threads per port)	No
MPHY-16	2xOC24 to 16xOC3	Up to 16 virtual ports	2 ME's	Yes

In SPHY 1x32 mode, there are no head of line blocking issues since there is a single port. In MPHY-4 and SPHY 4x8 modes as well, head of line blocking issues may be avoided by assigning two threads per port and a separate scratch ring for each port for transmit requests from the QM. In the MPHY-16 mode, head of line blocking issues cannot be avoided and need to be explicitly addressed in the design. This makes the overall algorithm much more complicated as compared to the other two cases. Therefore the focus is on the SPHY and MPHY-4 design cases first. [Chapter 9, “Packet TX for MPHY-16”](#) describes the design for an MPHY-16 configuration.

This section describes the design for a single port OC-48 configuration. The design for quad OC-12 is essentially identical to the single port design except that 2 (or 4) threads are assigned to each port and a separate scratch ring is allocated per port. The hardware splits the available TBUFs into four pools—one for each port.

The Packet Transmit block runs on a single microengine and receives transmit requests from the queue manager. Since the transmit requests are packet based, the microblock needs to segment the payload into mpackets, copy them into TBUFs and validate them for the MSF to transmit. Since packets cannot be interleaved, an entire packet needs to be segmented and transmitted before the next packet can be processed. This means that every time a thread is ready to process an mpacket, it needs to check if there is a current packet being segmented. If there is, then it needs to transmit one TBUF for that packet. If there is no current packet, then it needs to get the next packet transmit request and start a segmentation context for that packet.

To reduce the latency of getting the next transmit request from scratch, the design keeps a queue of transmit requests in local memory along with associated segmentation context. Each port has its own queue in local memory and the depth of the queue currently is sixteen entries. To avoid overflowing this queue, the TX block checks if there is room for one more packet in the queue before reading from the scratch ring. The Transmit block also keeps track of the number of packets transmitted and uses the reflector write mechanism to tell the scheduler about how many packets have been transmitted. The scheduler uses this information to keep track of how many packets are in flight. If this exceeds a configurable threshold, it stops scheduling on that port. If the Transmit block receives a large packet, it can take a long time to segment and transmit it. The packets in flight mechanism prevents the scheduler from overflowing the transmit queue in this case.

Since a large packet may span several buffers (in a link list), the microblock caches packet descriptor information on two buffers per packet at any time. One is for the current buffer being processed and the other is the next buffer in the list. When the processing of the current buffer is complete, the next buffer becomes the current buffer. Pre-fetching the descriptor information allows us to cover the latency associated with reading the descriptor from memory. The descriptor information on the current and the next buffer are stored in the transmit context for the packet in local memory.

8.2 Assumptions and Dependencies

This microblock needs to handle the following two restrictions imposed by the IXP2400 hardware:

- The `DRAM[tbuf_wr, ...]` instruction always accesses DRAM on an eight-byte or quad-word boundary
- The sum of the prepend length and payload length must be an integral multiple of the bus width], except for a TBUF in which EOP is set. In SPHY 1x32, MPHY-16 and MPHY-4 mode, the bus width is 32 bits—that is, four bytes. In SPHY 4x8 mode, the bus width is eight bits.

8.3 Data Structures

This section describes the data structures used for the packet transmit microblock.

8.3.1 Context-Relative Thread Execution Status Flag

The GPR below store various execution flags for a thread.

Table 8-2. Context-Relative Thread Execution Status Flag—`exe_stat_flag`

LW	Bits	Size	Field	Description
0	31:7	25		Reserved
	6:6	1	get_tx_request_flag	Flag to check tx_request, since scratch ring has been read
	5:5	1	eob_mpkt_flag	Flag for free buffer
	4:4	1	skip_transmit_flag	Flag to skip transmit a tbuf, due to no payload and no prepend allowed due to bus width restriction
	3:3	1	read_sop_bd_falg	Falg to save sop meta data

Table 8-2. Context-Relative Thread Execution Status Flag—exe_stat_flag (Continued)

LW	Bits	Size	Field	Description
	2:2	1	save_leftov_to_lm_flag	Flag to save leftover to local memory
	1:1	1	leftov_from_pb_flag	Flag to transmit leftover payload from previous buffer
	0:0	1	read_snd_bd_flag	Flag to save secondary buffer meta data

8.3.2 TX Request—One Longword

Table 8-3 describes the transmit request received from the Queue Manager.

Table 8-3. TX Request - One Longword

LW	Bits	Size	Field	Description
0	31:31	1		Valid bit
	30:28	3		Reserved
	27:24	4		Port number
	23:0	24		SOP meta handle >> 2

8.3.3 Queue Structure for Each Port

Queue information for each port is located in two different locations to provide the flexibility to increase the queue size for each port and to use both local memory pointers LM_ACTIVE_0 and LM_ACTIVE_1 effectively.

The first 256 long words in local memory are used to store the queue entries for all ports. Each queue entry is four long words. So if there are four ports on a microengine, then for each port we can queue up to 16 transmit requests. The upper 128 words are used for the queue descriptors—that is, head, tail, count, and so on. Each queue descriptor is 32 words. Only 15 words are currently used

8.3.3.1 Queue Entry Structure

Table 8-4 describes the structure of a single entry in the queue.

Table 8-4. Queue Entry Structure

LW	Bits	Size	Field	Description
0	31:31	1	p_sop_flag	SOP bit for the packet in queue entry 0
	31:30	1	ab_rd_nbd_flag	Read next buffer meta data bit for the active buffer
	29:28	2		Reserved
	27:24	4	ab_flist	Freelist of the active buffer
	23:0	24	ab_bd	Buffer descriptor (BD) of the active buffer
1	31:16	16	ab_paylo_rmnd	Remaining bytes of the payload to be sent out in the active buffer
	15:0	16	ab_offset_rpaylo	Offset of the first byte in the remaining payload in the active buffer

Table 8-4. Queue Entry Structure

LW	Bits	Size	Field	Description
2	31:31	1		Reserved
	30:30	1	sb_rd_nbd_flag	Read next buffer meta data bit of the secondary buffer
	29:28	2		Reserved
	27:24	4	sb_flist	Freelist of the secondary buffer
	23:0	24	sb_bd	Buffer descriptor (BD) of the secondary buffer
3	31:16	16	sb_buf_size	Buffer size of the secondary buffer
	15:0	16	sb_offset	Offset of the secondary buffer

8.3.3.2 Queue Descriptor Structure

Table 8-5 describes the structure for a queue descriptor. There is one queue descriptor per port.

Table 8-5. Queue Descriptor Structure

LW	Bits	Size	Field	Description
0	31:0	32	port_id	Port number
1	31:0	32	tcw_vd_base	base to compute TBUF_ELEMENT_CTRL_V# address for the specific port
2	31:0	32	tbuf_base	base to compute tbuf address for the specific port
3	31:0	32	queue_entry_info_base	Base local memory address of queue entry info for the specific port
4	31:0	32	head_offset	Queue head for this port (byte offset within queue info)
5	31:0	32	tail_offset	Queue tail for this port (byte offset within queue info)
6	31:1	31		Reserved
	0:0	1	leftover_flag	If set, means that there are some leftover bytes from previous buffer to be transmitted in the beginning of the mpkt for this port
12	31:8	24		Reserved
	7:0	8	lp_len	The length of leftover in bytes, less than 4 bytes
13	31:0	32	lp_lw0	Longword of payload leftover by previous buffer
14	31:0	32	snd_next_bd	Next buffer descriptor for secondary buffer for the specific port
15..31	31:0	32	Reserved	Reserved

8.3.4 Output Transmit Control Word—Two Longwords

Table 8-6 describes the format of the output transmit control word.

Table 8-6. Output Transmit Control Word - Two Longwords

LW	Bits	Size	Field	Description
0	31:24	8	payload_len	Payload length in bytes
	23:21	3	payload_offset	Prepend offset in bytes
	20:16	5	prepend_len	Prepend length in bytes
	15:13	3	prepend_offset	Payload offset in bytes
	12:12	1		Reserved
	11:11	1	skip	Skip
	10:10	1	err	ERR
	9:9	1	sop	SOP
	8:8	1	eop	EOP
	7:4	4		Reserved
	3:0	4	channel	Channel (always 0)
1	31:0	32		Reserved

8.3.5 Statistics

The Packet TX block maintains the following statistics in SRAM on a per port basis. The microblock maintains these as 32-bit counters and the Intel XScale® core component keeps a 64-bit version of these counters (see [Section 2.5, “Statistics and Handling of 64-bit Counters”](#) on page 63).

8.4 Build Switches

Table 8-7 describes the compile time options used in the CSIX Receive microblock.

Table 8-7. Compile Time Options Used in the CSIX Receive Microblock

Symbol	Description
SCHEDULER_ME	Set this to the microengine number of the scheduler. This is required for the reflect write operation that informs the scheduler of the number of packets transmitted. E.g. SCHEDULER_ME=2 implies that the second microengine is being used for the scheduler.
DISABLE_TX2SCHED_FEEDBACK	Disable the reflect write feedback to the scheduler
MICROC_SCHEDULER	Turn on if the scheduler is written in microC. The naming convention for the reflect write registers is different
TX_PHY_MODE	This may be set to SPHY_1_32, SPHY_4_8, or MPHY_4

Table 8-7. Compile Time Options Used in the CSIX Receive Microblock (Continued)

Symbol	Description
THIS_ME	In the SPHY_4_8 or MPHY_4 case, the Packet TX microblock may be run on 1 microengine (2 threads per port) or 2 microengines (4 threads per port)THIS_ME = ONLY_ONE_ME_FOR_PACKET_TXTHIS_ME = FIRST_ME_FOR_PACKET_TXTHIS_ME = SECOND_ME_FOR_PACKET_TX

8.5 Design

The Packet TX block runs on a single microengine. It dequeues transmit requests from a scratch ring and queues them in local memory—one scratch ring and one local memory queue per port. Each request is associated with a packet which needs to be segmented into TBUFs and transmitted. Unlike the CSIX TX block which is cell based, the Packet TX is a packet-based transmit block and receives only one transmit request per packet.

For each port, the Packet TX block caches information from the meta data for the packet at the head of the queue. For each packet, information from the meta data of the current buffer being handled (active buffer) and the meta data of the next buffer in the buffer chain—that is, the secondary buffer—are cached. The meta data for the next buffer is read in while the current buffer is being processed.

A restriction that the design needs to deal with is that the sum of the prepend length and payload length must be an integral multiple of the bus width, except for a TBUF in which EOP is set. In SPHY 1x32 mode, the bus width is 32 bits—or four bytes. To comply with this restriction, a maximum of three bytes at the end of a buffer may be saved in local memory and transmitted with the payload of the next TBUF for the packet. If at the end of a buffer, less than four bytes are left, then no TBUF is transmitted in that slot. Instead the bytes are saved in local memory and the SKIP bit is set in the TBUF allocated for that slot.

Flags in the transmit context are maintained to check for the case where the meta information for the next buffer has not been read in or if the leftover bytes are not ready to be transmitted.

8.6 Flow Chart

SPHY Packet Transmit operations are described in this section using a flow chart, which is divided into the following six figures.

Figure 8-1. SPHY Packet TX Flow Chart: Part 1 of 6

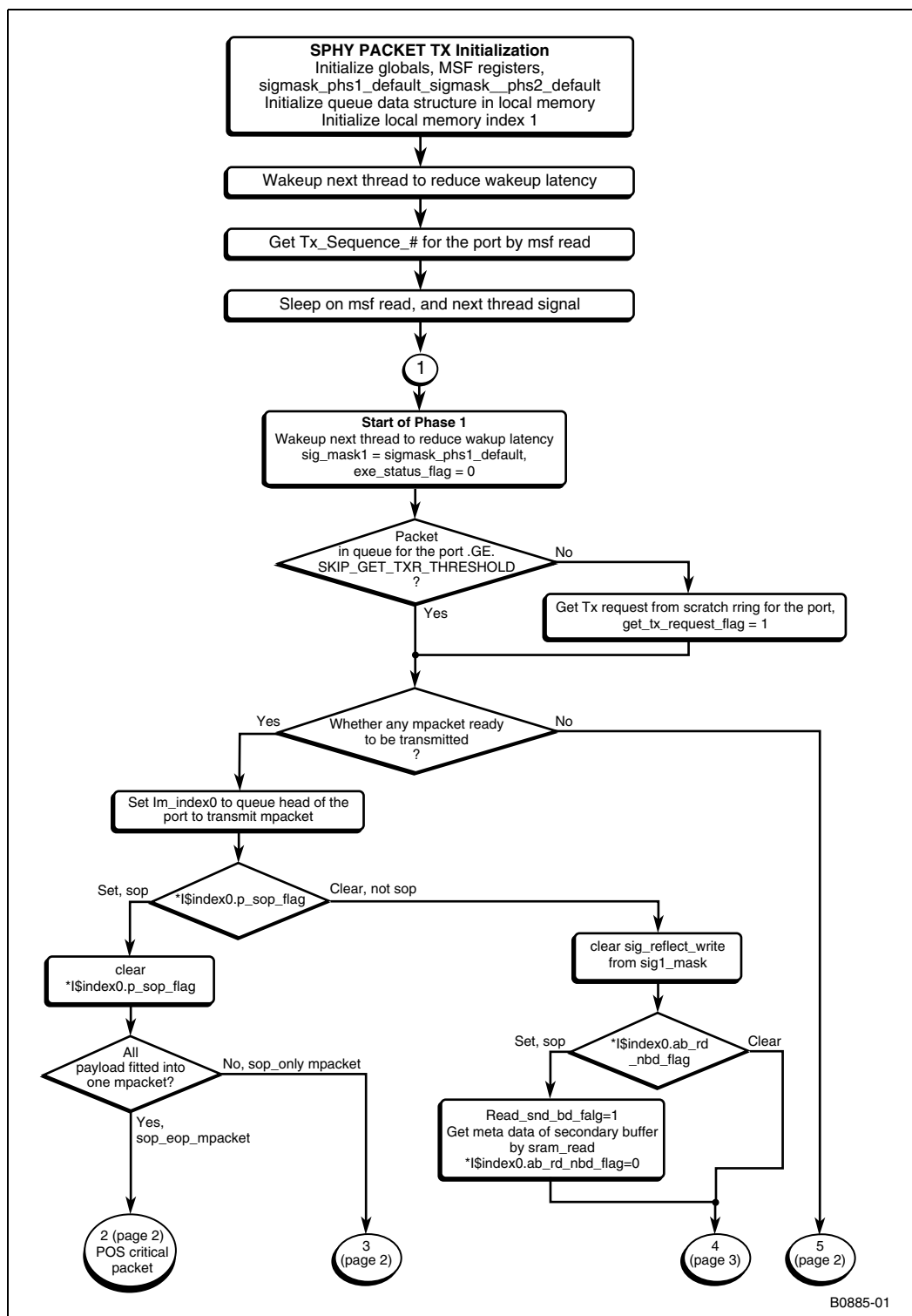


Figure 8-2. SPHY Packet TX Flow Chart: Part 2 of 6

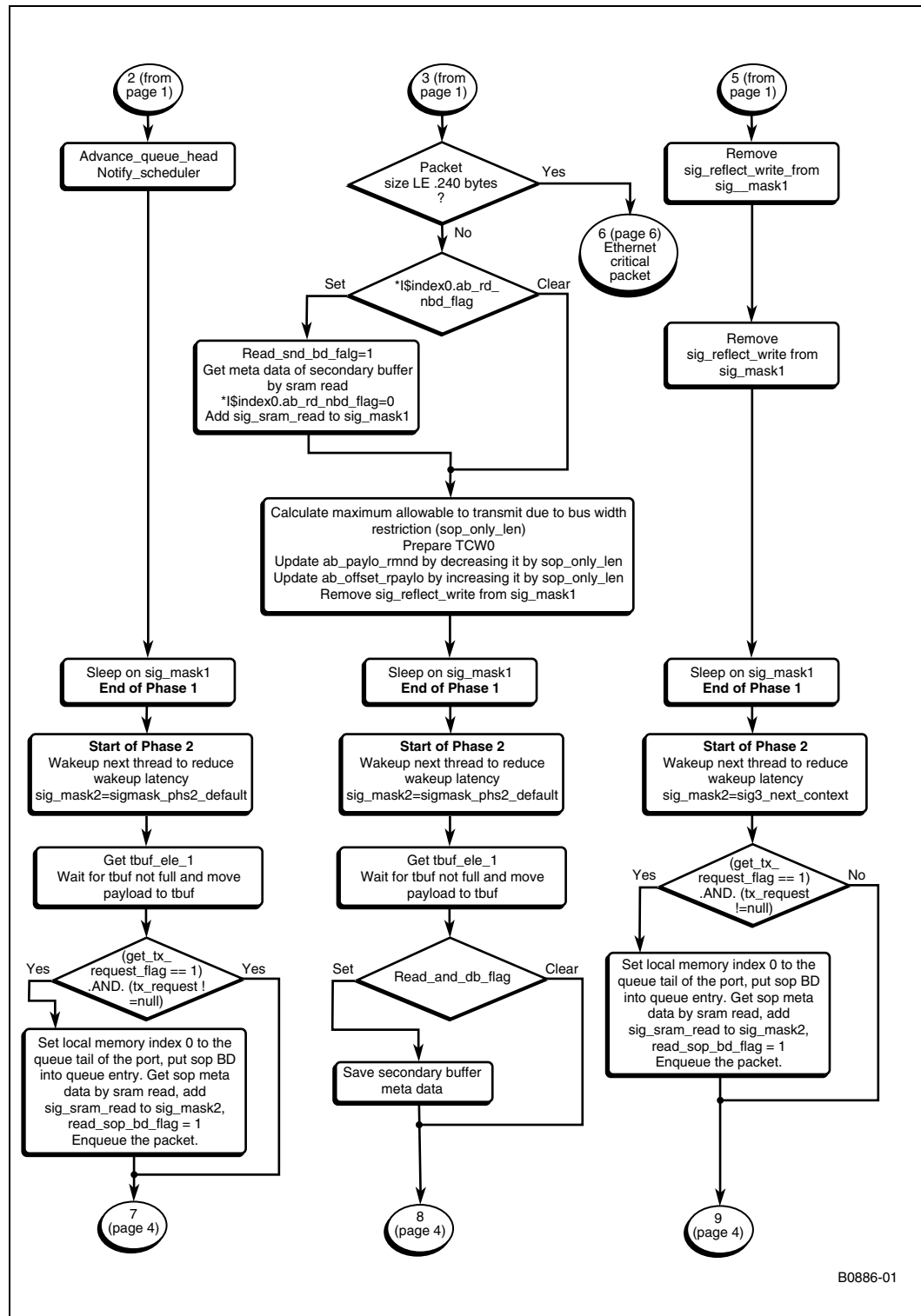
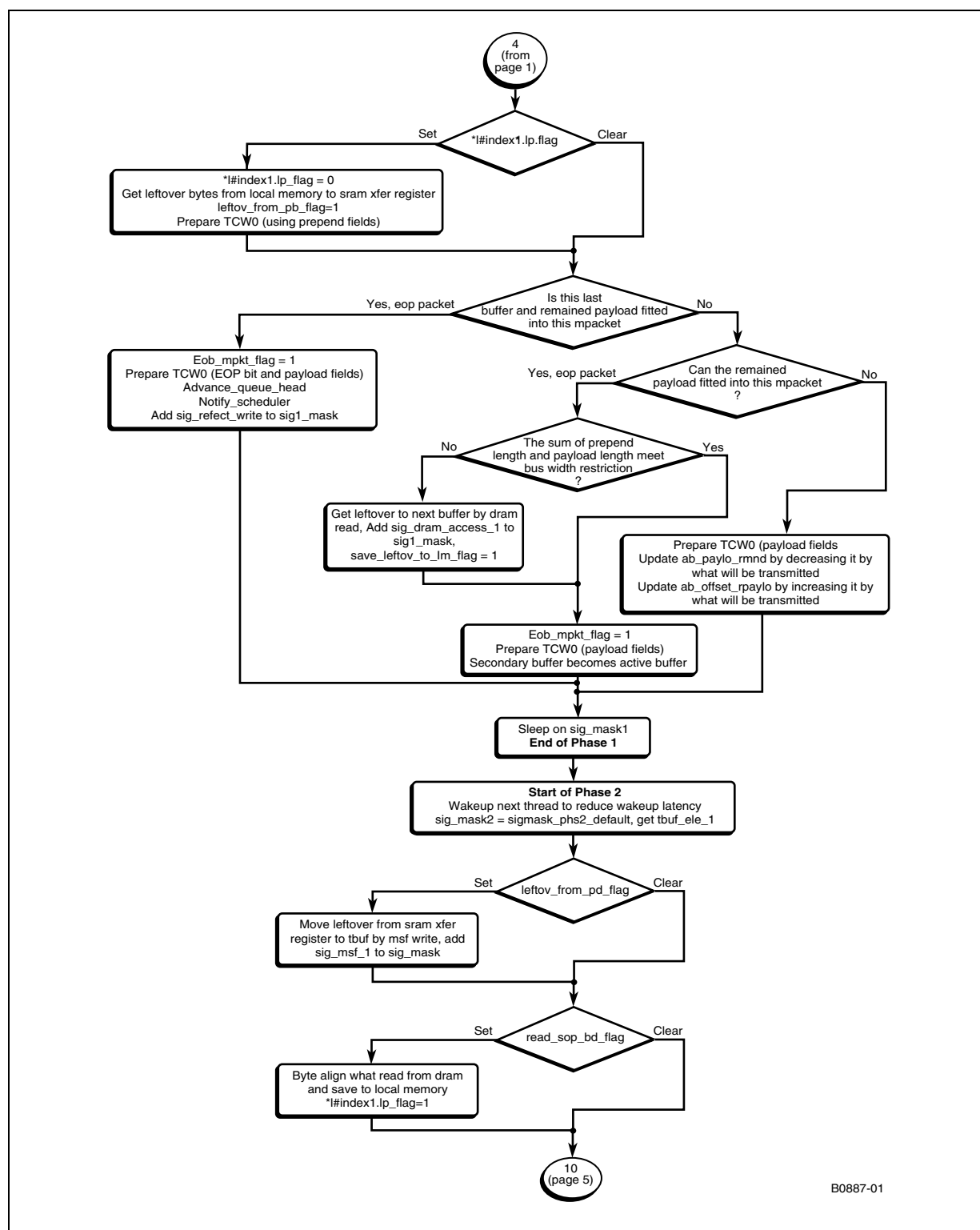


Figure 8-3. SPHY Packet TX Flow Chart: Part 3 of 6



B0887-01

Figure 8-4. SPHY Packet TX Flow Chart: Part 4 of 6

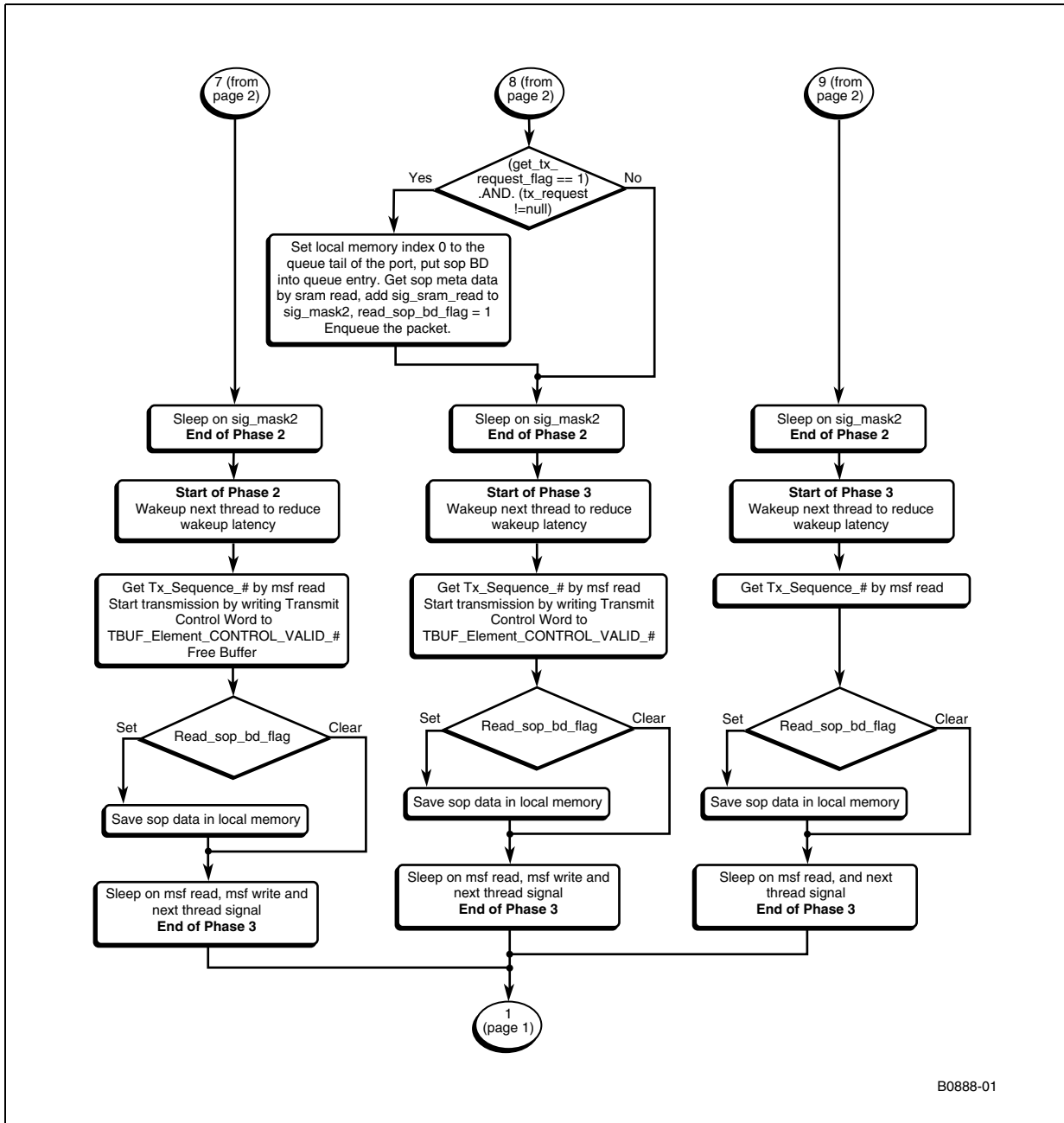


Figure 8-5. SPHY Packet TX Flow Chart: Part 5 of 6

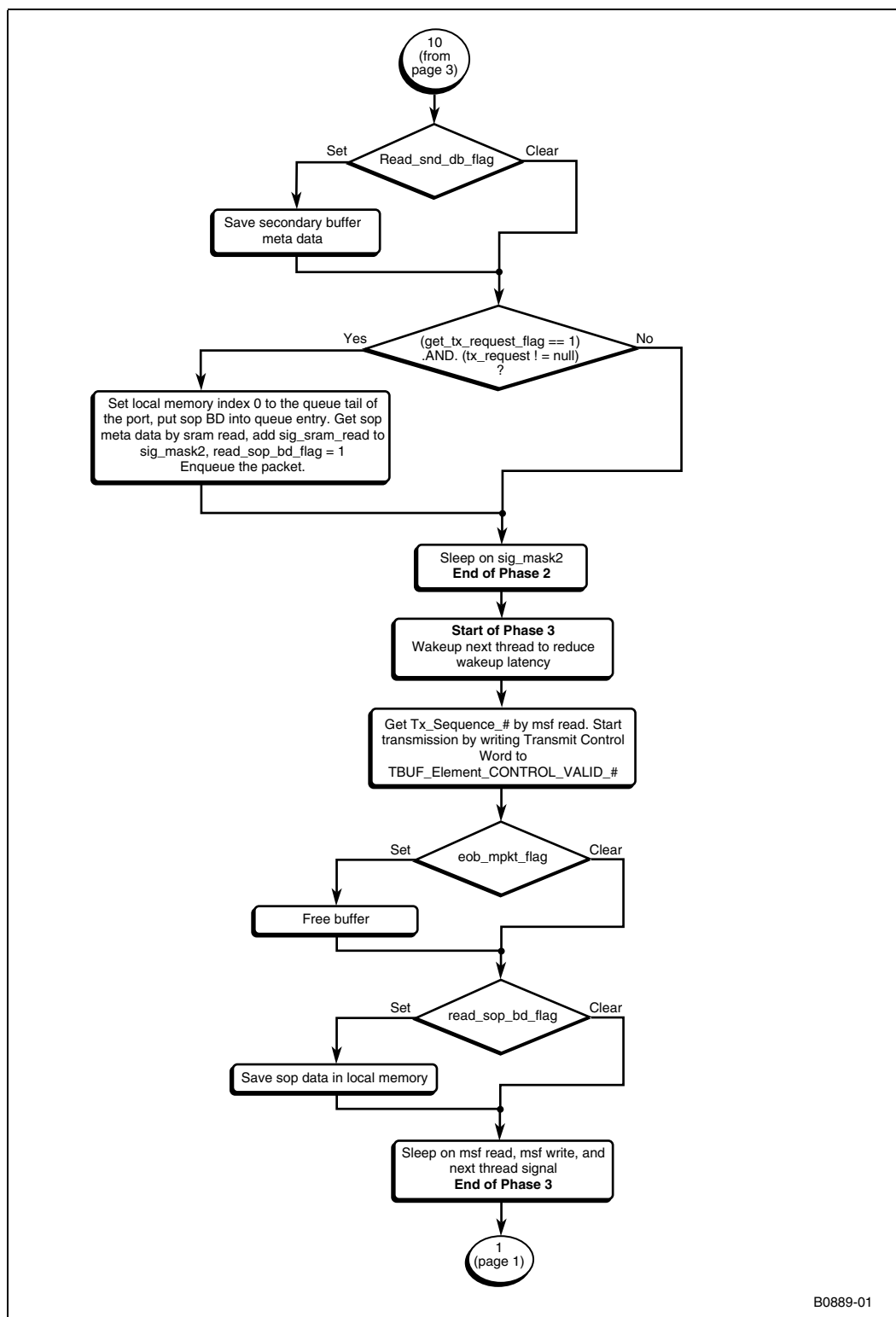
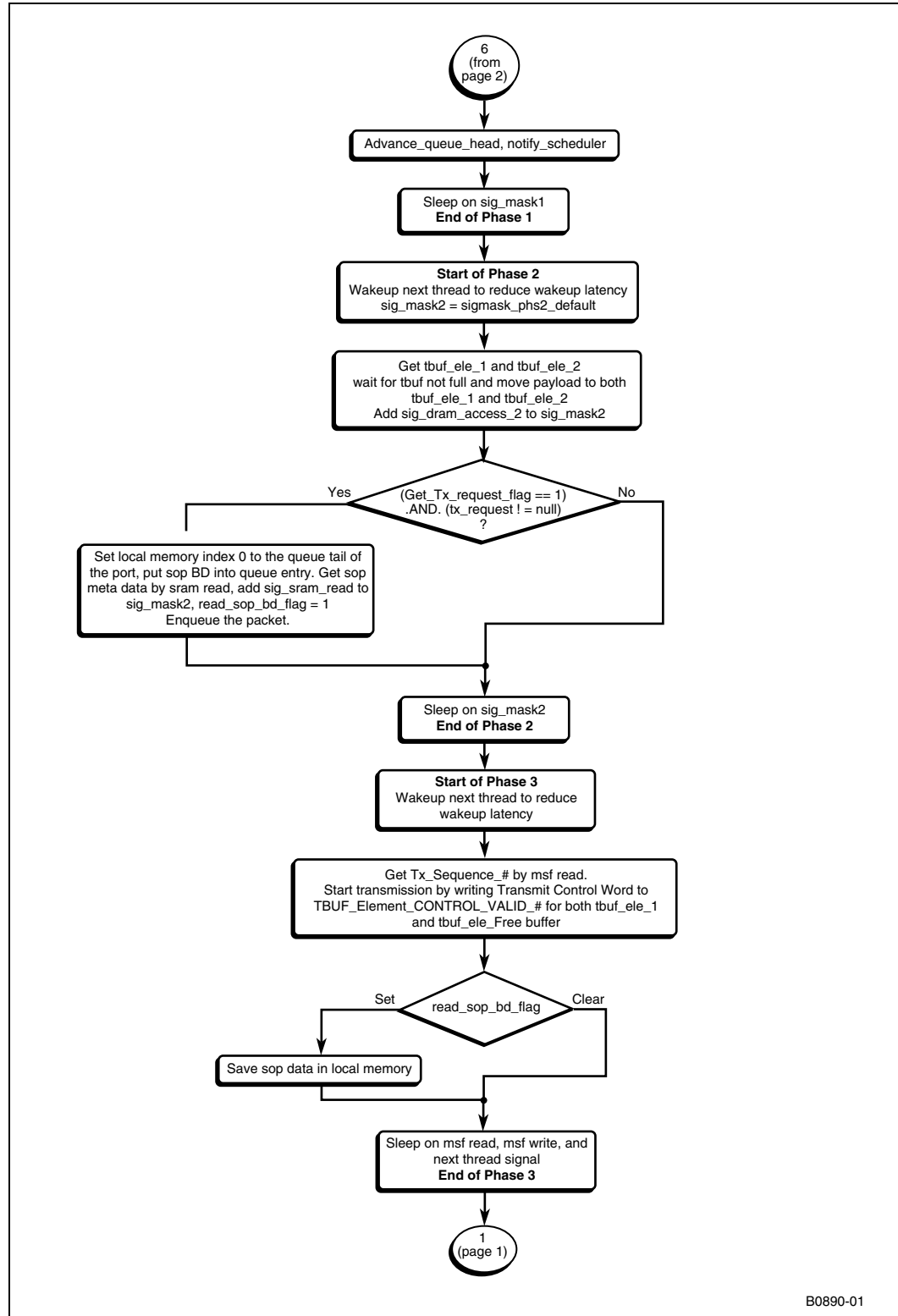


Figure 8-6. SPHY Packet TX Flow Chart: Part 6 of 6



B0890-01

8.7 Performance Analysis

The Packet TX microblock uses one microengine to handle up to OC-48 rates of traffic. This implies that the available cycle count budget to handle a POS minimum packet is 97 cycles. The cycle count budget to handle an Ethernet minimum size packet at four Gbps is 100 cycles. The table below shows the instruction estimate for the TX microblock to handle a minimum packet in the worst-case code path. As shown, the Packet TX microblock needs to be optimized to meet line rate.

Table 8-8. Packet TX Performance Analysis - Worst Case Cycle Count

Component	Worst Case Cycle count
Phase 1	43
Phase 2	40
Phase 3	13
Total	96 (Available budget is 97)

The table below shows the I/O operations performed in the different phases of the TX microblock for the worst case. Each thread has an available I/O latency of 97*8 cycles to handle a minimum POS packet and a latency of 100*8 for a minimum Ethernet packet at four Gbps.

Table 8-9. Packet TX Performance Analysis - Worst Case I/O Operations

I/O operations	Phase
Scratch read of transmit request (one long word)	1
MSF read of one long word (Tx_Sequence_x)	1
DRAM read of leftover (16 bytes)	1
SRAM read of 3 long words for secondary buffer meta data	1
Reflect write of one long word to notify scheduler	1
Dram to tbuf write of up to 128 bytes	2
SRAM read of 3 long words for new packet's SOP meta data	2
MSF write to validate TBUF and start transmit (2 long words)	3
Enqueue operation to free buffer	3

8.7.1 Characterization Data

Table 8-10. Packet TX for SPHY and MPHY-4 Microblock Characterization Data

Data	Value
General:	
Microblock Name	SPHY_MPHY4_TX
Microblock Version Number	1.0
Implementation Language	microcode

Table 8-10. Packet TX for SPHY and MPHY-4 Microblock Characterization Data (Continued)

Data	Value
Configuration Options use to gather this set of data	1. SCHEDULER_ME=0x02 2. THIS_ME=PACKET_TX_FIRST_ME 3. TX_PHY_MODE=SPHY_4_8 4. ADD_L2_HEADER 5. ETHERNET_TX
Measurement Environment (tool settings)	
Performance: (cycle counts, latencies, bandwidths)	
Instruction cycle count per packet (common case packet / worst case cycle count)	92 (available budget is 97)
Common-case packet/path assumptions to be documented here	Assumptions: <ul style="list-style-type: none"> • Tx request first saved in local memory and retrieved to TX in order for each port. • Scheduler will schedule TX requests based on how many packets has been transmitted out of MSF to prevent overflowing of TX requests in local memory queue. • This microblock can be configured to run in one or two ME. • All numbers reported are worst-case for for 4 Gbs Ethernet normal path.
Scratch Memory	
# of longwords read (for bandwidth calculations)	1
# of longwords written (for bandwidth calculations)	0
# and type of each atomic operation performed (for bandwidth calculations)	0
SRAM	
# of longwords read	Meta data: 5, L2Table: 4
# of longwords written	0
# and type of each atomic operation performed (for bandwidth calculations)	0
DRAM	
# of quadwords read	9
# of quadwords written	0
Other I/O instructions (i.e. MSF, PCI, CAP, and Hash)	4
List of dependent I/O accesses in the longest latency path	
Per-Microengine Resources:	
Control-Store Usage (# of instructions used)	1220
Local Memory Footprint (# of long words used)	196
Local Memory Configuration (shared, or per-context pointer)	per-context pointer

Table 8-10. Packet TX for SPHY and MPHY-4 Microblock Characterization Data (Continued)

Data	Value
Local Memory - # of LM pointers used	2
GPR Usage – minimum, static usage (absolute, static, globals)	
Transfer Reg. Usage – minimum, static usage	
Next Neighbor Reg. Usage – minimum, static usage	0
Signal Usage – minimum, static usage	0
CAM used? (yes or no)	No

Global Resources:

Scratch footprint (# of longwords used) – constant or formula ...	0
SRAM footprint (# of longwords used) – constant or formula ...	0
DRAM footprint (# of quadwords used) – constant or formula ...	0
Q-Array usage - # of queues used and if they need to be cached	0
CRC Unit used?	no
Hash Unit used? (yes or no)	no

MSF Usage Information:

Media Bus Configuration	SPHY/MPHY4 POS TX
RBUF, TBUF usage	TBUF
CBus signals	-

Other Information:

Critical Section Length (compute cycles + memory accesses)	0
# of phases	2
Packet Metadata - fields read	bfferSize, payloadOffset, nextBufferHandle, nextHopId
Packet Metadata - fields written	None
Header - fields read	None
Header - fields written	None

Documentation:

Thread Ordering Requirements	
OS dependencies	VxWorks, Linux
Chip/hardware dependencies (i.e. Crypto Unit in 2850)	2400

Table 8-10. Packet TX for SPHY and MPHY-4 Microblock Characterization Data (Continued)

Data	Value
Tested on which SDK Release(s)	IXA SDK 3.5
Tested on hardware? Which hardware configuration?	Yes, IXDP 2400
Tested in which applications (not an all inclusive list)	
Possible Configuration Options	
Major limitations (i.e. Rx/Tx microblocks written to use entire ME)	None
Packet Sequencing Issues (esp. in POT applications)	None
Core Component or Interface requirements or dependencies	ATM/POS TX and Ethernet TX core components

9.1 Overview

In the MPHY-16 mode, head of line blocking (HOLB) issues cannot be avoided and need to be explicitly addressed in the design. This makes the overall algorithm much more complicated as compared to the SPHY and MPHY-4 cases.

The Packet TX microblock for MPHY-16 runs on two microengines and handles up to 16 virtual ports. Each microengine handles some subset of the 16 ports—configured via a bit mask—and OC-24 data rates. Each microengine has a separate scratch ring on which it receives transmit requests from the QM. The data rates on the 16 ports do not have to be equal and can be adjusted based on the weights in the WRR scheduler.

In MPHY-16 mode, the TBUF elements function as a single large segment. All traffic for the sixteen ports funnel into this single segment—partitioning the TBUF element into sixteen equal sized segments would potentially throttle ports that carry higher bandwidth. The Tx_MPHY_Status MSF register in the IXP2400 is designed to provide PHY status to help the software to schedule the transmission of packets to avoid HOLB. For each port, there are two bits Tx_Pending and Tx_Status in the Tx_MPHY_Status register. When Tx_Pending=1, it indicates that there is pending transmit activity for that port, then Tx_Status is indeterminate. When Tx_Pending=0 and Tx_Status=1, the port is ready to accept at least n mpackets of data. This size of n —referred to as PHY_DEPTH in this document—depends on the characteristics of the port PHY and size of the mpacket.

Based on polling the Tx_MPHY_Status register and the n value for each port, a flow control vector is formed. This flow control vector and a round robin vector between ports is used to schedule the transmission of packets.

Each microengine receives transmit requests from the Queue Manager through a separate scratch ring. Since the TX requests are packet based, the microblock needs to segment the payload into mpackets, copy them into TBUFs and send them over the fabric interface. Since a long packet may span several buffers, the microblock caches information on two buffers per packet at any time. One is the current buffer being processed and the other is the next buffer in the chain.

To prevent head of line blocking, transmit requests are queued up in local memory—one local memory queue per port. The Packet Transmit block informs the scheduler of how many packets are transmitted on a port via a reflector write. The scheduler keeps track of packets in flight and ensures that the number of packets scheduled but not transmitted does not exceed the local memory storage in the transmit microengine.

9.2 Assumptions and Dependencies

This microblock needs to handle the following two restrictions imposed by the IXP2400 hardware:

- The `DRAM[tbuf_wr, ...]` instruction always accesses DRAM on an eight-byte or quad-word boundary
- The sum of the prepend length and payload length must be an integral multiple of the bus width, except for a TBUF in which EOP is set. In MPHY-16 mode, the bus width is 32 bits—that is, four bytes

9.3 Data Structures

This section describes the data structures used for the POS transmit microblock.

9.3.1 Globals Stored in Absolute Registers

Table 9-1 lists the variables are stored in absolute registers. They are used to find a port on which a packet may be transmitted.

Table 9-1. Packet TX for MPHY-16: Globals Stored in Absolute Registers

Global	Description
@qstat_vector	Each port occupies 1 bit, when set, it means that there are packets queued for this port, initialized to 0x00000000
@mphy_stat_fc_vector	Each port occupies 1 bit, when set, it means that this port is not blocked by PHY status, waiting port ready during initialization
@rr_vector	Round robin mask, initialized to 0xFFFF
@txc_left_paylo_vector	Each port occupies 1 bit, when set, it means that this port is not blocked by waiting for leftover bytes to be saved into local memory, initialized to 0xFFFF
@txc_rd_snd_bd_vector	Each port occupies 1 bit, when set, it means that this port is not blocked by waiting for the secondary buffer meta data read to complete, initialized to 0xFFFF. If the quotation of the buffer size divided by the tbuf element size is larger than the total thread number in one ME, this restriction can be removed.

9.3.2 Context Relative Thread Execution Status Flag—`exe_stat_flag`

The GPR below store various execution flags for a thread.

Table 9-2. Context Relative Thread Execution Status Flag - `exe_stat_flag`

LW	Bits	Size	Field	Description
0	31:6	26		Reserved
	5:5	1	eob_mpkt_flag	Flag for free buffer
	4:4	1	skip_transmit_flag	Flag to skip transmit a tbuf, due to no payload and no prepend

Table 9-2. Context Relative Thread Execution Status Flag - exe_stat_flag

LW	Bits	Size	Field	Description
	3:3	1	read_sop_bd_flag	Flag to save sop meta data
	2:2	1	save_leftov_to_lm_flag	Flag to save leftover to local memory
	1:1	1	leftov_from_pb_flag	Flag to transmit leftover payload from previous buffer
	0:0	1	read_snd_bd_flag	Flag to save secondary buffer meta data

9.3.3 TX Request—One Longword

Table 9-3 describes the transmit request received from the Queue Manager.

Table 9-3. TX Request - One Longword

LW	Bits	Size	Field	Description
0	31:31	1		Valid bit
	30:28	3		Reserved
	27:24	4		Port number
	23:0	24		SOP meta handle >> 2

9.3.4 Queue Structure for Each Port

Queue information for each port is located in two different locations for the flexibility to increase the queue size for each port and to use both local memory pointers LM_ACTIVE_0 and LM_ACTIVE_1 effectively.

The first 512 long words in local memory are used to store the queue entries for all ports. The upper 128 words are used to the queue descriptors—that is, head, tail, count, and so on. Each queue entry is four long words. So if there are eight ports on a microengine, then for each port we can queue up to 16 transmit requests. Each queue descriptor is eight long words, though only seven are used for now.

9.3.4.1 Queue Entry Structure

Table 9-4 describes the structure of a single entry in the queue.

Table 9-4. Queue Entry Structure

LW	Bits	Size	Field	Description
0	31:31	1	p_sop_flag	SOP bit for the packet in queue entry
	31:30	1	ab_rd_nbd_flag	Read next buffer meta data bit for the active buffer of queue entry
	29:28	2		Reserved
	27:24	4	ab_flist	Free list of the active buffer of queue entry
	23:0	24	ab_bd	Buffer descriptor (BD) of the active buffer of queue entry
1	31:16	16	ab_paylo_rmnd	Remaining bytes of the payload to be sent out in the active buffer of entry queue

Table 9-4. Queue Entry Structure

LW	Bits	Size	Field	Description
	15:0	16	ab_offset_rpaylo	Offset of the first byte in the remaining payload in the active buffer of queue entry
2	31:31	1		Reserved
	30:30	1	sb_rd_nbd_flag	Read next buffer meta data bit of the secondary buffer of queue entry
	29:28	2		Reserved
	27:24	4	sb_flist	Free list of the secondary buffer of queue entry
	23:0	24	sb_bd	Buffer descriptor (BD) of the secondary buffer of queue entry
3	31:16	16	sb_buf_size	Buffer size of the secondary buffer of queue entry
	15:0	16	sb_offset	Offset of the secondary buffer of queue entry

9.3.4.2 Queue Descriptor Structure

Table 9-5 describes the structure for a queue descriptor. There is one queue descriptor per port.

Table 9-5. Queue Descriptor Structure

LW	Bits	Size	Field	Description
0	31:24	8		Reserved
	23:16	8	pkt_in_queue	Packets in queue for this port
	15:8	8	tail_offset	Queue Tail for this port (Bytes offset within queue info)
	7:0	8	head_offset	Queue head for this port (Bytes offset within queue info)
1	31:0	32	allowed_mpkts	Remaining mpackets to transmit without checking
2	31:0	32	phy_depth	Remaining mpackets to transmit without checking port tx_status The number of mpackets which the port can accept without the risk to overflowing that port, when it's Tx_status equals to 1, this number depends on the tx FIFO size of the port and tbuf element size
3	31:0	32	pkt_txed	Packets transmitted in this port
4	31:17	15		Reserved
	16:16	1	lp_flag	Flag to fetch payload left by previous buffer in the first mpacket in this buffer due to the restriction put by hardware
	15:8	8		Reserved
	7:0	8	lp_len	The length of leftover in bytes, less than 4 bytes
5	31:0	32	lp_lw0	Long word of payload left by previous buffer
6	31:24	8		Reserved
	23:0	24	sbd_next_bd	Buffer descriptor (BD) of the next buffer of the secondary buffer for the packet currently begin transmitted
7:15				Reserved

9.3.5 Output Transmit Control Word —Two Longwords

Table 9-6 describes the format of the output transmit control word.

Table 9-6. Output Transmit Control Word - Two Longwords

LW	Bits	Size	Field	Description
0	31:24	8	payload_len	Payload length in bytes
	23:21	5	payload_offset	Prepend offset in bytes
	20:16	3	prepend_len	Prepend length in bytes
	15:13	3	prepend_offset	Payload offset in bytes
	12:12	1		Reserved
	11:11	1	skip	Skip
	10:10	1	err	ERR
	9:9	1	sop	SOP
	8:8	1	eop	EOP
	7:4	4		Reserved
	3:0	4	channel	Channel
1	31:0	32		Reserved

9.3.6 Statistics

The Packet TX block maintains the following statistics in SRAM on a per port basis. The microblock maintains these as 32-bit counters and the Intel XScale® core component keeps a 64-bit version of these counters (see [Section 2.5, “Statistics and Handling of 64-bit Counters”](#) on page 63).

Table 9-7. Packet TX Statistics

Counter	Offset from base for port
Packets Transmitted	0x0
Packets Dropped	0x4
Bytes Transmitted	0x8
Bytes Dropped	0xc

9.4 Design

The Packet Transmit block for MPHY-16 runs on two microengines each of which handle a mutually exclusive subset of the ports in the system. Each microengine handles traffic at up to OC-24 rates. The QM sends transmit requests to each TX microengine on a different scratch ring.

The Packet Transmit block queues all requests in local memory—one queue per port. Each request is associated with a packet, which needs to be segmented into TBUFs and transmitted. Unlike the CSIX TX block which is cell based, the Packet TX is a packet based transmit block and receives only one transmit request per packet.

To select a specific port on which to send a TBUF, the Packet TX block maintains a set of bit vectors—bit vector for ports with data, bit vector for ports that are not flow controlled etc. In addition it keeps a bit mask to round robin among the ports. Using these bit masks and the ffs instruction, the Packet TX block schedules a port to transmit a TBUF from.

For each port, the microblock caches information from the meta data for the packet at the head of the queue. For each packet, information from the meta data of the current buffer being handled and the meta data of the next buffer in the buffer chain are cached. The meta data for the next buffer is read in while the current buffer is being processed. To prevent a port from being scheduled while the meta data of a buffer at the head of the queue is being read in, a bit vector of ports that are not waiting on the meta data read is maintained.

Another restriction that the design needs to deal with is that the sum of the prepend length and payload length must be an integral multiple of the bus width, except for a TBUF in which EOP is set. In MPHY-16 mode, the bus width is 32 bits—or four bytes. To comply with this restriction, a maximum of three bytes at the end of a buffer may be saved in local memory and transmitted with the payload of the next TBUF for the packet. To prevent a port from being scheduled before these leftover bytes are saved in local memory, a bit vector of ports that are not waiting for the copy of the leftover bytes is maintained. If at the end of a buffer, less than four bytes are left, then no TBUF is transmitted in that slot. Instead the bytes are saved in local memory and the SKIP bit is set in the TBUF allocated for that slot.

9.5 Flow Chart

Figure 9-1. Packet Transmit—MPHY-16 Configuration—Flowchart Page 1 of 5

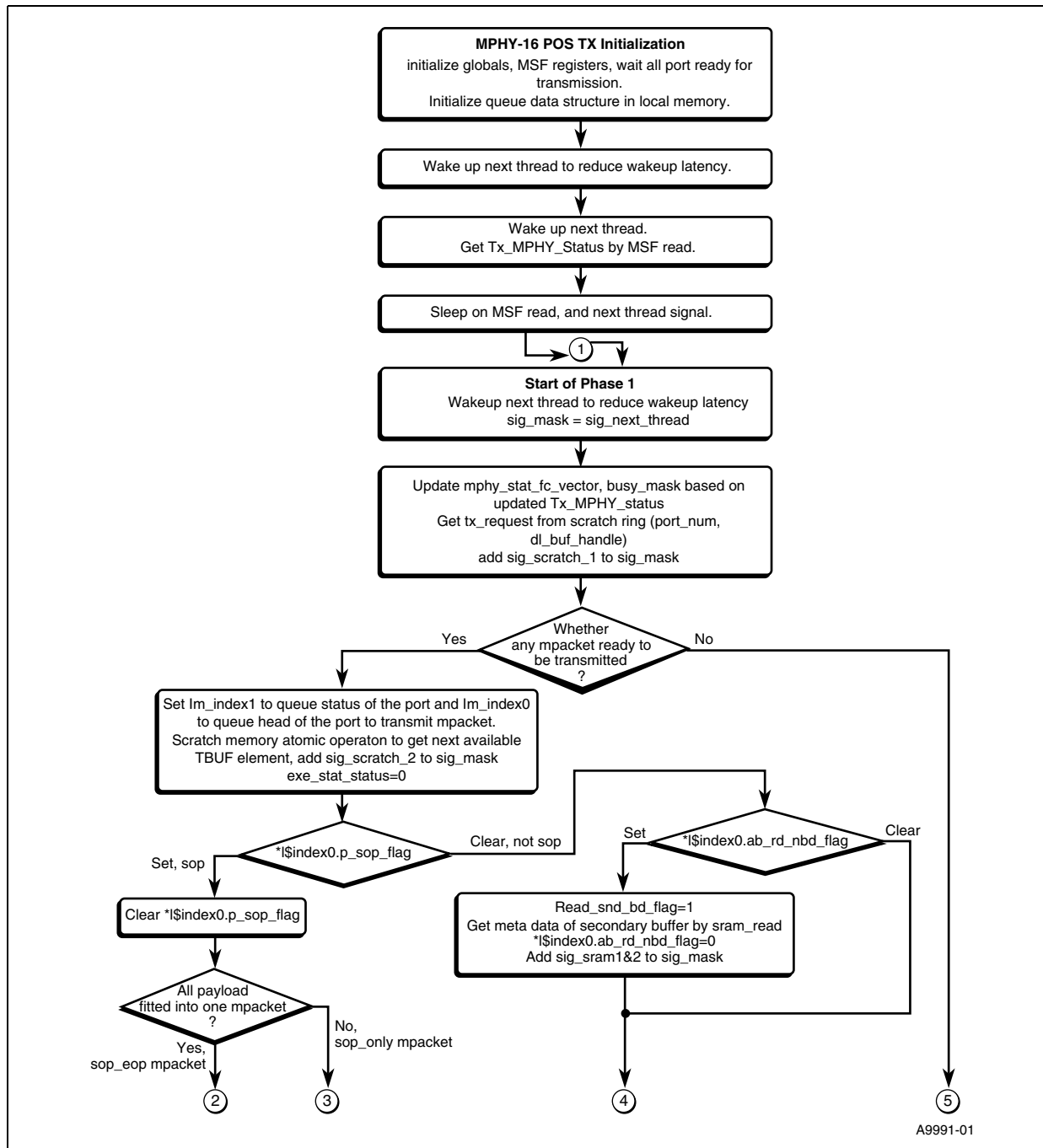


Figure 9-2. Packet Transmit—MPHY-16 Configuration—Flowchart Page 2 of 5

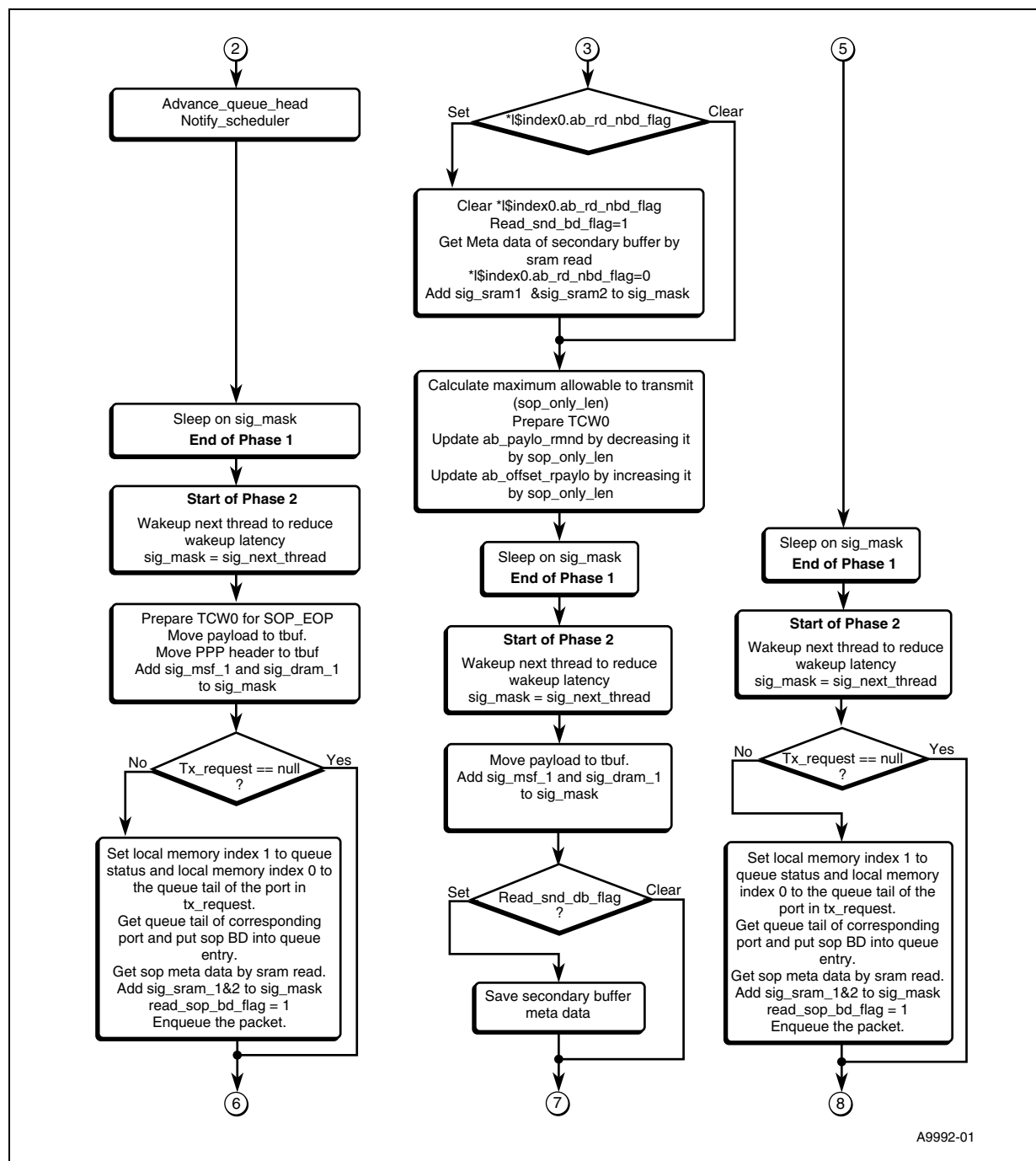
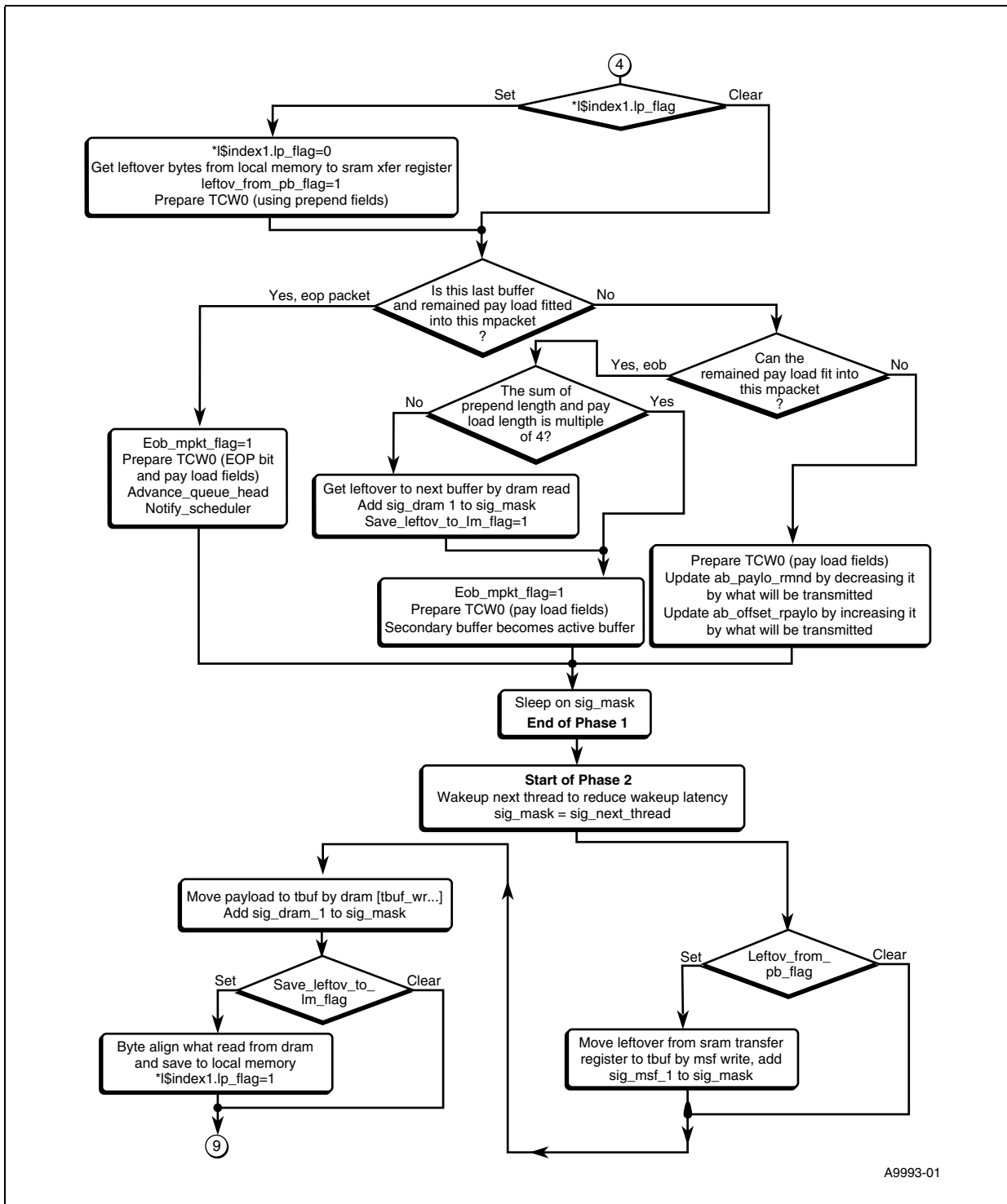


Figure 9-3. Packet Transmit—MPHY-16 Configuration—Flowchart Page 3 of 5



A9993-01

Figure 9-4. Packet Transmit—MPHY-16 Configuration—Flowchart Page 4 of 5

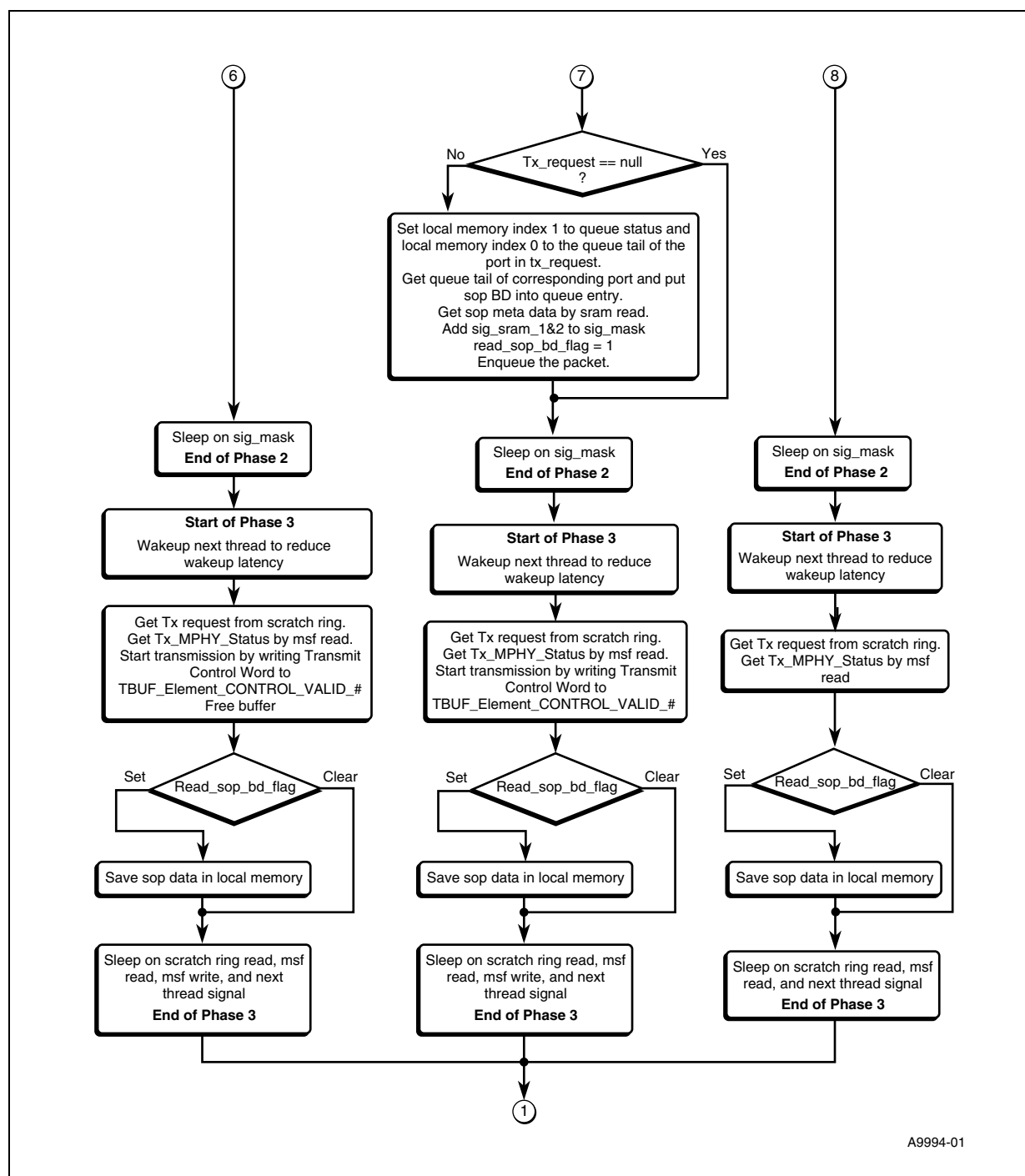
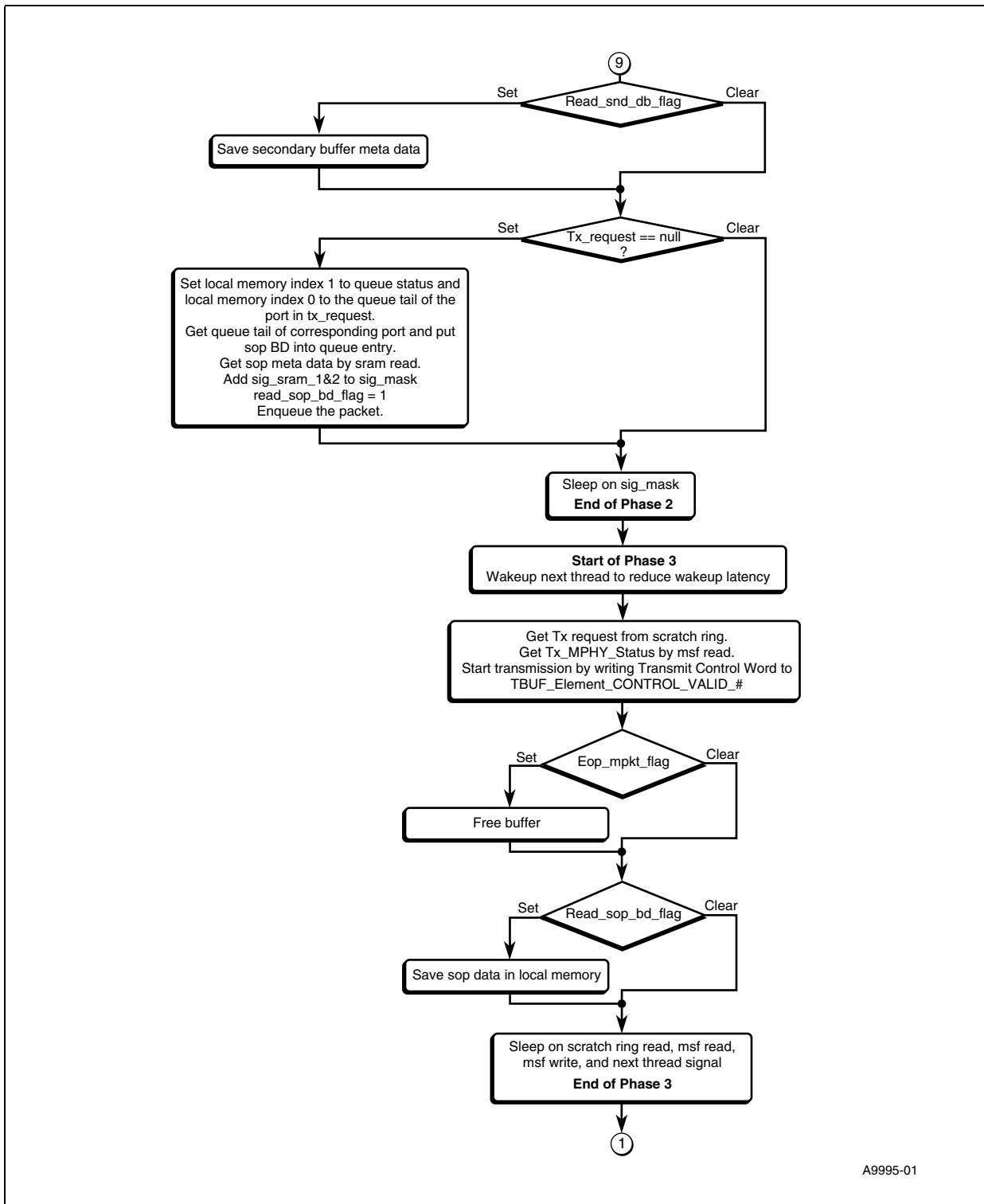


Figure 9-5. Packet Transmit—MPHY-16 Configuration—Flowchart Page 5 of 5



9.6 Performance Analysis

The Packet Transmit microblock uses two microengines each of which handles up to OC-24 rates of traffic. This implies that the available cycle count budget to handle a POS min packet is 97×2 cycles. Table 9-8 shows the instruction estimate for the POS microblock to handle a POS min packet in the worst-case code path.

Table 9-8. Performance Analysis - Instruction Estimate

Component	Worst Case Cycle count
Phase 1	91
Phase 2	54
Phase 3	18
Total	163 (Available budget is 176)

Table 9-9 shows the I/O operations performed in the different phases of the microblock for the worst case. Each thread has an available I/O latency of $97 \times 8 \times 2$ cycles to handle a min sized POS packet.

Table 9-9. Performance Analysis - I/O Operations

I/O operations	Phase
Scratch read of transmit request	1
Scratch read of next available TBUF element	1
MSF read of one long word (Tx_Sequence_0)	1
DRAM read of 16 bytes	1
SRAM read of 3 words for next buffer meta data	1
MSF write of 2 byte PPP header	2
Dram to tbuf write of 128 bytes	2
SRAM read of 3 words for new packet's SOP meta data	2
MSF read for Tx_MPHY_Status (1 word)	3
MSF write to validate TBUF and start transmit	3
Enqueue operation to free buffer	3

9.6.1 Characterization Data

Table 9-10. Packet TX for MPHY-16 Microblock Characterization Data

Data	Value
General:	
Microblock Name	MPHY16_TX
Microblock Version Number	1.0

Table 9-10. Packet TX for MPHY-16 Microblock Characterization Data (Continued)

Data	Value
Implementation Language	microcode
Configuration Options use to gather this set of data	For first TX microengine: 1. THIS_ME=MPHY16_PACKET_TX_FIRST_ME 2. SCHEDULER_ME=0x03 3. USE_ME_INIT_SIGNAL
	For second TX microengine: 1. THIS_ME=MPHY16_PACKET_TX_SECOND_ME 2. SCHEDULER_ME=0x03 3. USE_ME_INIT_SIGNAL
Measurement Environment (tool settings)	
Performance: (cycle counts, latencies, bandwidths)	
Instruction cycle count per packet (common case packet / worst case cycle count)	First TX ME: 163 (available budget: 176) Second TX ME: 163 (available budget: 176)
Common-case packet/path assumptions to be documented here	Assumptions <ul style="list-style-type: none"> This is a two ME TX microblock. Each ME supports 8 ports, the two ME are exactly the same, except the first ME initializes the MSF interface. Scheduler will schedule TX requests based on how many packets has been transmitted out of MSF to prevent overflowing of TX requests in local memory queue. All numbers reported are for worst-case for OC48_POS normal path
Scratch Memory	
# of longwords read (for bandwidth calculations)	2
# of longwords written (for bandwidth calculations)	0
# and type of each atomic operation performed (for bandwidth calculations)	0
SRAM	
# of longwords read	Meta data: 3
# of longwords written	0
# and type of each atomic operation performed (for bandwidth calculations)	0
DRAM	
# of quadwords read	6
# of quadwords written	6
Other I/O instructions (i.e. MSF, PCI, CAP, and Hash)	5
List of dependent I/O accesses in the longest latency path	
Per-Microengine Resources:	

Table 9-10. Packet TX for MPHY-16 Microblock Characterization Data (Continued)

Data	Value
Control-Store Usage (# of instructions used)	First TX ME: 1089, second TX ME: 1034
Local Memory Footprint (# of long words used)	640
Local Memory Configuration (shared, or per-context pointer)	per-context pointer
Local Memory - # of LM pointers used	2
GPR Usage – minimum, static usage (absolutes, static, globals)	
Transfer Reg. Usage – minimum, static usage	
Next Neighbor Reg. Usage – minimum, static usage	0
Signal Usage – minimum, static usage	0
CAM used? (yes or no)	No

Global Resources:

Scratch footprint (# of longwords used) – constant or formula ...	0
SRAM footprint (# of longwords used) – constant or formula ...	0
DRAM footprint (# of quadwords used) – constant or formula ...	0
Q-Array usage - # of queues used and if they need to be cached	0
CRC Unit used?	
Hash Unit used? (yes or no)	no

MSF Usage Information:

Media Bus Configuration	SPI3 MPHY16 POS TX
RBUF, TBUF usage	TBUF
CBus signals	-

Other Information:

Critical Section Length (compute cycles + memory accesses)	0
# of phases	First TX ME: 3, Second TX ME: 3
Packet Metadata - fields read	bufferSize payloadOffset nextBufferHandle
Packet Metadata - fields written	-
Header - fields read	
Header - fields written	

Table 9-10. Packet TX for MPHY-16 Microblock Characterization Data (Continued)

Data	Value
Documentation:	
Thread Ordering Requirements	
OS dependencies	VxWorks, Linux
Chip/hardware dependencies (i.e. Crypto Unit in 2850)	2400
Tested on which SDK Release(s)	IXA SDK 3.1
Tested on hardware? Which hardware configuration?	None
Tested in which applications (not an all inclusive list)	Several IXS SDK 3.1 applications
Possible Configuration Options	
Major limitations (i.e. Rx/Tx microblocks written to use entire ME)	None
Packet Sequencing Issues (esp. in POT applications)	None
Core Component or Interface requirements or dependencies	ATM/POS TX and Ethernet TX core components

This section describes the low-level design and implementation of the Packet Transmit microblock for the IXP2800 for a single OC-192 port. This microblock was created by modifying the IXP2400 Packet Transmit block (SPHY 1x32).

10.1 Overview

The Packet Transmit microblock runs on two microengines in a context pipeline connected by a Next Neighbor ring. It receives requests to transmit packets from the queue manager via a Next Neighbor ring. Since the transmit requests are packet based, the microblock needs to segment the payload into mpackets, copy them into TBUFs and validate them for the MSF to transmit. Since packets cannot be interleaved, an entire packet needs to be segmented and transmitted before the next packet can be processed. This means that every time a thread is ready to process an mpacket, it needs to check if there is a current packet being segmented. If there is, then it needs to transmit one TBUF for that packet. If there is no current packet, then it needs to get the next packet transmit request and start a segmentation context for that packet.

The microblock is implemented such that the first microengine essentially deals with packets and the second microengine with m-packets. The first microengine keeps a queue of transmit requests in local memory. Each thread of the first microengine always works on the packet at the head of the local memory queue and issues a request to the second microengine to send one m-packet from it. The Transmit microblock also keeps track of the number of packets transmitted and uses the reflector write mechanism to tell the scheduler about how many packets have been transmitted. The scheduler uses this information to keep track of how many packets are in flight. If this exceeds a configurable threshold, it stops scheduling on that port. If the Transmit block receives a large packet, it can take a long time to segment and transmit it. The packets in flight mechanism prevents the scheduler from overflowing the transmit queue in this case.

10.2 Assumptions and Dependencies

- The DRAM[tbuf_wr, ...] instruction always accesses DRAM on a 8-byte or quad-word boundary
- This microblock uses 128 byte tbufs allowing for a maximum of 64 tbufs in the pool. The MSF provides a CSR to read the number of tbufs transmitted. This is read periodically by the microblock and used to check for overflow before copying from DRAM to tbuf.

10.3 Data Structures

The data structures are identical to the data structures used for the IXP2400 design described in [Section 8.3, “Data Structures”](#) on page 156 of [Chapter 8, “Packet TX for SPHY and MPHY-4”](#).

The format of the Next Neighbor ring between the first microengine and the second microengine is described below:

[Table 10-1](#) shows each entry for three words NN ring.:

Table 10-1. Three-word NN Ring between the first and the Second Microengine

LW	Bits	Size	Description
0	31:0	32	Pointer to meta data (used to free buffer)
1	31	1	Bit is clear if the m-packet is sop
	30	1	Bit is clear if the m-packet is eop
	29:0	29	Offset of payload to be transmitted
2	31:0	32	Payload size to be transmitted

[Table 10-2](#) shows each NN ring entry, if the m-packet is non-sop, where 3 more long words are included on the ring.

Table 10-2. Six-word NN Ring for Non-sop M-packet

LW	Bits	Size	Description
3	31:0	32	Bytes from previous buffer to be prepended to the current buffer
4	31:0	32	Exe_stat_flag: information about various condition flags
5	31:0	32	Partially created transmit control word

10.4 Design

Figure 10-1 describes the high-level design for the first microengine. All eight threads run in strict thread order which is ensured by an inter-thread signal.

Figure 10-1. High-level Design for the First Microengine

```
while (1)
{
    Read Transmit Request for packet p1 from the Next Neighbor Ring
    Issue read of the metadata for packet p1
    Get the request at the head of local memory queue (packet p2)
    Calculate the DRAM address for the next m-packet for p2
    Send a request to the next microengine to send out the next m-packet from p2
    Signal next thread
    Wait for previous thread signal and meta read signal
    Store the meta data read for p1 into local memory
}
```

Figure 10-2 illustrates the high level design of the second microengine. This microengine only works on m-packets. It reads an m-packet transmit request, copies data for the m-packet from DRAM to TBUF, and validates the TBUF for transmit. To cover DRAM latency, the design allows any level of packet interleaving (`max_phase`), which may be configured at compile time. For example, if `max_phase` is 2, it handles $2 * \text{max_phase} = 4$ packet at a time, that is, the microblock handles two packets in every iteration of the while loop and delays the wait for the DRAM signal accordingly.

Figure 10-2. High-level Design for the Second Microengine

```
while (1)
{
    for (i=0; i< max_phase; i++)
    {
        phase# i:
        wait for signals - tbuf_validate(i-1), dram(i), prepend(i)
        issue tbuf validate of mpacket read in previous iteration and generate signal tbuf_validate_i
        read a mpacket transmit request from NN ring
        allocate a tbuf for the mpacket
        sends scheduler feedback about packet transmit if sop is detected
        issue dram to tbuf transfer request and generate signal dram_i
        add PPP header and generate signal prepend_i
        branch to phase_(i+1);
    } // here (i+1) means (i+1) mod max_phase;
    // and (i-1) means (i-1) mod max_phase;
}
```

10.5 Performance Analysis

The table below shows the instruction cycle counts in the critical path (min 49 byte POS packet) for the two microengine design which is used in the OC-192 POS application.

Table 10-3. Instruction Cycle Counts in the Critical Path for Two Microengine Design

Microengine	Worst Case Cycle count	IXP2800 budget for POS min packet at OC-192 rates
Microengine 1	54	57
Microengine 2	53	57

The table shows that the microblock meets the performance requirements for OC-192 POS with the two microengine design.

Table 10-4. Performance Requirements for OC-192 POS—Two Microengine Design

I/O operations	Number of long words
MSF read (Tx_Sequence_x)	1
Reflect write to notify scheduler	1
Dram to tbuf write of up to 128 bytes	Up to 16 quadwords
SRAM read for new packet's SOP meta data	3
MSF write to validate TBUF and start transmit	2
Enqueue operation to free buffer	1

10.5.1 Characterization Data

Table 10-5. Packet TX for OC-192 POS Microblock Characterization Data

Data	Value
General:	
Microblock Name	SPHY_MPHY4_2800_TX
Microblock Version Number	1.0
Implementation Language	microcode

Table 10-5. Packet TX for OC-192 POS Microblock Characterization Data (Continued)

Data	Value
Configuration Options use to gather this set of data	<p>For First TX ME:</p> <ol style="list-style-type: none"> 1. SCHEDULER_ME=0x03 2. ADD_L2_HEADER 3. POS_TX 4. DYNAMIC_PPP_ENCAP 5. ADD_PPP_ADDR_AND_CNTRL <p>For Second TX ME:</p> <ol style="list-style-type: none"> 1. ADD_L2_HEADER 2. POS_TX 3. DYNAMIC_PPP_ENCAP 4. ADD_PPP_ADDR_AND_CNTRL 5. VALIDATE_TBUF_IN_THIRD_TX_ME <p>For Third TX ME:</p> <ol style="list-style-type: none"> 1. CHECK_PORT_TX_FIFO_STATUS
Measurement Environment (tool settings)	
Performance: (cycle counts, latencies, bandwidths)	
Instruction cycle count per packet (common case packet / worst case cycle count)	First TX ME: 56, Second TX ME: 53, Third TX ME: 47
Common-case packet/path assumptions to be documented here	<p>Assumptions:</p> <ul style="list-style-type: none"> • This is a three ME TX microblock, the first Tx ME processes packets, the second ME processes mpackets, and the third ME validates TBUF elements and handles flow control based on media port TX FIFO status. • Scheduler will schedule TX requests based on how many packets have been transmitted out of MSF to prevent overflowing of TX requests in local memory queue. • This microblock can be configured to run in two ME configuration, if the worst-case instruction budget is not tight, e.g., 1x10 Gbs ethernet application. • All numbers reported are for worst-case for OC192_POS normal path.
Scratch Memory	
# of longwords read (for bandwidth calculations)	0
# of longwords written (for bandwidth calculations)	0
# and type of each atomic operation performed (for bandwidth calculations)	0
SRAM	
# of longwords read	Meta data: 3
# of longwords written	0
# and type of each atomic operation performed (for bandwidth calculations)	0
DRAM	

Table 10-5. Packet TX for OC-192 POS Microblock Characterization Data (Continued)

Data		Value
# of quadwords read	6	
# of quadwords written	6	
Other I/O instructions (i.e. MSF, PCI, CAP, and Hash)	6	
List of dependent I/O accesses in the longest latency path		

Per-Microengine Resources:		
Control-Store Usage (# of instructions used)		First TX ME: 1407, Second TX ME: 394, Third TX ME: 499
Local Memory Footprint (# of long words used)	528	
Local Memory Configuration (shared, or per-context pointer)		per-context pointer
Local Memory - # of LM pointers used	2	
GPR Usage – minimum, static usage (absolutes, static, globals)		
Transfer Reg. Usage – minimum, static usage		
Next Neighbor Reg. Usage – minimum, static usage	0	
Signal Usage – minimum, static usage	0	
CAM used? (yes or no)	No	

Global Resources:		
Scratch footprint (# of longwords used) – constant or formula ...	0	
SRAM footprint (# of longwords used) – constant or formula ...	0	
DRAM footprint (# of quadwords used) – constant or formula ...	0	
Q-Array usage - # of queues used and if they need to be cached	0	
CRC Unit used?	no	
Hash Unit used? (yes or no)	no	

MSF Usage Information:		
Media Bus Configuration		SPI4 POS TX
RBUF, TBUF usage		TBUF
CBus signals		-

Other Information:		
Critical Section Length (compute cycles + memory accesses)	0	

Table 10-5. Packet TX for OC-192 POS Microblock Characterization Data (Continued)

Data	Value
# of phases	First TX ME: 1, Second TX ME: 1, Third TX ME: 2
Packet Metadata - fields read	bfferSize, payloadOffset, nextBufferHandle
Packet Metadata - fields written	-
Header - fields read	None
Header - fields written	None

Documentation:

Thread Ordering Requirements

OS dependencies VxWorks, Linux

Chip/hardware dependencies (i.e. Crypto Unit in 2850) 2800

Tested on which SDK Release(s) IXA SDK 3.5

Tested on hardware? Which hardware configuration? Yes, IXDP 2800

Tested in which applications (not an all inclusive list)

Possible Configuration Options

Major limitations (i.e. Rx/Tx microblocks written to use entire ME) None

Packet Sequencing Issues (esp. in POT applications) None

Core Component or Interface requirements or dependencies ATM/POS TX and Ethernet TX core components

11.1 Overview

The ATM AAL5 TX microblock performs AAL5-CPCS, AAL5-SAR and ATM layer functions. The AAL5-SSCS layer is not used.

11.1.1 CPCS Functions

The AAL5 TX microblock receives a CPCS-SDU—that is, a user data frame, between 1-655,35 bytes.

It pads the SDU such that the SDU plus an 8-byte trailer are a multiple of 48 bytes. It also adds the 8-byte trailer to the SDU to form the CPCS-PDU.

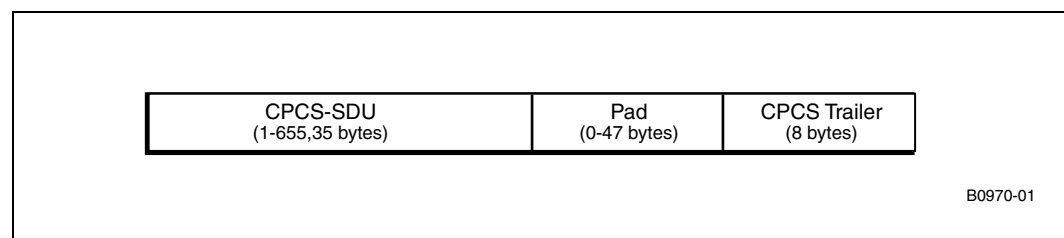
The CPCS-PDU is then passed to the SAR layer. [Table 11-1](#) shows the format of 8-byte CPCS trailer:

Table 11-1. Format of 8-byte CPCS Trailer

LW	Bits	Size	Field	Description
0	31:24	8	CPCS-UU	Used to transparently transfer CPCS user-to-user information. Set to 0 in this implementation.
	23:16	8	CPI	Currently zero. Exists for boundary alignment, other uses are under study.
	15:0	16	CPCS-SDU length	Size of the CPCS-SDU in bytes (between 1-655,35).
1	31:0	32	CRC	Detects bit errors in the CPCS-PDU. It contains the value of a CRC-32 calculation, which is performed over the entire CPCS-PDU, including the CPCS-PDU payload, the pad field, and LW0 of the CPCS-PDU trailer (i.e. the entire CPCS-PDU except the CRC-32 field itself).

[Figure 11-1](#) shows the format of the entire CPCS-PDU.

Figure 11-1. Format of CPCS-PDU



11.1.2 AAL5-SAR Functions

The CPCS-PDU received by the SAR layer becomes the SAR-SDU. The AAL5 TX microblock also implements the SAR layer and segments this SDU into 48-byte SAR-PDUs. The SAR layer does not add any headers. The SAR layer passes the 48-byte PDUs to the ATM layer.

11.1.3 ATM Layer Functions

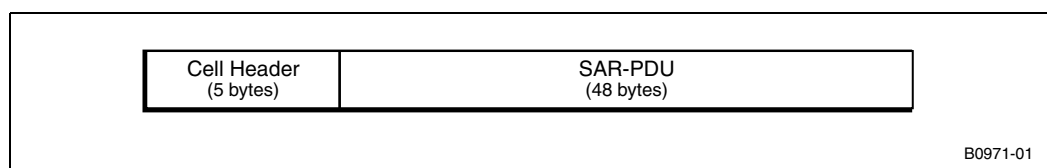
The ATM layer adds a 4-byte cell header to the each SAR-PDU. The last SAR-PDU of the packet also has the ATM user-to-user (AUU) bit set in its cell header. The fifth (HEC) header byte is added by the framer (hardware) and not by the AAL5 TX microblock. The 52-byte cell is sent out in a TBUF element. [Table 11-2](#) shows the format of the 5-byte cell header.

Table 11-2. Format of 5-byte Cell Header

LW	Bits	Size	Field	Description
0	31:28	4	Generic Flow Control	Set to 0 for uncontrolled equipment. Set to 0 in this implementation.
	27:19	8	Virtual Path Identifier	VPI and VCI identify the connection that this cell belongs to. In this implementation, a layer 3 lookup of incoming parameters obtains the outgoing VPI/VCI.
	19:4	16	Virtual Con Identifier	See description of VPI above.
	3:3	1	Payload Type	When set to 0 it indicates user data cell. When set to 1 it indicates control or mgmt cell. Set to 0 in this implementation.
	2:2	1	Payload Type	When set to 0 it indicates no congestion. When set to 1 it indicates congestion. Set to 0 in this implementation.
	1:1	1	Payload Type (AUU)	When set to 0 for AAL5 it indicates non-EOP cell. When set to 1 for AAL5 it indicates EOP cell of the packet. Set accordingly in this implementation.
	0:0	1	Cell loss priority	Sets priority of cells during congestion. When set, the cell may be discarded during congestion. Set to 0 in this implementation.
1	31:8	24	Not used	
	7:0	7	HEC	Allows for correction of all single bit errors in the cell header. Not set by the AAL5 TX microblock. Set by the ATM framer (hardware).

[Figure 11-2](#) shows the format of the 53-byte ATM cell.

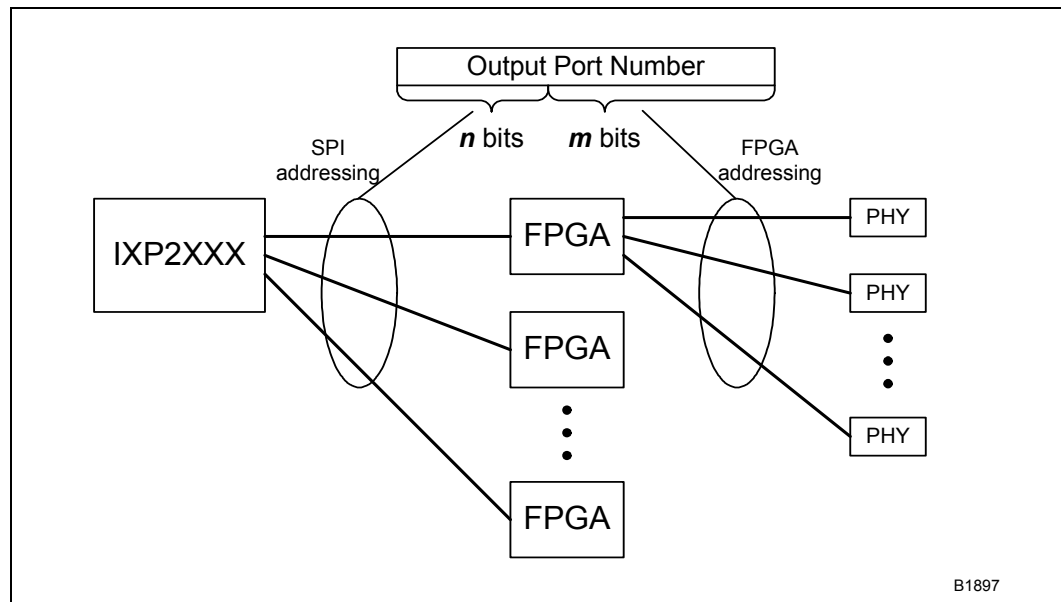
Figure 11-2. Format of the 53-byte ATM Cell



11.1.4 Extended Port Addressing

The ATM cell can be sent out to one of the output ports. If an application requires more external ports than the IXP2XXX Network Processor can address (on the SPI bus) a specialized hardware (FPGA) must be used to achieve the required number of external ports. The port number used by the AAL-5 Transmit microblock contains information about the SPI address and the FPGA address. Figure 11-3 illustrates the output port addressing scheme.

Figure 11-3. Extended Port Addressing



The n most significant bits of the port number are used for device addressing on the SPI bus. The m least significant bits are used for addressing the output PHY connected to the FPGA device.

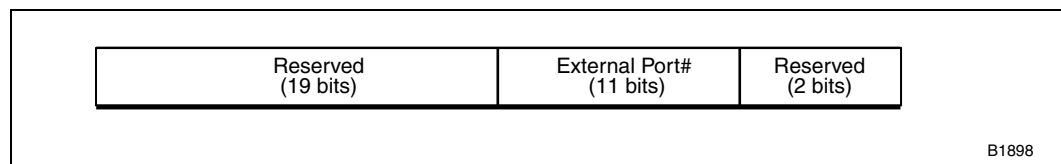
The port number split (SPI and FPGA addressing) is compile-time defined by using appropriate compile switches.

11.1.4.1 FPGA Addressing

If the extended port addressing is used, each ATM cell sent by the IXP2XXX is followed by the FPGA header. This header contains information about the external port to which the specified cell was directed and has to be removed before sending the cell to the external port.

Figure 11-4 shows the FPGA header format.

Figure 11-4. Format of the FPGA Header



11.1.5 Design Overview

In every iteration, the AAL5 TX microblock processes 48 bytes of the CPCS-SDU (called packet payload from now on). The last iteration for the packet processes between 0-48 bytes.

At the start of every iteration, the Queue Manager puts a dequeue request on a scratch ring. This indicates there is data on that queue that needs to be transmitted. The AAL5 TX microblock reads this request. It creates a new context if the data belongs to a new packet. It then tries to read at most 48 bytes worth of data from that queue. Padding is added if less than 48 bytes are remaining in the current packet. Next, it computes CRC on the data and stores the result in the context. Finally it transmits the 48-byte data along with a 4-byte header (52 bytes in all) as an ATM cell into a TBUF element. If the end of packet is reached, the packet length and stored CRC are used to prepare the 8-byte CPCS trailer, which is also sent out with the remaining payload. In this manner, AAL5-CPCS, AAL5-SAR and ATM layer features listed above are implemented.

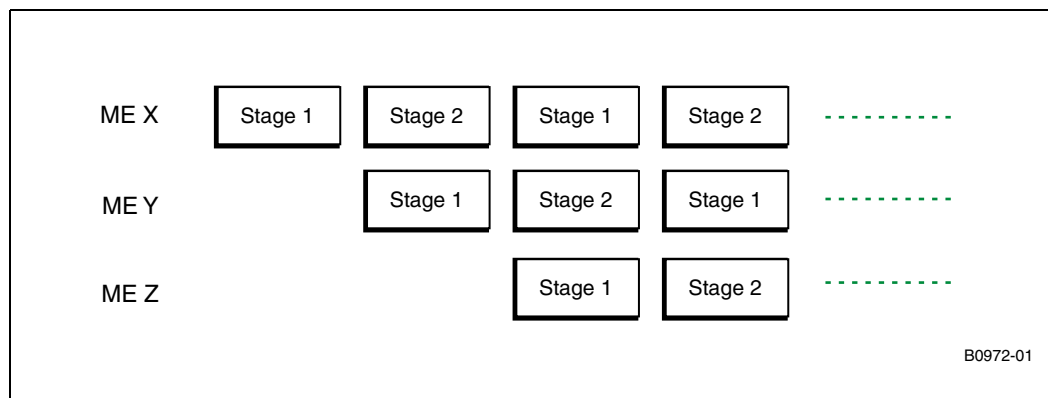
A 53-byte ATM cell with a 4% SONET overhead, i.e. 55.12 bytes need to be transmitted at OC-48 (or quad OC12) i.e. 2500 Mbps. This implies (55.12×8) bits i.e. 440.96 bits need to be processed in $440.96 / (2500 \times 106)$ seconds. Assuming IXP2400 runs at 600Mhz, we get

$(600 \times 106) \times 440.96 / (2500 \times 106)$ i.e. 105.8 cycles, or approximately 106 cycles to send out a 53-byte ATM cell at OC-48 (or quad OC-12).

11.1.5.1 OC-48

The UTOPIA protocol in SPHY 1x32 mode is used. Three micro-engines running in a two stage functional pipeline are used to achieve OC-48 performance.

Figure 11-5. Two Stage Functional Pipeline for OC-48

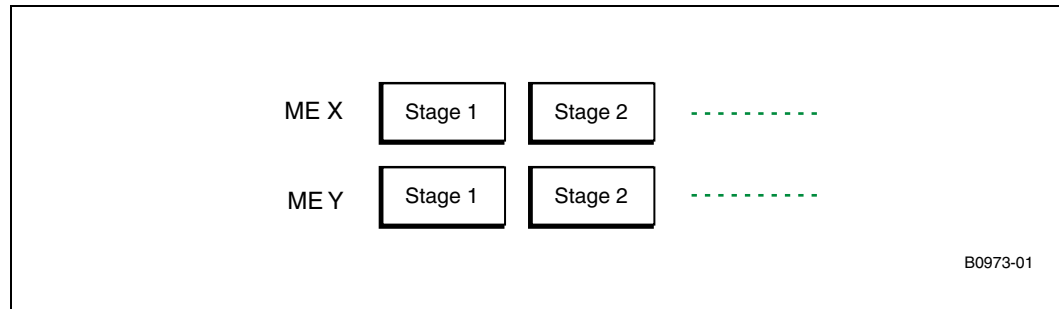


To meet OC-48 line rate on IXP2400 using three ME's, 52 bytes (an ATM cell worth) need to be sent out every 106×3 —that is, 318 cycles.

11.1.5.2 Quad OC-12

The UTOPIA protocol in SPHY 4x8 mode. Two micro-engines running independently (4 threads per port) are used to achieve quad OC-12 performance.

Figure 11-6. Two MEs Running Independently



To meet quad OC-12 line rate on IXP2400 using two MEs, 52 bytes (an ATM cell worth) need to be sent out every 106x2 i.e. 212 cycles.

11.2 Assumptions, Dependencies and Risks

- The AAL5 TX microblock supports up to 64k queues. If there are more queues, additional space needs to be allocated in SRAM to store contexts for each queue.
- The AAL5 TX microblock assumes that cell-transmit requests for packets on the same queue are not interleaved, i.e. all the cell transmit requests for a packet appear contiguously before a request is received to transmit a cell from a different packet.
- In the transmit request obtained from the queue manager via the scratch ring, the SOP bit is set only for the first transmit request for a packet.
- If the transmit request for the last cell of a packet has more than 40 bytes of payload to be transmitted, the 8-byte CPCS trailer cannot be appended in this cell. This is because every cell transmitted should have no more than 48 bytes of payload. In this case, an extra cell needs to be scheduled by the scheduler. So now, the penultimate cell of the packet is sent out with necessary padding but without the CPCS trailer, while the extra cell that is scheduled contains 40 bytes of padding and the 8-byte CPCS trailer. The extra cell is now the last cell transmitted for the packet.
- A 52-byte cell is sent out by AAL5 TX via a TBUF element. The ATM framer (hardware) adds the HEC byte in the cell header.
- Head of line blocking is not addressed in this implementation. However, we can still achieve OC-48 line rates using SPHY mode. In SPHY mode, head of line blocking does not occur. Quad OC12 line rate in MPHY-4 mode or SPHY 4x8 mode can be achieved as well (with no head of line blocking) with minor modifications to this implementation. The design does not work on MPHY-16 since head of line blocking becomes an issue in that case.

11.3 Data Structures

11.3.1 Packet Metadata

The queue manager sends a dequeue request to the AAL5 TX microblock. The request contains a buffer handle that is an address in SRAM containing the metadata for the packet to be dequeued. The buffer handle is also used to compute the DRAM location that contains the actual packet data.

The meta data structure is defined in section Error! Reference source not found. The AAL5 TX microblock uses the following fields from the meta data structure.

- `buffer_next`: This is used to get the next buffer of the packet.
- `buffer_size`: This is used to compute the bytes remaining in the current buffer.
- `offset`: This is used to compute the absolute address of the data in DRAM.
- `packet_size`: This is used to figure out end of packet. `packet_size` is also stored in the CPCS trailer.
- `free_list_id`: This is used while freeing a used buffer.

11.3.2 Transmit Context (TXC)

In order to compute the CPCS trailer for a given packet, AAL5 TX needs to maintain a packet specific context. Since the entire packet appears continuously on a given queue, we can maintain a queue specific context instead, which is reset every time the SOP bit is set in the dequeue request (indicating start of a new packet). This context is called the Transmit Context (TXC). There are as many transmit contexts, as there are queues.

For OC-48, since we have a two-stage functional pipeline running on more than one microengine, we need to split up the transmit context into two parts, TXC1 and TXC2. TXC1 is read, modified and written only by stage 1 of the functional pipeline while TXC2 is read, modified and written only by stage 2 of the functional pipeline. Each stage flushes its TXC back to SRAM before allowing the next microengine to start that stage. This ensures parallel execution of the two stages without loss of TXC data integrity.

For quad OC-12, there is only one TXC (TXC1 + TXC2).

Table 11-3 shows the format of TXC1.

Table 11-3. Format of TXC1

LW	Bits	Size	Field	Description
0	31:16	16	<code>curr_buf_size</code>	Size of current buffer in bytes.
	15:0	16	<code>bytes_done_in_curr_buf</code>	A counter of the bytes processed in the current buffer.
1	31:0	32	<code>curr_payload_addr</code>	Absolute DRAM address of the start of payload for the current cell.
2	31:30	2	Reserved	
	29:24	6	<code>bytes_done_in_next_buf</code>	A counter of the bytes processed in the next buffer. (does not exceed 47 bytes)

Table 11-3. Format of TXC1 (Continued)

LW	Bits	Size	Field	Description
	23:0	24	curr_buf_hdl	Buffer handle of the current buffer of this packet.
3	31:24	8	Reserved	
	23:0	24	next_buf_hdl	Buffer handle of the next buffer of this packet.
4	31:16	16	next_buf_size	Size of the next (pre-fetched) buffer in bytes.
	15:0	16	bytes_done_in_packet	A counter of the bytes processed in the entire packet.
5	31:31	1	end_of_payload	Indicates that the last byte of the packet has been processed. If we receive a dequeue request with this bit set in TXC, it indicates this is the additional ATM cell scheduled by the QM to send the 8-byte CPCS trailer.
	30:28	3	Reserved	
	27:24	4	freelist	Buffer free-list ID.
	23:0	24	next_next_buf_hdl	Buffer handle of the next buffer with respect to the pre-fetched buffer.
6	31:16	16	packet_size	Size of the entire packet in bytes.
	15:0	16	outport	Output port number.

Table 11-4 shows the format of TXC2.

Table 11-4. Format of TXC2

LW	Bits	Size	Field	Description
0	31:0	32	cell_header	4-byte ATM cell header.
1	31:0	32	crc	Current CRC residue of the packet.

11.3.3 Counters

Table 11-5 shows the counters available via the COUNTERS build switch. Each counter is 32-bit and is stored in a specific location in SRAM. The AAL5 TX core component reads the 32-bit counter and may keep a 64-bit version of the counter.

Table 11-5. Counters Available via the Counters Build Switch

Description	Number of counters
Number of packets transmitted per VCQ	64k (one per VCQ)
Number of cells transmitted per VCQ	64k (one per VCQ)
Number of packet transmitted per output port	Up to 4 (one per physical port)
Number of cells transmitted per output port	Up to 4 (one per physical port)

11.3.4 Switches

The AAL5 TX microblock supports the following pre-processor switches.

Table 11-6. AAL5 TX Microblock Switches

Switch	Description
IXP2400	Use this switch when running the microblock on IXP2400.
IXP2800	Use this switch when running the microblock on IXP2800.
TX_PHY_MODE (in dl_system.h)	Set this switch to SPHY_1_32 when running the microblock in SPHY 1x32 mode. When run in this mode, we need 3 ME's running AAL5 TX as a functional pipeline to achieve OC-48 line rate and 1 ME running AAL5 TX to achieve OC-12 line rate.
TX_PHY_MODE (in dl_system.h)	Set this switch to SPHY_4_8 when running the microblock in SPHY 4x8 mode and the AAL-5 TX microblock has exclusive access to the MSF. When run in this mode, we need 2 ME's running AAL5 TX as a context pipeline to achieve OC-48 line rate.
TX_PHY_MODE (in dl_system.h)	Set this switch to SPHY_4_8_SHARED when running the microblock in SPHY 4x8 mode and the AAL-5 Transmit microblock does not have exclusive access to the MSF (there is another microblock which sends data to the MSF). When run in this mode, we need 2 ME's running AAL5 TX as a context pipeline to achieve OC-48 line rate.
NUMBER_OF_FP_MEs	Use this to define the number of ME's running the AAL5 TX microblock as a functional pipeline. When (TX_PHY_MODE == SPHY_1_32), set to the number desired number. When (TX_PHY_MODE == SPHY_4_8 or SPHY_4_8_SHARED), set to 1.
FIRST_AAL5_TX_ME	When (TX_PHY_MODE == SPHY_1_32), this switch identifies the first ME to start processing. When (TX_PHY_MODE == SPHY_4_8 or SPHY_4_8_SHARED), this switch identifies the 1st of 2 ME's used as a context pipeline.
NEXT_AAL5_TX_ME (only for TX_PHY_MODE == SPHY_1_32)	Use this to identify the next ME in the functional pipeline relative to this ME. Used for inter-me signaling.

Table 11-6. AAL5 TX Microblock Switches (Continued)

Switch	Description
SECOND_AAL5_TX_ME (only for TX_PHY_MODE == SPHY_4_8 or SPHY_4_8_SHARED)	Use this to identify the 2nd of 2 ME's as a context pipeline.
AAL5_TX_ME_INIT_MSF	Set this switch to indicate that the AAL-5 TX microblock needs to perform the MSF initialize operation during startup. This switch must be set if the switch TX_PHY_MODE (in dl_system.h) is set to SPHY_1_32 or SPHY_4_8. If TX_PHY_MODE (in dl_system.h) is set to SPHY_4_8_SHARED this switch should only be set when another microblock which has access to the MSF does not perform the MSF initialization during the startup process.
FPGA_HEADER	Use this switch to enable extended port addressing (only for MSF working in the SPI mode).
AAL5_FPGA_PORT_BITS (in dl_system.h)	Identifies the number of bits in the output port number used for FPGA addressing (see Figure 11-3 on page 199). This switch must be set if the extended port addressing is used.
AAL5_PHY_IF_BITS (in dl_system.h)	Identifies the number of bits in the output port number used for SPI addressing (see Figure 11-3 on page 199). This switch must be set if the extended port addressing is used.
RAW_MODE_ENABLED	Use this switch to allow the AAL-5 TX microblock to send out the ATM cells in the RAW mode (currently not used).
COUNTERS (optional)	Use this to define to enable counters explained above.

11.4 Algorithm

11.4.1 OC-48—One Page Summary

AAL5 TX is implemented as a two-stage functional pipeline.

The function of each stage and its phases is summarized in [Table 11-7](#) and [Table 11-8](#).

Table 11-7. OC-48 Algorithm - Pipestage 1

Phase	Description
Phase 1	Read dequeue request.
Phase 2	Check for null dequeue and setup CAM data
Phase 3	Fetch TXC1 in local memory. (Critical section 1 starts here)
Phase 4	Read payload from DRAM.
	Update TXC1.
	Pre-fetch next buffers' metadata if needed.
Phase 5	Update and flush TXC1. (Critical section 1 ends here)
	If needed, prepare 48-byte cell payload in local memory.

Table 11-8. OC-48 Algorithm - Pipestage 2

Phase	Description
Phase 1	Fetch TXC2 in local memory. (Critical section 2 starts here)
Phase 2	As an optimization, read the next dequeue request from scratch ring.
	Compute CRC on 48-byte payload.
	Update and flush TXC2. (Critical section 2 ends here)
	Write 52-byte cell (4-byte header and 48-byte payload) to TBUF.
Phase 3	Validate TBUF.
	Loop to pipestage 1, phase 2.

11.4.2 Quad OC-12—One page Summary

AAL5 TX is implemented in two ME's running independently.

For clarity, we divide the algorithm in two stages.

Table 11-9. Quad OC-12 Algorithm Pipestage 1

Phase	Description
Phase 1	Read dequeue request.
Phase 2	Check for null dequeue and setup CAM data.
Phase 3	Fetch TXC in local memory.
	Flush existing TXC to SRAM if necessary.
Phase 4	Read payload from DRAM.
	Update TXC.
	Pre-fetch next buffers' metadata if needed.
Phase 5	If needed, prepare 48-byte cell payload in local memory.

Table 11-10. Quad OC-12 Algorithm Pipestage 2

Phase	Description
Phase 1	Initialize some TXC fields for SOP.
Phase 2	As an optimization, read the next dequeue request from scratch ring.
	Compute CRC on 48-byte payload.
	Write 52-byte cell (4-byte header and 48-byte payload) to TBUF.
Phase 3	Validate TBUF.
	Loop to pipestage 1, phase 2.

11.4.3 OC-48—Second Level of Detail

Pipestage 1

Phase 1: Issue and wait on scratch ring I/O to read dequeue request.

Phase 2:

```
If dequeue request is null
    Follow null path.
Else
    Issue scratch I/O to read next available TBUF element.
    Setup CAM data.
```

Phase 3: (Critical section 1 starts here)

```
If SOP
    Using SOP buffers' meta-data, initialize TXC1.
Else
    If TXC1 a miss in CAM
        Fetch TXC1 from SRAM.
```

Phase 4:

```
If additional cell
    Goto end of this phase.
If current buffer is over
    Switch to pre-fetched buffer.
If next buffers' meta-data needs to be pre-fetched
    Issue SRAM I/O to read next buffers' meta-data.
If last cell of payload
    Compute cell payload (1-48 bytes).
If last cell in current buffer
    Set bit to free current buffer in pipestage 2.
If cell payload split across current and next buffer
    If next buffer has no more payload
        Set bit to free next buffer in pipestage 2.
    Issue DRAM I/O to read cell payload from next buffer.
Issue DRAM I/O to read cell payload from current buffer.
Update TXC1 fields to reflect packet payload read in this phase.
```

Phase 5:

```

If next buffers' meta-data was pre-fetched
    Update relevant TXC1 fields
If TXC1 update done
    Flush TXC1 to SRAM.
    If thread 7
        Issue inter-me signal to next me to indicate end of pipestage 1.
        (Critical section 1 ends here)
        Issue MSF I/O to read cells transmitted by fabric.
    Ctx-arb to allow thread 7 to signal next me as soon as possible.
If cell payload in current buffer is neither 40 nor 48 bytes,
    Prepare 48-byte cell payload in a thread specific location in local
    memory.

```

Pipestage 2**Phase 1:** (Critical section 2 starts here)

```

If SOP
    Initialize TXC2.
Else
    If TXC2 a miss in CAM
        Fetch TXC2 from SRAM.

```

Phase 2:

```

As an optimization, issue scratch ring I/O to read next dequeue request.
    If free current buffer bit set
Free current buffer.
    If free next buffer bit set
Free next buffer.
    Compute cells transmitted by code using available TBUF element number.
    Compute CRC on 48-byte cell payload in local memory.
        If end of payload, append 8-byte CPCS trailer.
        If end of payload, set AUU bit in 4-byte cell header.
        Store residue in TXC2.
If TXC2 update done
    Flush TXC2 to SRAM.
    If thread 7
        Issue inter-me signal to next me to indicate end of pipestage 2.
        (Critical section 2 ends here)
        Issue scratch I/O to write cells transmitted by fabric to
        scheduler microblock.
        If cells transmitted by code exceeds cells transmitted by fabric
        by a set threshold
            Stall until the difference falls back within threshold.
        Create the FPGA header (if the extended port addressing is used)
        Write 52-byte cell (4-byte header and 48-byte payload) and the FPGA
        header (if the extended port addressing is used) to TBUF.

```

Phase 3:

Validate TBUF.

Loop to pipestage 1, phase 2.

11.4.4 Quad OC-12—Second Level of Detail

Pipestage 1

Phase 1: Issue and wait on scratch ring I/O to read dequeue request.

Phase 2:

If dequeue request is null

 Follow null path.

Else

 Issue scratch I/O to read next available TBUF element.

 Setup CAM data.

Phase 3:

 If SOP

 Using SOP buffers' meta-data, initialize TXC.

 Else

 If TXC a miss in CAM

 Flush existing TXC to SRAM.

 Fetch TXC from SRAM.

Phase 4:

 If additional cell

 Goto end of this phase.

 If current buffer is over

 Switch to pre-fetched buffer.

 If next buffers' meta-data needs to be pre-fetched

 Issue SRAM I/O to read next buffers' meta-data.

 If last cell of payload

 Compute cell payload (1-48 bytes).

 If last cell in current buffer

 Set bit to free current buffer in pipestage 2.

 If cell payload split across current and next buffer

 If next buffer has no more payload

 Set bit to free next buffer in pipestage 2.

 Issue DRAM I/O to read cell payload from next buffer.

 Issue DRAM I/O to read cell payload from current buffer.

 Update TXC fields to reflect packet payload read in this phase.

Phase 5:

```

If next buffers' metadata was pre-fetched
    Update relevant TXC fields
If thread 3 or thread 7
    Issue MSF I/O to read cells transmitted by fabric.
If cell payload in current buffer is neither 40 nor 48 bytes,
    Prepare 48-byte cell payload in a thread specific location in local
    memory.

```

Pipestage 2**Phase 1:**

```

If SOP
    Initialize crc field in TXC.

```

Phase 2:

```

As an optimization, issue scratch ring I/O to read next dequeue request.
    If free current buffer bit set
        Free current buffer.
    If free next buffer bit set
        Free next buffer.
    Compute cells transmitted by code using available TBUF element number.
    Compute CRC on 48-byte cell payload in local memory.
        If end of payload, append 8-byte CPCS trailer.
        If end of payload, set AUU bit in 4-byte cell header.
    Store residue in TXC.
    If thread 3 or thread 7
        Issue scratch I/O to write cells transmitted by fabric to scheduler
        microblock.
        If cells transmitted by code exceeds cells transmitted by fabric by
        a set threshold
            Stall until the difference falls back within threshold.
    Create the FPGA header (if the extended port addressing is used)
    Write 52-byte cell (4-byte header and 48-byte payload) and the FPGA
    header (if the extended port addressing is used) to TBUF.

```

Phase 3:

```

Validate TBUF.
Loop to pipestage 1, phase 2.

```

11.5 Performance Analysis

Shown below are the performance metrics for the ATM TX microblock.

Average case is the most frequently (at least 97% of the time) taken critical path.

Worst case is the longest critical path.

11.5.1 OC-48

As seen by the transmit rate, the microblock meets OC-48 (5.66 million packets per second) rate, despite the fact that in the worst case, it exceeds the instruction cycle budget by 12 instructions. This is because the worst case only occurs when a cell crosses a buffer boundary and is spread across two buffers. This case is always preceded by the average case (cells that lie completely in one buffer) and for which the ATM TX microblock is well within budget. The extra instructions available in the average case allows the ATM TX microblock to catch up after handling the worst case.

Table 11-11. OC-48 Performance Analysis

Parameter	Average case	Worst case	Budget	Unit
Critical path	201	330	318	Cycles
Critical section 1	75	96	106	Cycles
Critical section 2	74	81	106	Cycles
Transmit rate	5.66	5.66	5.66	Mpps

Table 11-12 provides the breakup of the critical path cycles per stage in the average and (worst) case.

Table 11-12. OC-48 Critical Path Cycles per Stage

Stage	Non-CS	CS 1	Non-CS	CS 2	Non-CS	Non-CS	Total
Pipestage 1 Stage 2	11 (11)						011 (011)
Pipestage 1 Stage 3		18 (18)					018 (018)
Pipestage 1 Stage 4		38 (59)					038 (059)
Pipestage 1 Stage 5		19 (19)	19 (19)				038 (038)
Pipestage 1 Stage 5*			0 (101)				000 (101)
Pipestage 2 Stage 1				9 (9)			009 (009)
Pipestage 2 Stage 2				65 (72)	15 (15)		080 (087)
Pipestage 2 Stage 3						7 (7)	007 (007)
Total							201 (330)

Non-CS: Non-critical section CS: Critical section *: Cycles used to prepare cell in LM

11.5.2 Quad OC-12

Table 11-13. Quad OC-12 Performance Analysis

Parameter	Average case	Worst case	Budget	Unit
Critical path	187	313	212	Cycles
Transmit rate	5.66	5.66	5.66	Mpps

As seen by the transmit rate, the reference design meets quad OC-12 (5.66 million packets per second) rate.

The breakup of critical path cycles per stage in average and (worst) case is shown in [Table 11-14](#).

Table 11-14. Quad OC-12 Critical Path Cycles per Stage

Stage	Non-CS	CS 1	Non-CS ^a	CS 2	Non-CS ^b	Non-CS	Total
Pipestage 1 Stage 2	30 (30)						030 (030)
Pipestage 1 Stage 3			21 (21)				021 (021)
Pipestage 1 Stage 4			31 (66)				031 (066)
Pipestage 1 Stage 5			21 (19)				021 (019)
Pipestage 1 Stage 5 ^c			0 (100)				000 (100)
Pipestage 2 Stage 1					5 (5)		005 (005)
Pipestage 2 Stage 2					62 (65)		062 (065)
Pipestage 2 Stage 3						7 (7)	007 (007)
Total							187 (313)

- a. Critical section
b. Non-critical section
c. Cycles used to prepare cell in LM

[Table 11-15](#) provides the resource usage for the microblock:

Table 11-15. AAL5 TX Microblock Resource Usage

Parameter	Used	Budget	Unit
Control store	1700	4096	Instructions
Local memory	400	640	Long words
GP registers	27 ^a	32	
SRAM registers	16 ^a	16	
DRAM registers	16 ^a	16	
Signals	12 ^a	15	

- a. This is maximum usage during some execution path. It does not imply all these registers/signals are used all the time

11.5.3 Characterization Data

Table 11-16. AAL5 TX Microblock Characterization Data

Data	Value
General:	
Microblock Name	aal5_tx - oc48 and quad OC12 modes
Microblock Version Number	1.0
Implementation Language	microcode

Table 11-16. AAL5 TX Microblock Characterization Data (Continued)

Data	Value
Configuration Options use to gather this set of data	1. FIRST_AAL5_TX_ME
	2. SECOND_AAL5_TX_ME
	3. USE_IMPORT_VAR
	4. RAW_MODE_ENABLED
	5. ATM_OAM_ENABLED
	6. COUNTERS
	7. FPGA_HEADER
	8. IXP2400
	9. IXP2800
	10. TX_PHY_MODE == SPHY_1_32
	11. TX_PHY_MODE == SPHY_1_32
	12. TX_PHY_MODE == SPHY_4_8
	13. TX_PHY_MODE == SPHY_4_8_SHARED
	14. TX_PHY_MODE == MPHY_4
	15. AAL5_TX_NUMBER_OF_FP_MEs
	16. AAL5_TX_ME_INIT_MSf
	17. USE_ME_INIT_SIGNAL
	18. FLOW_CONTROL
Measurement Environment (tool settings)	SDK 3.5

Performance: (cycle counts, latencies, bandwidths)

		quad oc12
Instruction cycle count per packet (common case packet / worst case cycle count)	common case: 87, worst case: 313, budget: 212	
	oc48	
		common case: 201, worst case: 330, budget: 318
Common-case packet/path assumptions to be documented here		
Scratch Memory		
# of longwords read (for bandwidth calculations)	2	
# of longwords written (for bandwidth calculations)	0	
# and type of each atomic operation performed (for bandwidth calculations)	1	
SRAM		
# of longwords read	8	
# of longwords written	8	
# and type of each atomic operation performed (for bandwidth calculations)	0	
DRAM		
# of quadwords read	6	
# of quadwords written	0	
Other I/O instructions (i.e. MSF, PCI, CAP, and Hash)	2 msf[write], 1 sram[enq]	
List of dependent I/O accesses in the longest latency path		

Table 11-16. AAL5 TX Microblock Characterization Data (Continued)

Data	Value
Per-Microengine Resources:	
Control-Store Usage (# of instructions used)	OC48: 1795, quad OC12:1700
Local Memory Footprint (# of long words used)	400
Local Memory Configuration (shared, or per-context pointer)	shared and per-context
Local Memory - # of LM pointers used	2
GPR Usage – minimum, static usage (absolutes, static, globals)	
Transfer Reg. Usage – minimum, static usage	
Next Neighbor Reg. Usage – minimum, static usage	
Signal Usage – minimum, static usage	
CAM used? (yes or no)	Yes
Global Resources:	
Scratch footprint (# of longwords used) – constant or formula ...	128*4
SRAM footprint (# of longwords used) – constant or formula ...	AAL5_TX_CTX_SIZE*AAL5_TX_CTX_TOTAL
DRAM footprint (# of quadwords used) – constant or formula ...	None
Q-Array usage - # of queues used and if they need to be cached	None
CRC Unit used?	Yes
Hash Unit used? (yes or no)	No
MSF Usage Information:	
Media Bus Configuration	UTOPIA
RBUF, TBUF usage	64 byte TBUF element
CBus signals	
Other Information:	
Critical Section Length (compute cycles + memory accesses)	0
# of phases	7
Packet Metadata - fields read	
Packet Metadata - fields written	
Header - fields read	
Header - fields written	

Table 11-16. AAL5 TX Microblock Characterization Data (Continued)

Data	Value
Documentation:	
Thread Ordering Requirements	Yes
OS dependencies	VxWorks, Linux
Chip/hardware dependencies (i.e. Crypto Unit in 2850)	IXP2400, IXP2800
Tested on which SDK Release(s)	IXA SDK 3.5
Tested on hardware? Which hardware configuration?	Yes, Quad OC-12
Tested in which applications (not an all inclusive list)	Quad OC-12 and OC-48 IPv4
Possible Configuration Options	
Major limitations (i.e. Rx/Tx microblocks written to use entire ME)	
Packet Sequencing Issues (esp. in POT applications)	
Core Component or Interface requirements or dependencies	

11.6 Possible Optimizations

As seen in the previous section, the worst-case critical path is 340 cycles. To meet OC-48 rates we need 3 microengines running the AAL5 TX microblock as a functional pipeline. There may be configurations where this microengine allocation may not be desirable or even possible. This section explores how to use the AAL5 TX microblock in such configurations and well as how to reduce the instruction cycles in the critical path.

Table 11-17 shows the number of cycles taken in the worst case for some tasks performed in the critical path.

Table 11-17. Number of Cycles in Worst Case for Tasks Performed in Critical Path

No.	Task	Cycles
1	Inter-me signaling	12
2	Using two TXC's instead of one	18
3	Computing current buffer to use instead of using QM scratch ring buffer handle data ^a	26
4	Processing a cell split across buffers	61
5	Processing a packet that exceeds buffer size (includes cycles taken for tasks 3 and 4)	87

- a. The QM dequeues a buffer as many times as specified in its cell count field. CSIX RX sets the cell count for each buffer based on the payload size within the buffer. However, the payload size within the SOP buffer can change, if for example a microblock between CSIX RX and QM inserts some protocol specific headers. This may result in cell counts changing not only for the SOP buffer but all buffers for the packet. Currently the L2 encap microblock re-computes the cell count only for the SOP buffer such that the sum of all bufer cell counts equals the cell count for the packet. This however results in the QM getting a possibly incorrect cell count for a buffer. Thus, AAL5 TX cannot rely on the buffer handle field sent by the QM to compute the buffer address for the current cell. Instead, it computes the buffer address on its own.

All these tasks need not be performed in certain configurations.

For example, for OC-12 data rates, the cycle budget is 106×4 —that is, 424 cycles. Therefore we can run the AAL5 TX microblock on a single microengine. Running on one microengine eliminates the need for inter-ME signaling and two TXC's (one TXC does suffice since there is no functional pipeline and the TXC need not be split between the functional stages). This saves ~30 cycles in the critical path.

As another example, in an Ethernet to ATM design, each incoming Ethernet packet can fit entirely in a 2K buffer. This implies that the packet never exceeds one buffer. This also implies that no cell is split across buffers. This saves ~87 cycles in the critical path.

Therefore, depending on the requirements of an application, specific optimizations are possible. In some cases, the control flow within the code automatically reflects this reduced length. In others, explicit pre-processor switches may be needed to reduce the length. This implementation is generic in the sense that it handles all the tasks tabulated above.

Note: Future revisions will include pre-processor switches for compiling certain critical path tasks.

This section describes the low-level design and implementation of the Packet Transmit (TX) microblock for the IXP24XX MPHY-16/IXP28XX multiports (up to 16 ports) configuration.

12.1 Overview

A maximum of 16 ports are supported in IXP24XX MPHY-16/IXP28XX multiports Packet Transmit. In the supported multiports operation mode, the head of line blocking (HOLB) issues cannot be avoided and need to be explicitly addressed. This makes the overall algorithm much more complicated as compared to the SPHY and MPHY-4 in IXP24XX or single port operation in IXP28XX.

This Packet TX microblock runs on two microengines and handles up to 16 ports. These two ME runs independently, but the threads inside each ME uses folding mechanism to run in strict order of two phases. Each microengine handles some subset of the 16 ports (up to 8 ports for each ME, configured via a bit mask) and OC-24 (IXP24XX)/OC-96 (IXP28XX) data rates. Each microengine has a separate scratch ring on which it receives transmit requests from the upstream microblock (QM in IXP24XX, TX Helper in IXP28XX). The data rates on the 16 ports do not have to be equal and can be adjusted based on the weights in the Egress scheduler.

The TBUF elements function as a single large segment. All traffic for all ports funnel into this single segment. Partitioning the TBUF element into sixteen equal sized segments would potentially throttle ports that carry higher bandwidth. IXP24XX is a System Packet Interface Level 3 (SPI-3) compliance device. In IXP24XX, the Tx_MPHY_Status MSF register is designed to provide PHY status to help the software to schedule the transmission of packets to avoid HOLB (the head of line blocking). For each port, there are two bits Tx_Pending and Tx_Status in the Tx_MPHY_Status register. Tx_Pending=1 indicates a pending transmit activity for that port, then Tx_Status is indeterminate. When Tx_Pending=0 and Tx_Status=1, the port is ready to accept at least “n” mpackets of data. The size of “n” (referred to as PHY_DEPTH in this document) depends on the characteristics of the port PHY and size of the mpacket.

IXP28XX is a System Packet Interface Level 4 (SPI-4) compliance device. In IXP28XX, the Tx_Multiple_Port_Status_# MSF registers are designed to provide PHY status to help the software to schedule the transmission of packets to avoid HOLB, there are two adjacent bits to show the port PHY status for each port, for example, bit 1 and bit 0 in Tx_Multiple_Port_Status_0 is for port 0, bit 3 and bit 2 is for port 1, and so on. The following describes various example scenarios:

- Both MSB and LSB as 1 is reserved for framing or to indicate a disabled status link.
- MSB as 1 and LSB as 0 is “SATISFIED” status, which indicates that the corresponding port's transmit FIFO is almost full.
- MSB as 0 and LSB as 1 is “HUNGRY” status which means that up to MaxBurst2 16-byte blocks or the remainder of what was previously granted (whichever is greater) may be sent to the corresponding port until the next status update.
- Both MSB and LSB as 0 is “STARVING” status, which indicates that the buffer underflow is imminent in the corresponding PHY port and transfers for up to MaxBurst1 16-byte blocks

may be sent to the corresponding port until next status update. The values of MaxBurst1 and MaxBurst2 are the characteristics of the interface PHY and usually are programmable.

Based on polling the Tx_MPHY_Status/Tx_Multiple_Port_Status_0 register and the “n/MaxBurst2 and MaxBurst1” value for each port, a flow control vector is formed. This flow control vector, empty port vector (corresponding bit set to 1 for the port which has no packet pending in its packet queue in the local memory), and a round robin vector between ports are used to schedule the transmission of packets.

Each microengine receives transmit requests from its upstream microblock through a separate scratch ring. Since the TX requests are packet based, the microblock needs to segment the payload into mpackets, copy them into TBUFs and validate that TBUF to send this mpacket out. Since a long packet may span several buffers, the microblock caches information on two buffers per packet at any time. One is the current buffer being processed and the other is the next buffer in the chain.

In order to optimize the performance, the microblock transmits the packet in the tx_request received immediately—that is, the microblock does not first queue the received packet in the local memory queue to process later, under the following conditions:

- If the port on the tx_request is not flow-controlled (TX FIFO of the PHY is near full, the TX FIFO overflows if more mpackets are sent to the PHY)
- If there is no pending packets in the local memory queue of that port.

If the above conditions are not valid, the packet in the tx_request is put into the local memory queue of that port and processed later. The packet Transmit block can transmit a maximum of three mpackets for the same packet in one iteration. This reduces the total instruction counts spent on one mpacket and also boosts the performance of the microblock.

The Packet Transmit block informs the scheduler of how many packets are transmitted on a port via a reflector write. The scheduler keeps track of packets in flight and ensures that the number of packets scheduled but not transmitted does not exceed the local memory storage in the transmit microengine.

12.2 Assumptions and Dependencies

This microblock needs to handle the following two restrictions imposed by the IXP24XX/IXP28XX hardware:

- The DRAM[tbuf_wr, ...] instruction always accesses DRAM on a 8-byte or quad-word boundary
- The sum of the prepend length and payload length of a mpacket must be an integral multiple of 4 bytes (IXP24XX)/16 bytes (IXP28XX), except for a mpacket in which EOP bit in its transmit control word is set.

12.3 Data Structures

This section describes the data structures used for the Packet transmit microblock

12.3.1 Globals Stored in Absolute Registers

Table 12-1 shows the globals stored in absolute registers. The variables in Table 12-1 are stored in absolute GPRs. They are used to find a port on which a packet may be transmitted.

Table 12-1. Globals Stored in Absolute Registers

Global	Description
@ep	Empty ports. Each port occupies 1 bit, when set, it means that there is no packets queued in local memory for this port, initialized to 0xFFFF
@rp	Ports ready to process packet in tx request immediately, because there is no packets in the queue in the local memory, and it is not blocked by flow control. @rp = @ep AND not(@bfcf).
@vap	Virtual available ports. Each port occupies 1 bit, when set, it means that the packet queue in local memory for that port is not empty, initialized to 0x00000000
@vpp	Virtual port queue to pick next packet to transmit for the next iteration. Each port occupies one bit, when set, means that the virtual count for that port is not zero and that port is not blocked by flow control, @vpp = @vap AND not(@bfcf), initialized to 0x0000FFFF
@bfcf	Ports blocked by flow control. Each port occupies 1 bit, when set, it means that this port is blocked by PHY status, waiting port ready to transmit, set to 0x00000000 during initialization
@busy_port	Each port occupies 1 bit for IXP24XX, 2 bits for IXP28XX (using high bit) due to the way they report the PHY status, initialized to 0x00000000, when mpackets transmitted, credit to transmit mpackets without checking PHY status for the specific port reduced, when it reduced to a pre-selected low threshold, the bit is set, so the PHY status will be checked to update the PHY status for that port, if new credit is decided to granted to that port, this bit for that port is reset to 0.
@blocked_by_read_next_buf	Each port occupies 1 bit, when set, it means that this port is blocked by waiting next buffer meta data reading to complete, initialized to 0x0000. If the quotation of the buffer size divided by the (tbuf element size times Maximum mpacket transmitted in one run) is larger than the total thread number in one ME, this restriction can be removed.
@blocked_by_read_leftover	Each port occupies 1 bit, when set, it means that this port is blocked by waiting leftover reading to complete, initialized to 0x0000

12.3.2 Context Relative Thread Execution Status Flag

Table 12-2 shows the context relative thread execution status flag (exe_stat_flag)— various execution flags for a thread.

Table 12-2. Context Relative Thread Execution Status Flag

LW	Bits	Size	Field	Description
0	31:22	10		Reserved
	21:21	1	ESF_LAST_BUF_BIT	Flag means the buffer handled is the last buffer of a packet
	20:20	1	ESF_LEFTOVER_TO_TX	Flag to transmit leftover payload from previous buffer
	19:19	1	ESF_READ_NEXT_BUF_META	Flag to save next buffer meta data
	18:18	1	ESF_READ_LEFTOVER	Flag to save leftover to local memory
	17:17	1	ESF_SOP_MPKT_THREAD	Flag for the first mpacket to be a SOP mpacket

Table 12-2. Context Relative Thread Execution Status Flag (Continued)

	16:3	14		Reserved
	2:2	1	ESF_EOP_MPKT_THREAD	Flag for the last mpacket to be an EOP mpacket
	1:1	1	ESF_PICK_VIRTUAL_QUEUE	Flag for that a port is picked from virtual queue for next iteration
	0:0	1	ESF_FREE_BUF	Flag to free current buffer in the beginning of the next iteration

12.3.3 TX Request—One Long Word

Table 12-3 shows the transmit request received from the upstream microblock.

Table 12-3. Transmit Request—One Long Word

LW	Bits	Size	Field	Description
0	31:31	1		Valid bit
	30:28	3		Reserved
	27:24	4		Port number
	23:0	24		SOP meta handle >> 2

12.3.4 Packet Queue Structure for Each Port

Queue information for each port is located in two different locations for the flexibility to increase the queue size for each port and to use both local memory pointers LM_ACTIVE_0 and LM_ACTIVE_1 effectively.

The first 512 long words in local memory are used to store the packet queue entries for all ports. The upper 128 words are used to store the port status (queue head, queue tail, packet count in queue, packets transmitted, etc) of each port. Each packet queue entry is 4 long words. So if there are 8 ports on a microengine, then for each port we can queue up to 16 transmit requests (16 packets). The size for port status is 16 longwords for each port.

12.3.4.1 Packet Queue Entry for Each Port

Table 12-4 shows the packet queue entry for each port.

Table 12-4. Packet Queue Entry for Each Port

LW	Bits	Size	Field	Description
0	31:31	1	sop_flag	Whether the current buffer is the SOP buffer of the packet in queue entry 0
	31:30	1	read_next_buf_flag	Read next buffer meta data flag bit for the current buffer of queue entry 0
	29:24	6		Reserved
	23:0	24	cur_buf_handle	Buffer handle of the current buffer of queue entry 0
1	31:16	16	cur_buf_paylo_rmnd	Remaining bytes of the payload to be sent out in the current buffer of entry queue 0

Table 12-4. Packet Queue Entry for Each Port (Continued)

	15:0	16	cur_buf_rmnd_paylo_offset	Offset of the first byte in the remaining payload in the current buffer of queue entry 0
2	31:31	1		Reserved
	30:30	1	next_buf_read_next_buf_flag	Read next buffer meta data bit of the next buffer of queue entry 0
	29:24	6		Reserved
	23:0	24	next_buf_handle	Buffer handle of the next buffer of queue entry 0
3	31:16	16	next_buf_size	Buffer size of the next buffer of queue entry 0
	15:0	16	next_buf_offset	Offset of the next buffer of queue entry 0
4:7				Info for queue entry 1
8:11				Info for queue entry 2
12:15				Info for queue entry 3
16:19				Info for queue entry 4
20:23				Info for queue entry 5
24:27				Info for queue entry 6
28:31				Info for queue entry 7
32:35				Info for queue entry 8
36:39				Info for queue entry 9
40:43				Info for queue entry 10
44:47				Info for queue entry 11
48:51				Info for queue entry 12
52:55				Info for queue entry 13
56:59				Info for queue entry 14
60:63				Info for queue entry 15

12.3.4.2 Port Status Info for Each Port—16 Long Words

Table 12-5 shows the structure of port status for a port in the local memory, some location (LW0, LW1, LW2, LW3, and LW14) are used to store global constant to save instruction cycles and also resolve the lack of available GPRs.

Table 12-5. Structure of Port Status for a Port in the Local Memory

LW	Bits	Size	Field	Description
0	31:0	32	tbuf_base	Base to compute tbuf address
1	31:0	32	ind_ref_base	Base for indirect ref for dram[tbuf_wr...]
2	31:0	32	tcw0_sop_eop_base	Base for tcw 0 for sop-eop mpacket
3	31:0	32	tx_req_ring	Scratch ring for tx request

Table 12-5. Structure of Port Status for a Port in the Local Memory (Continued)

4	31:24	8	virtual_count	Virtual count of number of packets in queue for the port, it is the count used to determine whether or not to read scratch ring to get new transmit request, the main point is that there is no need to read scratch ring when there are mpackets available for transmit in queue, so keep the new tx request in the scratch ring, so those packet may be transmitted without being moved into local memory and gotten out of local memory later to save instruction cycles for critical path. The operation of virtual_count (each packet in queue always have value equal to 1, no matter how big the packet is) guarantees one port will not block other port transmission.
	23:20	4		Reserved
	19:16	4	pkt_in_queue	Packets in queue for this port
	15:8	8	tail_offset	Queue Tail for this port (Bytes offset within queue info)
	7:0	8	head_offset	Queue head for this port (Bytes offset within queue info)
5	31:0	32	pkt_txed	Packets transmitted in this port
6	31:0	32	fc_credit_remaining	Remaining mpackets to transmit without checking port status.
7	31:0	32	fc_ready_credit	Number of mpackets to transmit without checking port status, when port status bit is 1 and pending bit is 0 for IXP24XX. This value depends on PHY hardware.
	31:0	32	fc_hungry_credit	Number of mpackets to transmit without checking port status, when port status is “hungry” for IXP28XX. This value depends on PHY hardware.
8	31:0	32	fc_starving_credit	Number of mpackets to transmit without checking port status, when port status is “starving” for IXP28XX. This value depends on PHY hardware.
9	31:31	1	leftover_flag_bit	Some bytes leftover from previous buffer, need to be transmitted before the payload of this buffer
	30:24	7	leftover_length	Length of leftover to next buffer in bytes, max 3 bytes for IXP24XX, and 15 bytes for IXP28XX
	23:0	24	next_next_buf_hndl	Temporary storage of next buffer handle of next buffer, before next buffer becomes current buffer
10	31:0	32	leftover_lw0	First longword of payload leftover by previous buffer
11	31:0	32	leftover_lw1	Second longword of payload leftover by previous buffer (only for IXP28XX)
12	31:0	32	leftover_lw2	Third longword of payload leftover by previous buffer (only for IXP28XX)
13	31:0	32	leftover_lw3	Fourth longword of payload leftover by previous buffer (only for IXP28XX)
14	31:0	32	tbuf_valid_base	Base to compute tbuf validation address
15	31:0	32	tx_requests_discarded	tx requests discarded for the port to prevent the port packet queue from overflow when “CHECK_PORT_QUEUE_OVERFLOW” option enabled

12.3.5 Output Transmit Control Word—2 Long Words

Table 12-6 shows the format of the output transmit control word.

Table 12-6. Output Transmit Control Word—2 Long Words

LW	Bits	Size	Field	Description
0	31:24	8	payload_len	Payload length in bytes
	23:21	3	prepend_offset	Prepend offset in bytes
	20:16	5	prepend_len	Prepend length in bytes
	15:13	3	payload_offset	Payload offset in bytes
	12:12	1		Reserved
	11:11	1	skip	Skip
	10:10	1	err	ERR
	9:9	1	sop	SOP
	8:8	1	eop	EOP
	7:4	4		Reserved
	3:0	4	channel	Channel
1	31:0	32		Reserved

12.3.6 Statistics

Table 12-7 shows the statistics (32-bit counters in SRAM) on a per port basis, which enables the microblock to provide information to the XScale core component to maintain 64-bit version of these counters.

Table 12-7. 32-bit Statistics Counters in SRAM

Counter	Offset from base for port
Packets Transmitted	0x0
Packets Dropped	0x4
Bytes Transmitted	0x8
Bytes Dropped	0xc

Note: Will be provided in the next release of this microblock.

12.4 Design

The Packet Transmit block in this design runs on two microengines, each of which handles a mutually exclusive subset of a maximum of 16 ports in the system. Each microengine handles traffic up to OC-24 (IXP24XX)/OC-96 (IXP28XX) rates. The upstream microblock sends transmit requests to each TX microengine on a different scratch ring.

This Packet Transmit microblock processes packet transmission between [Picked_from_tx_request](#) and [Picked_from_virtual_queue_or_picked_none](#) path depend up on whether a valid `tx_request` is received in the previous iteration.

12.4.1 Picked_from_tx_request

A valid `tx_request` is received in the previous iteration. If there is no pending packet/packets in the local memory queue for the port in the `tx_request` and this port is not flow-controlled, this Packet Transmit block will start to transmit the packet in the `tx_request`, if this packet can be fitted into 3 mpackets, than it can be transmitted without queuing the remaining in the local memory queue, if it is longer than 3 mpackets, the remaining after 3 mpackets will be queued in the local memory queue for further processing. In the case that either there is pending packet in the local memory queue for the port in `tx_request` or the port in the `tx_request` is flow-controlled, this packet in the `tx_request` is not transmitted and put into the local memory queue of that port, and this Packet TX block tries to select one packet from other port to transmit based on round-robin base from the local memory queue for ports which are not flow-controlled.

At the end of phase 1 in this path, if a port from virtual queue (port/ports with pending packet/packets in local memory and not-flow-controlled) is available, then no `tx_request` read from the scratch ring is made and that port is selected as a candidate port to be transmitted in the next iteration and a flag bit is set, the point behind this algorithm is to consume any pending packet/packets in local memory queue as much as possible before we get another new `tx_request` from scratch ring, so increase the chance that any packet from `tx_request` will be transmitted immediately without first queued into local memory and read back to process in future.

If a `tx_request` is read, at the end of phase 1, a check is made to decide whether this read results a valid `tx_request` at the end of phase 2.

- For valid `tx_request` the execution is branched to [Figure 12-1, “picked_from_tx_request” on page 225](#) for the next iteration
- For invalid `tx_request` and for no `tx_request` read, the execution is branched to [Figure 12-2, “picked_from_virtual_queue_or_picked_none Path” on page 227](#).

[Figure 12-1](#) shows the high-level design for `picked_from_tx_request` path.

Figure 12-1. picked_from_tx_request

```

{
phase_1#:

    wakeup_next_thread();
    sig_mask_1=default_sig_mask_1;

    if (mpacket_number)
        validate_mapcket_mapckets_in_previous_iteration();

    if (ESF_FREE_BUF_set_in_exe_stat_flag)
        free_buffer_for_packet_in_previous_iteration();

    mpacket_number=0;
    exe_stat_flag=0;

    if (port in tx_request ready to transmit packet in tx_request)
    {
        calculate_mpacket_size_up_to_three_mapckets();
        if (packet longer than three mapckets)
            save_remaing_into_local_memory();
    }
    else
    {
        save_packet_in_tx_request_into_local_memory();
        f(pending_packets_in_lm_for_ports_not_flow_controlled)
        {
            select_port_based_on_round_robin();
            calculate_mpacket_size_up_to_three_mapckets();
            update_queue_info_in_lm_for_the_port();
        }
    }

    if (virtual_port_available)
        set_ESF_PICK_VIRTUAL_QUEUE_in_exe_stat_flag();
    else
        read_tx_request();

    local_csr_wr[ACTIVE_CTX_WAKEUP_EVENTS, sig_mask_1]
    ctx_arb[--]

```

Figure 12-1. picked_from_tx_request (Continued)

```

phase_2#:
    wakeup_next_thread();
    sig_mask_2=default_sig_mask_2;

    if(mpacket_number)
        move_payload_to_tbuf_and_prepare_tcw();

    if(ctx == 7)
        update_flow_control_info_based_on_ports_status()

    if(ESF_PICK_VIRTUAL_QUEUE_set_in_exe_stat_flag)
        br[virtual_queue_picked#]

    if(!valid_tx_request)
        br[none_picked#]

    read_sop_meta_data_for_packet_in_tx_request();

    local_csr_wr[ACTIVE_CTX_WAKEUP_EVENTS, sig_mask_2]
    ctx_arb[--], br[picked_from_tx_request_path]

virtual_queue_picked#:
none_picked#:
    local_csr_wr[ACTIVE_CTX_WAKEUP_EVENTS, sig_mask_2]
    ctx_arb[--], br[picked_from_virtual_queue_or_picked_none_path]
}

```

12.4.2 Picked_from_virtual_queue_or_picked_none

This path can be divided into separate `picked_from_virtual_queue` path and `picked_none` (no port available from virtual queue and no valid `tx_request` received in the previous iteration) path. However, `picked_from_virtual_queue` path and `picked_none` are combined into one path to fit the whole microblock into 4 K microstore.

If a port is selected from the virtual queue in the previous iteration, based on the flag bit, then the first pending packet from the local memory queue for the selected port is transmitted:

- if there is one or more pending packet for the port (the pending packet may be transmitted out by other thread in the previous iteration)
- the port is not flow-controlled (the port may become flow-controlled)

Otherwise, the Packet TX tries to select a packet from a different port to transmit, based on round-robin base from the local memory queue for ports which are not flow-controlled.

Same algorithm as in [Picked_from_tx_request](#) path is used to determine whether a `tx_request` should be read and whether the read is valid to determine the path for the next iteration.

Figure 12-2 shows the high-level design for `picked_from_virtual_queue_or_picked_none` path.

Figure 12-2. `picked_from_virtual_queue_or_picked_none` Path

```
{
phase_1#:
    wakeup_next_thread();

    sig_mask_1=default_sig_mask_1;

    if(mpacket_number)
        validate_mapcket_mapckets_in_previous_iteration();

    if(ESF_FREE_BUF_set_in_exe_stat_flag)
        free_buffer_for_packet_in_previous_iteration();

    mpacket_number=0;

    if(ESF_PICK_VIRTUAL_QUEUE_set_in_exe_stat_flag)
    {
        exe_stat_flag=0;
        if(pending_packet_still_available_and_port_not_flow_controlled)
        {
            calculate_mpacket_size_up_to_three_mapckets();
            update_queue_info_in_lm_for_the_port();
        }
        else
        {
            if(pending_packets_in_lm_for_ports_not_flow_controlled)
            {
                select_port_based_on_round_robin();
                calculate_mpacket_size_up_to_three_mapckets();
                update_queue_info_in_lm_for_the_port();
            }
        }
    }
    else
    {
        exe_stat_flag=0;
        if(pending_packets_in_lm_for_ports_not_flow_controlled)
        {
            select_port_based_on_round_robin();
            calculate_mpacket_size_up_to_three_mapckets();
            update_queue_info_in_lm_for_the_port();
        }
    }

    if(virtual_port_available)
        set_ESF_PICK_VIRTUAL_QUEUE_in_exe_stat_flag();
    else
        read_tx_request();

    local_csr_wr[ACTIVE_CTX_WAKEUP_EVENTS, sig_mask_1]
    ctx_arb[--]
```

Figure 12-2. picked_from_virtual_queue_or_picked_none Path (Continued)

```

phase_2#:

    wakeup_next_thread();
    sig_mask_2=default_sig_mask_2;

    if(mpacket_number)
        move_payload_to_tbuf_and_prepare_tcw();

    if(ctx == 7)
        update_flow_control_info_based_on_port_status();

    if(ESF_PICK_VIRTUAL_QUEUE_set_in_exe_stat_flag)
        br[virtual_queue_picked#]

    if(!valid_tx_request)
        br[none_picked#]

        read_sop_meta_data_for_packet_in_tx_request();

    local_csr_wr[ACTIVE_CTX_WAKEUP_EVENTS, sig_mask_2]
    ctx_arb[--], br[picked_from_tx_request_path]

virtual_queue_picked#:
none_picked#:
    local_csr_wr[ACTIVE_CTX_WAKEUP_EVENTS, sig_mask_2]
    ctx_arb[--], br[picked_from_virtual_queue_or_picked_none_path]
}

```

The goal of this design is to always transmit POS min-packet through [Picked_from_tx_request](#) path to get the optimized performance by eliminating the instructions cycles to save the packet into the local memory queue first and read back to transmit later.

The information for each packet in the local memory queue includes the meta data of the current buffer being handled and the meta data of the next buffer in the buffer chain. The meta data for the next buffer is read while the current buffer is being processed. After all the payload in the current buffer have been processed, the next buffer becomes the current buffer. To prevent the next buffer from becoming the current buffer before all the necessary meta data information is available for the next buffer, a bit vector @blocked_by_read_next_buf of ports waiting on the next buffer meta data read is maintained.

Another restriction is that the sum of the prepend length and payload length of a mpacket must be an integral multiple of 4 bytes (IXP24XX)/16 bytes (IXP28XX), except for a mpacket in which EOP bit in the transmit control word is set. To comply with this restriction, a maximum of 3 bytes (IXP24XX)/15 bytes (IXP28XX) at the end of a buffer is saved in local memory and transmitted with the payload of the first mpacket of the next buffer for the packet. To prevent a port from being scheduled before these leftover bytes are saved in local memory, a bit vector @blocked_by_read_leftover of ports waiting for completion of the saving of the *leftover* bytes is maintained.

Also, in the calculation of the mpacket size, the microblock ensures that no non-EOP mpacket is required to set .SKIP bit in the transmit control word, as the sum of prepend and payload is less than 4 bytes/16 bytes.

12.5 Performance Analysis

The Packet Transmit microblock uses two microengines, each of which handles up to OC-24 (IXP24XX)/OC-96 (IXP28XX) rates of traffic. Tables 12-8 and 12-10 shows the performance analysis for IXP24XX and IXP28XX Packet Transmit.

12.5.1 Performance Analysis for IXP24XX

For the IXP24XX case, the available cycle count budget to handle a POS min-packet is 97×2 cycles. Table 12-8 shows the instruction estimate for this Packet Transmit microblock to handle a POS min-packet in the POS min-packet code path.

Table 12-8. Instruction Estimate for IXP24XX Packet Transmit Microblock

Component	POS Min-packet Cycle count
Phase 1 (all threads)	60
Phase 2 (thread 0 to 6)	41
Phase 2 (thread 7, update flow control info based on Tx_MPHY_Status read in phase 1)	61
Phase 2 (average of all threads)	44
Total	104 (Available budget is 194)

Table 12-9 shows the I/O operations performed in different phases of the microblock for POS min-packet. Each thread has an available I/O latency of $97 \times 8 \times 2$ cycles to handle a min sized POS packet.

Table 12-9. I/O Operations Performed in Different Phases

I/O operations	Phase
MSF write of 2 longwords to validate TBUF and start transmit of the packet of the previous iteration	1
Sram enqueue operation to free buffer of the packet of the previous iteration	1
Scratch read of transmit request (one longword)	1
Scratch atomic operation to get the next available TBUF element	1
MSF read of Tx_Sequence_0 (one longword)	1
MSF read for Tx_MPHY_Status (one longword)	1
Dram to tbuf write of 48 bytes	2
SRAM read of new packet's meta data (three longwords)	2

12.5.2 Performance Analysis for IXP28XX

For the IXP28XX, the available cycle count budget to handle a POS min-packet is 57×2 cycles. Table 12-10 shows the instruction estimate for this Packet Transmit microblock to handle a POS min-packet in the min-packet code path.

Table 12-10. Instruction Estimate for IXP28XX Packet Transmit Microblock

Component	POS Min-packet Cycle count
Phase 1 (all threads)	62
Phase 2 (thread 0 to 6)	41
Phase 2 (thread 7, update flow control info based on Tx_Multiple_Port_Status_0 read in phase 1)	67
Phase 2 (average of all threads)	45
Total	107 (Available budget is 114)

Table 12-11 shows the I/O operations performed in different phases of the microblock for the POS min-packet. Each thread has an available I/O latency of $57 \times 8 \times 2$ cycles to handle a min sized POS packet.

Table 12-11. I/O Operations Performed in Different Phases of the Microblock

I/O operations	Phase
MSF write of 2 longwords to validate TBUF and start transmit of the packet of the previous iteration	1
Sram enqueue operation to free buffer of the packet of the previous iteration	1
Scratch read of transmit request (one longword)	1
Scratch atomic operation to get the next available TBUF element	1
MSF read of Tx_Sequence_0 (one longword)	1
MSF read for Tx_Multiple_Port_Status_0 (one longword)	1
Dram to tbuf write of 48 bytes	2
SRAM read of new packet's meta data (three longwords)	2

12.5.3 Characterization Data

Table 12-12. Packet TX-Multiports Microblock Characterization Data

Data	Value
General:	
Microblock Name	PACKET_TX_16PORTS
Microblock Version Number	1.0
Implementation Language	microcode

Table 12-12. Packet TX-Multiports Microblock Characterization Data (Continued)

Data	Value
Configuration Options use to gather this set of data	For first TX microengine: 1. THIS_ME=PACKET_TX_FIRST_ME 2. SCHEDULER_ME=0x05 3. ADD_L2_HEADER 4. ETHERNET_TX For second TX microengine: 1. THIS_ME=MPHY16_PACKET_TX_SECOND_ME 2. SCHEDULER_ME=0x05 3. ADD_L2_HEADER 4. ETHERNET_TX
Measurement Environment (tool settings)	

Performance: (cycle counts, latencies, bandwidths)	
Instruction cycle count per packet (common case packet / worst case cycle count)	First TX microengine: 107 (available budget: 114) Second TX microengine: 107 (available budget: 114)
Common-case packet/path assumptions to be documented here	Assumptions <ul style="list-style-type: none"> This is a two ME TX microblock, each ME supports up to 8 ports. The two MEs are exactly the same, except the first ME initializes MSF interface. Scheduler will schedule TX requests based on how many packets has been transmitted out of MSF, to prevent overflowing of TX requests in local memory queue for each port. All numbers reported are for worst-case for 10x1 Gbs Ethernet normal path.
Scratch Memory	
# of longwords read (for bandwidth calculations)	2
# of longwords written (for bandwidth calculations)	0
# and type of each atomic operation performed (for bandwidth calculations)	0
SRAM	
# of longwords read	Meta data: 5, L2Table: 4
# of longwords written	0
# and type of each atomic operation performed (for bandwidth calculations)	0
DRAM	
# of quadwords read	6
# of quadwords written	6
Other I/O instructions (i.e. MSF, PCI, CAP, and Hash)	5
List of dependent I/O accesses in the longest latency path	0

Table 12-12. Packet TX–Multiports Microblock Characterization Data (Continued)

Data	Value
Per-Microengine Resources:	
Control-Store Usage (# of instructions used)	First TX ME: 3739, Second TX ME: 3706
Local Memory Footprint (# of long words used)	640 (for eight ports for each TX ME)
Local Memory Configuration (shared, or per-context pointer)	per-context pointer
Local Memory - # of LM pointers used	2
GPR Usage – minimum, static usage (absolute, static, globals)	
Transfer Reg. Usage – minimum, static usage	
Next Neighbor Reg. Usage – minimum, static usage	0
Signal Usage – minimum, static usage	0
CAM used? (yes or no)	No
Global Resources:	
Scratch footprint (# of longwords used) – constant or formula ...	0
SRAM footprint (# of longwords used) – constant or formula ...	0
DRAM footprint (# of quadwords used) – constant or formula ...	0
Q-Array usage - # of queues used and if they need to be cached	0
CRC Unit used?	
Hash Unit used? (yes or no)	no
MSF Usage Information:	
Media Bus Configuration	SPI4 POS TX / SPI3 MPHY16 POS TX
RBUF, TBUF usage	TBUF
CBus signals	-
Other Information:	
Critical Section Length (compute cycles + memory accesses)	0
# of phases	First TX ME: 2, Second TX ME: 2
Packet Metadata - fields read	bfferSize, payloadOffset, nextBufferHandle, nextHopId
Packet Metadata - fields written	-
Header - fields read	
Header - fields written	

Table 12-12. Packet TX-Multiports Microblock Characterization Data (Continued)

Data	Value
Documentation:	
Thread Ordering Requirements	
OS dependencies	VxWorks, Linux
Chip/hardware dependencies (i.e. Crypto Unit in 2850)	2800/2400
Tested on which SDK Release(s)	IXA SDK 3.5
Tested on hardware? Which hardware configuration?	Yes, IXDP 2400 and IXDP 2800
Tested in which applications (not an all inclusive list)	several SDK 3.1 applications
Possible Configuration Options	
Major limitations (i.e. Rx/Tx microblocks written to use entire ME)	None
Packet Sequencing Issues (esp. in POT applications)	None
Core Component or Interface requirements or dependencies	ATM/POS TX and Ethernet TX core components

Queue Manager

The Queue Manager microblocks include the following:

- [Chapter 13, “Queue Manager For OC-48 Microblock”](#)
- [Chapter 14, “Queue Manager For OC-192 Microblock”](#)
- [Chapter 15, “Packet Queue Manager Microblock”](#)
- [Chapter 16, “ATM Queue Manager Microblock”](#)

The Queue Manager manages the queuing hardware assist (Q-Array) on the IXP2400/2800. It provides the following basic functionality:

- Enqueue of packets, where the enqueue request come from an upstream microblock
- Dequeue of cells or packets, where the dequeue requests come from a scheduler microblock
- Sending a transmit request for the dequeued cell or packet to the downstream transmit microblock

In addition, it may also support:

- Sending queue transition messages to the scheduler
- Writing queue state information required by WRED to SRAM
- Dropping large packets that span multiple buffers

There are several versions of the Queue Manager microblock on the SDK, which differ based on mode of processing (cell or packet), data rate (OC48 or OC-192) and queue transition handling. Some of these may be combined into a single code base in a subsequent revision of the SDK.

The different Queue Manager microblocks are designed to either a 57 cycle budget for OC-192/10G data rates on the IXP2800 or a 97 cycle budget for OC-48 data rates. To meet the instruction cycle budget, the OC-192 queue manager does not generate transition messages. The following summarizes the queue manager microblocks on the SDK:

Queue Manager	Description	Usage	Cycle Budget
OC-48 Cell QM	Cell mode, Enqueues packets, dequeues cells.	IXP2400 ingress on the CSIX fabric side	97
OC-48 Packet QM	Packet mode, enqueues and dequeues packets	IXP2400 egress on the POS and Ethernet output interfaces	97
OC-48 ATM QM	Cell mode (for TM 4.1) Requires messages to shaper microblock on every enqueue and dequeue	IXP2400/IXP2800 ATM	105
OC-48 DiffServ QM	Packet mode (for DiffServ). Writes queue information required by WRED to SRAM	IXP2400 egress on the POS and Ethernet output interfaces	97
OC-192 QM	Supports Cell and Packet mode. Does not send queue transition messages, or handle dropping large frames or writing WRED information. All these are handled by other microblocks	IXP2800 ingress and egress on CSIX, POS and Ethernet interfaces	57

Queue Manager For OC-48 Microblock

13

13.1 Overview

The Queue Manager (QM) is implemented as a microblock running on a single microengine.

Note: Since the QM is not combined with any other microblock on the same thread of execution, there is no need for a dispatch loop.

The QM is responsible for performing enqueue and dequeue operations on the transmit queues which are implemented using the hardware SRAM link lists. It accepts enqueue requests from the packet processing microengines via a scratch ring. Dequeue requests come from the transmit scheduler microengine.

The Queue Manager uses the Q-Array hardware in the SRAM controller to support link lists. The queue descriptors for the queues are maintained in SRAM. 16 queue descriptors are cached in the Q-Array in the SRAM controller. The Queue Manager uses the CAM to maintain this cache of queue descriptors. When an enqueue operation comes in, the queue manager checks the CAM to see if the queue descriptor for this queue is cached in local memory. If it is, then it simply performs the enqueue operation. If not, the LRU (Least Recently Used) queue descriptor is evicted from the Q-Array and written back to SRAM. Then the queue descriptor specified in the enqueue operation is read into the Q-Array cache and the enqueue operation is performed. A similar algorithm is used for the dequeue operation. In order to meet the performance budget, the Queue Manager must handle in the worst case, one enqueue and one dequeue operation every 88 cycles (min packet time).

To maintain coherence, the threads on the QM microengine execute in strict order using local inter-thread signaling.

Whenever an enqueue results in the queue state going from empty to non-empty or a dequeue operation results in the queue state going from non-empty to empty, the Queue Manager sends a message to the transmit scheduler via a Next Neighbor Ring. Also after every dequeue operation, the QM passes a transmit request via a scratch ring to the Packet TX microblock. For the 16 queue descriptors cached in the Q-Array, the QM keeps track of the number of packets in the queue using local memory.

The Queue Manager operates in two modes—cell mode and packet mode. The cell mode is used to transmit into CSIX and ATM media while the packet mode is used for transmit into POS and Ethernet media. In the cell mode, the QM enqueues packets and dequeues cells, while in the packet mode the QM enqueues and dequeues complete packets.

13.2 Assumptions/Dependencies

- The Queue Manager assumes that Queue Array entries 0..15 are reserved for the QM cache.
- Even though the same algorithm can be applied for SRAM rings too, this implementation supports only link lists.
- Depending on the scheduling algorithm used in the scheduler microengine, the Queue Manager may need to send the packet size to the scheduler along with the queue transition messages.

13.3 Data Structures

This section describes the data structures relevant to the Queue Manager.

13.3.1 Queue Descriptor

Figure 13-1. Format of Queue Descriptor

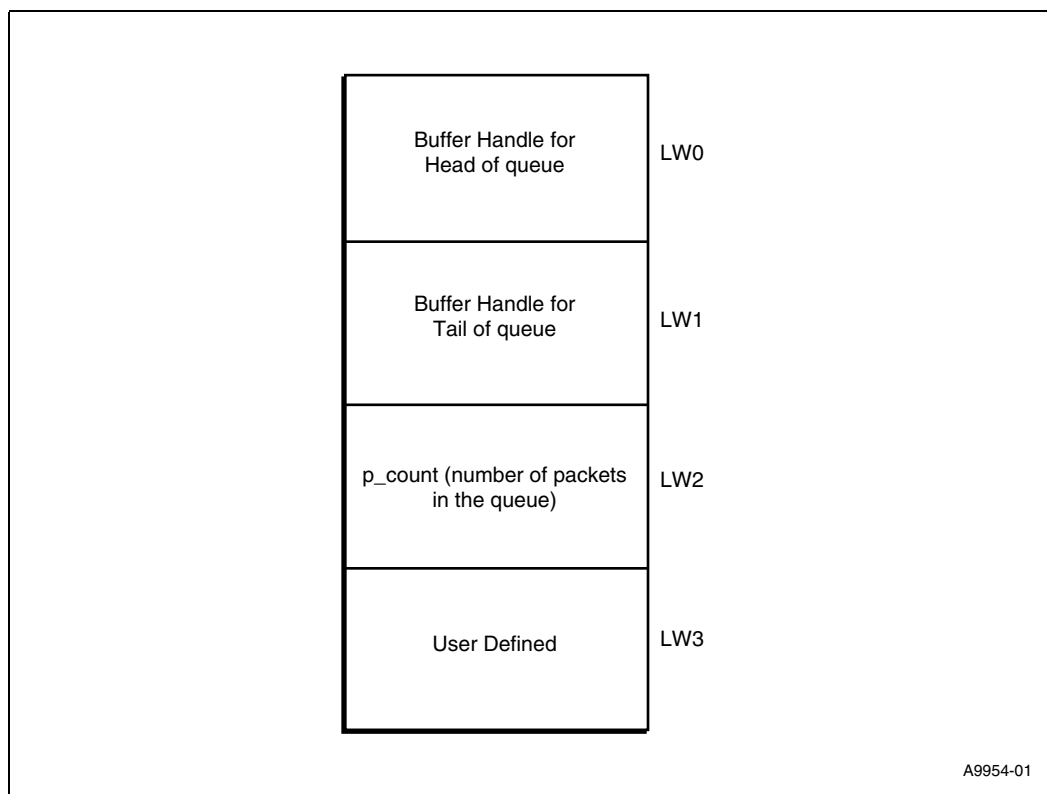


Figure 13-1 shows the format of queue descriptor as stored in SRAM. When the queue is referenced, the Q-Array moves the first three words (or four depending on how the Q-Array is configured) into the SRAM controller. The Q-Array can cache up to 64 queue descriptors per SRAM channel. However, for the Queue Manager we uses only 16 entries. The remaining may be used for buffer free lists, SRAM rings, etc.

Note: Even though SRAM is byte addressed, the Q-Array operations are all based on long words.

13.3.2 Packet Counts in Local Memory

The Queue Manager stores the current packet count for the cached queues in local memory. 16 words are used, each representing the packet count of one entry in the Q-Array cache. The packet count is initialized when the entry is read into the Q-Array (the `sram[rd_qdesc]` instruction returns the value from the queue descriptor stored in SRAM). Subsequently, the packet count is incremented/decremented as appropriate during the enqueue/dequeue operation.

13.4 Build Switches

Table 13-1 describes the compile time options used in the Queue Manager.

Table 13-1. Compile Time Options Used in the Queue Manager

Symbol	Description
PACKET_MODE	Set this build switch to use the Queue Manager in Packet Mode. Default is cell mode.
TM4_1	This build switch is used when the Queue Manager is used with the TM 4.1 Shaper/Scheduler.
TX_PHY_MODE	This build switch is set to SPHY_1_32, SPHY_4_8, MPHY_4 or MPHY_16. In the SPHY_4_8 or MPHY_4 case, the QM writes the transmit request to 4 scratch rings one per port.

13.5 Design

The QM uses eight threads running on a single microengine to handle OC-48 data rates. Every 88 cycles (min packet time) each thread handles in the worst case:

- One enqueue request from the packet processing code
- One dequeue request from the scheduler
- An enqueue transition
- A dequeue transition (or invalid dequeue).

The Queue Manager uses the Q-Array hardware in the SRAM controller to support link lists. The queue descriptors for the queues are maintained in SRAM. 16 queue descriptors are cached in the Q-Array in the SRAM controller. The Queue Manager uses the CAM to maintain this cache of queue descriptors. When an enqueue operation comes in, the queue manager checks the CAM to see if the queue descriptor for this queue is cached in local memory. If it is, then it simply performs the enqueue operation. If not, the LRU (Least Recently Used) queue descriptor is evicted from the

Q-Array and written back to SRAM. Then the queue descriptor specified in the enqueue operation is read into the Q-Array cache and the enqueue operation is performed. A similar algorithm is used for the dequeue operation.

The design has two phases and requires the threads to execute in strict order in each phase. The enqueue and dequeue requests are interleaved between the two phases to even out the I/O operations. So the enqueue request is handled in the first phase, while the dequeue request is handled in the second phase.

Whenever an enqueue results in the queue state going from empty to non-empty or a dequeue operation results in the queue state going from non-empty to empty, the Queue Manager sends a message to the transmit scheduler via a Next Neighbor Ring. Also after every dequeue operation, the QM passes a transmit request via a scratch ring to the Packet TX microblock. For the 16 queue descriptors cached in the Q-Array, the QM keeps track of the number of packets in the queue using local memory.

Figures 13-2, 13-3, 13-4 and 13-5 show the basic functionality in four phases. The initialize phase is followed by phases one, two and three which is actually the start of phase one of the next round-loop unroll.

Figure 13-2. Ingress Queue Manager: Phase Initialize

Initialize

Init memory and QArray's. All threads except thread 0, wait on previous thread signal. signal next thread;
read enqueue request from scratch ring. wait(enqueue scratch sig, previous thread signal);

Figure 13-3. Ingress Queue Manager: Phase 1

Phase 1

```
signal next thread
read dequeue request from scratch ring.
if( valid enq request )
{
    if( ! enqueue request is for drop queue )
    {
        check CAM;
        if( CAM hit )
        {
            issue enqueue
            Wait(deq scratch signal, previous thread sig) ;
        }
        else // CAM miss
        {
            evict LRU from CAM. Read new Queue Descriptor from SRAM
            Issue enqueue
            Wait(deq scratch signal, QD read signal, previous thread sig) ;
        }
    }
    else // its a drop queue enqueue request
    {
        enqueue to drop queue
        update local memory count.
        Wait(deq scratch signal, previous thread sig) ;
    }
}
else // no enqueue
    Wait( deq scratch signal, previous thread sig) ;
```

A9955-01

Figure 13-4. Ingress Queue Manager: Phase 2

Phase 2

```

signal next thread
set up message for enqueue transition.
read dequeue request from scratch ring.
if( valid dequeue request )
{
    check CAM
    If( CAM hit )
    {
        issue dequeue
        Wait(deq sig, enq scratch signal, previous thread sig) ;
    }
    else // CAM miss
    {
        evict LRU from CAM. Read new Queue Descriptor from SRAM
        issue dequeue
        Wait(deq sig, enq scratch signal, QD read signal, previous thread sig) ;
    }
}
else // no dequeue
{
    check local memory for drop queue, queue count;
    if( LM count !=zero )
    {
        issue dequeue from drop queue
        decrement local memory count ;
        Wait(deq sig, enq scratch signal, previous thread sig) ;
    }
    else // no dequeue for drop queue
        Wait( enq scratch signal, previous thread sig) ;
}

```

A9956-01

Figure 13-5. Ingress Queue Manager: Phase 3

Phase 3 (Actually start of Phase 1 of next round - loop unroll)

```

if( valid dequeue request )
{
    set up message for dequeue transition ;
    send the message to scheduler ;
    write the dequeued buffer to TX scratch ring ;
}
else
{
    enqueue the dequeued buffer from drop queue to FREE LIST ;
}
jump to phase 1 ;

```

A9957-01

13.6 Differences Between Cell and Packet QM

The following are the significant differences between the Cell and Packet Queue Managers.

13.6.1 Q-Array Hardware Configuration

The QM for the POS/Ethernet egress pipeline works in packet mode while the ingress QM works in cell mode. This means that while the egress QM dequeues an entire packet and sends it to the transmit block, the ingress QM dequeues a cell in a packet and sends it for transmit.

The packet mode QM configures the SRAM Q-Array CSR's to ignore the cell count field in the buffer handle. The EOP bit still needs to be set for the Q-Array hardware to accurately keep track of the number of packets in a queue.

13.6.2 Hierarchical Queuing

The packet mode QM queues the packets using the `packet_next` field in the meta data. However the SRAM offset in the buffer handle for the packet points to the start of the metadata, which is the `buffer_next` field. To work around this, the QM moves the offset in the handle to point to the `packet_next` field before the enqueue operation. When the packet is dequeued, the QM moves the offset in the handle to point back to the start of the metadata before sending it for transmit.

13.6.3 Dequeue Response

The ingress cell mode QM sends a message to the scheduler only in the case of an invalid dequeue or if a queue transitions from empty to non-empty or vice-versa. The egress packet mode QM however sends a response message to the scheduler for every dequeue request. The response message contains the packet length and is combined with the transition messages if any. The packet length in the dequeue response is used by the scheduler for DRR.

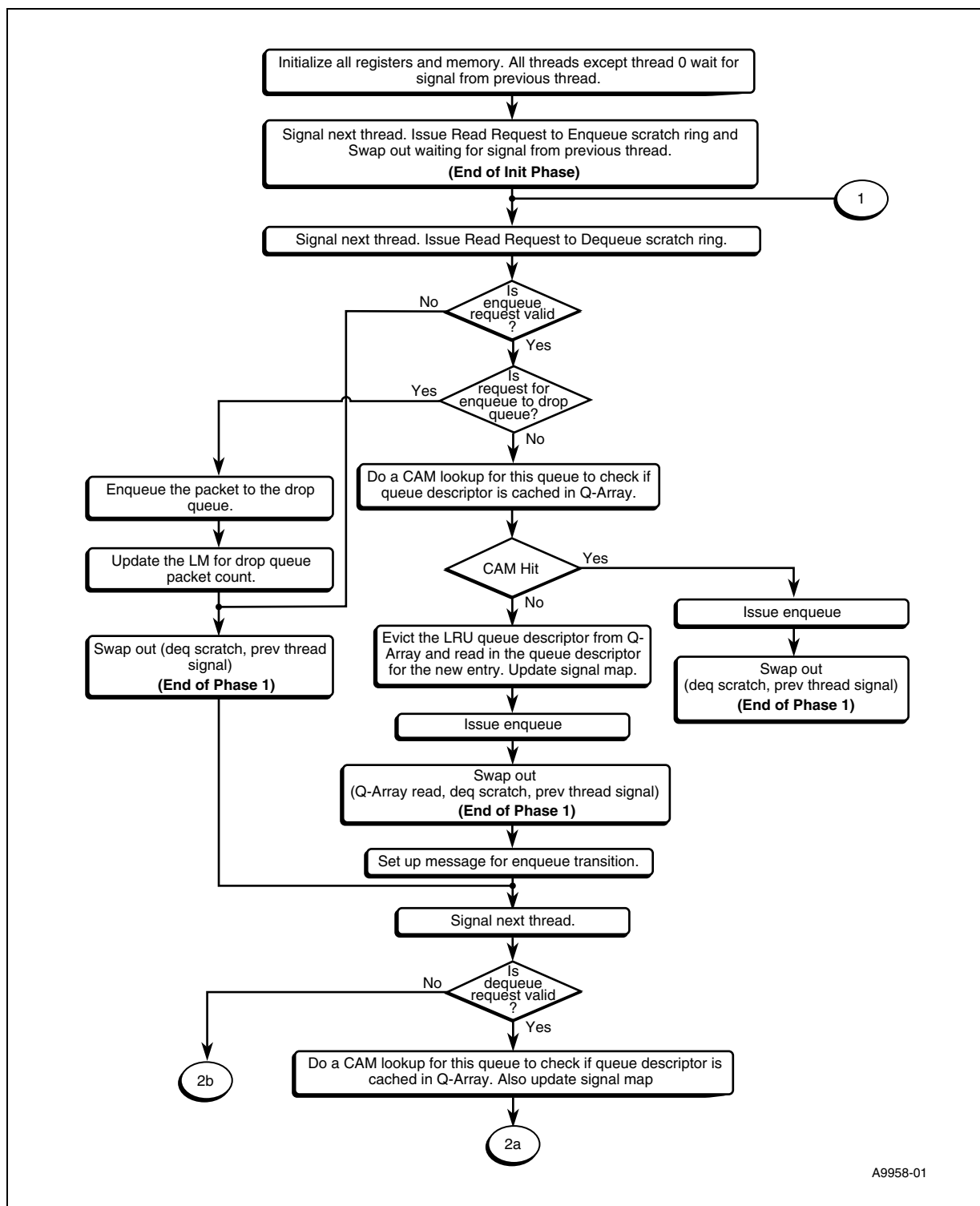
13.6.4 Multiple Queues on Transmit

In MPHY-16 mode, the Packet Transmit block runs on two microengines, each of which has its own scratch ring for transmit requests. In MPHY-4 or SPHY 4x8 mode, the packet transmit block uses four scratch rings - one per port. The packet mode QM may be configured at compile time to support the different configurations.

13.7 Flow Chart

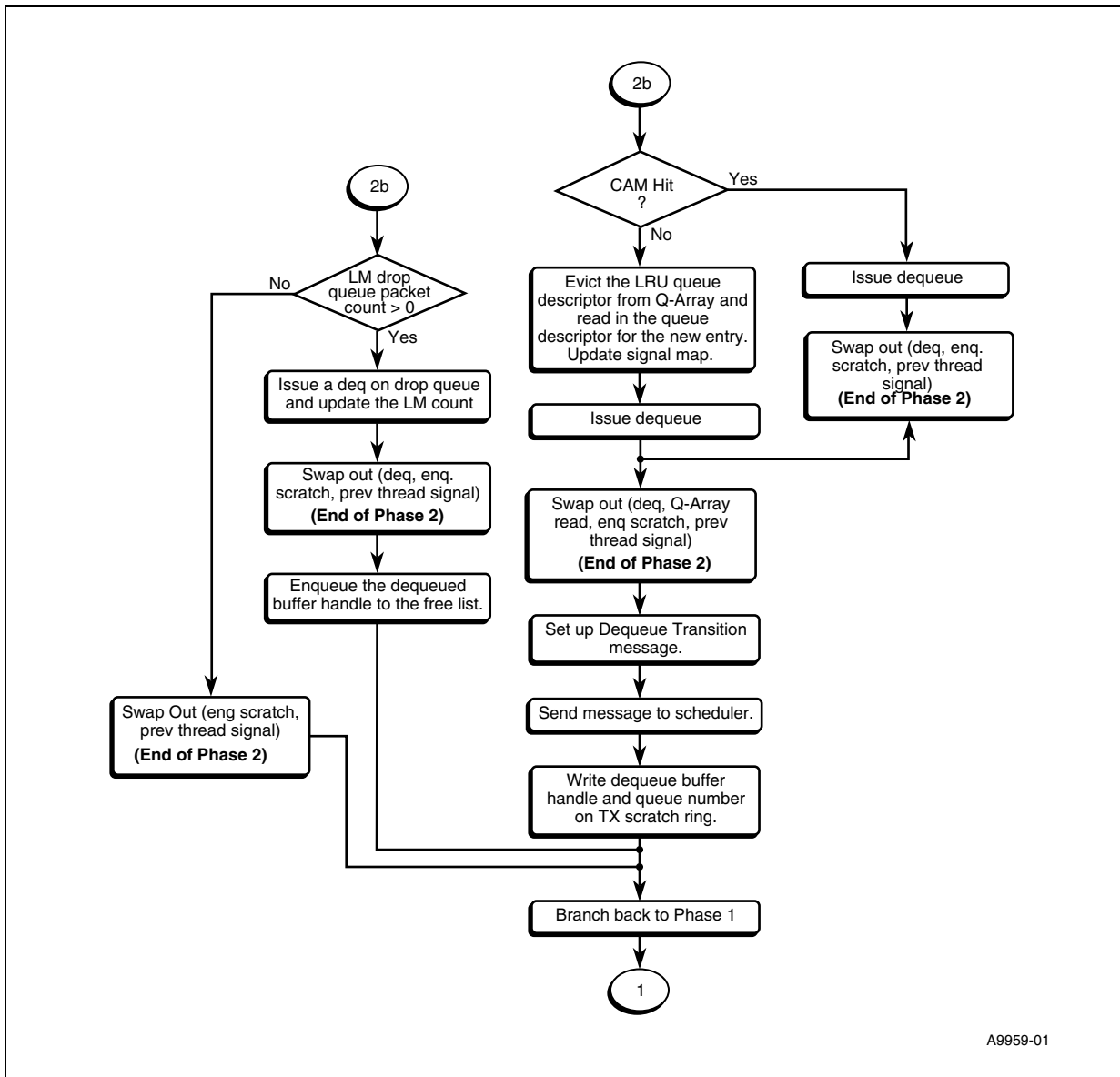
Figures 13-6 and 13-7 are a two-page flowchart for the Ingress Queue Manager functionality.

Figure 13-6. Ingress Queue Manager: Flow Chart Page 1 of 2



A9958-01

Figure 13-7. Ingress Queue Manager: Flow Chart Page 2 of 2



13.8 Performance Analysis

Table 13-2 shows the worst cycle count estimate for the different phases of the QM microblock in the critical path.

The worst case for the QM occurs when all the following occur in one iteration of the loop:

- the enqueue stage has a CAM miss
- the dequeue stage has a CAM miss
- an enqueue transition occurs
- a dequeue transition occurs

Table 13-2. Ingress Queue Manager Performance Analysis: Worst Case Cycle Count

Component	Worst Case Cycle Count for Packet Mode	Worst Case Cycle Count for Cell Mode
Phase 1	56	49
Phase 2	25	25
Total	81	74

Since the available cycle count for the min packet case is 88 cycles, the QM microblock meets the performance requirements in terms of cycle count.

Table 13-3 shows the I/O operations performed in the different phases of the QM microblock. The IO latency available to a thread is 88*8 cycles for the two phases combined.

Table 13-3. Ingress Queue Manager Performance Analysis: I/O Operations

I/O Operations	Phase
Scratch read of enq request	2 and init phase
Scratch read of deq request	1
Write enq queue descriptor from Q-Array to SRAM using <code>sram[write_qdesc]</code>	1 and 2
Read deq queue descriptor from SRAM to Q-Array using <code>sram[read_qdesc]</code>	1 and 2
SRAM Enqueue operation	1
SRAM Dequeue operation	2
Write to TX scratch ring	1

Queue Manager For OC-192 Microblock

14

This section describes the low-level design and implementation of the IXP2800 Queue Manager (QM).

14.1 Overview

The IXP2800 QM is implemented as a microblock running on a single microengine.

Note: Since the QM is not likely to be combined with any other microblock on the same thread of execution, there is no need for a dispatch loop.

The QM is responsible for performing enqueue and dequeue operations on the transmit queues which are implemented using the hardware SRAM link lists. It accepts enqueue and dequeue requests from the scheduler microengine via a Next Neighbor ring.

The key difference between the IXP2800 running on OC-192 POS data rates and the IXP2400 running at OC-48 POS data rates is that the instruction cycle budget for the Queue Manager is significantly smaller (57 as compared to 97). Also the IXP2400 design has the potential for invalid dequeues (dequeues issued to a queue with no data) which are not acceptable at OC-192 data rates. For this reason, the scheduler is moved in-line with the queue manager. The scheduler receives enqueue requests from the packet processing microengines and uses it to accurately keep track of the number of entries in every queue. It forwards the enqueue requests to the QM and also issues dequeue requests to the QM for queues with data.

The Queue Manager uses the Q-Array hardware in the SRAM controller to support link lists. The queue descriptors for the queues are maintained in SRAM. Sixteen queue descriptors are cached in the Q-Array in the SRAM controller. The Queue Manager uses the CAM to maintain this cache of queue descriptors. When an enqueue operation comes in, the queue manager checks the CAM to see if the queue descriptor for this queue is cached in local memory. If it is, then it simply performs the enqueue operation. If not, the LRU (Least Recently Used) queue descriptor is evicted from the Q-Array and written back to SRAM. Then the queue descriptor specified in the enqueue operation is read into the Q-Array cache and the enqueue operation is performed. A similar algorithm is used for the dequeue operation. In order to meet the performance budget, the Queue Manager must handle in the worst case, one enqueue and one dequeue operation every 57 cycles (min packet time for OC-192 POS). To maintain coherence, the threads on the QM microengine execute in strict order using local inter-thread signaling.

The Queue Manager operates in two modes - cell mode and packet mode. Cell mode is used to transmit into CSIX and ATM media while the packet mode is used for transmit into POS and Ethernet media. In cell mode, the QM enqueues packets and dequeues cells, while in the packet mode the QM enqueues and dequeues complete packets.

14.2 Assumptions/Dependencies

- The Queue Manager assumes that the Queue Array entries 0..15 are reserved for the QM cache.
- Since the scheduler tracks the number of entries in a queue accurately, the QM does not maintain this information, nor does it send transition messages to the scheduler.
- In the OC-48 design, the QM handled the dropping of large packets. This functionality is now moved to the statistics microengine.

14.3 Data Structures

This section describes the data structures relevant to the Queue Manager.

14.3.1 Queue Descriptor

Figure 14-1 illustrates the Format of queue descriptor.

Figure 14-1. QM OC-192 Format of Queue Descriptor

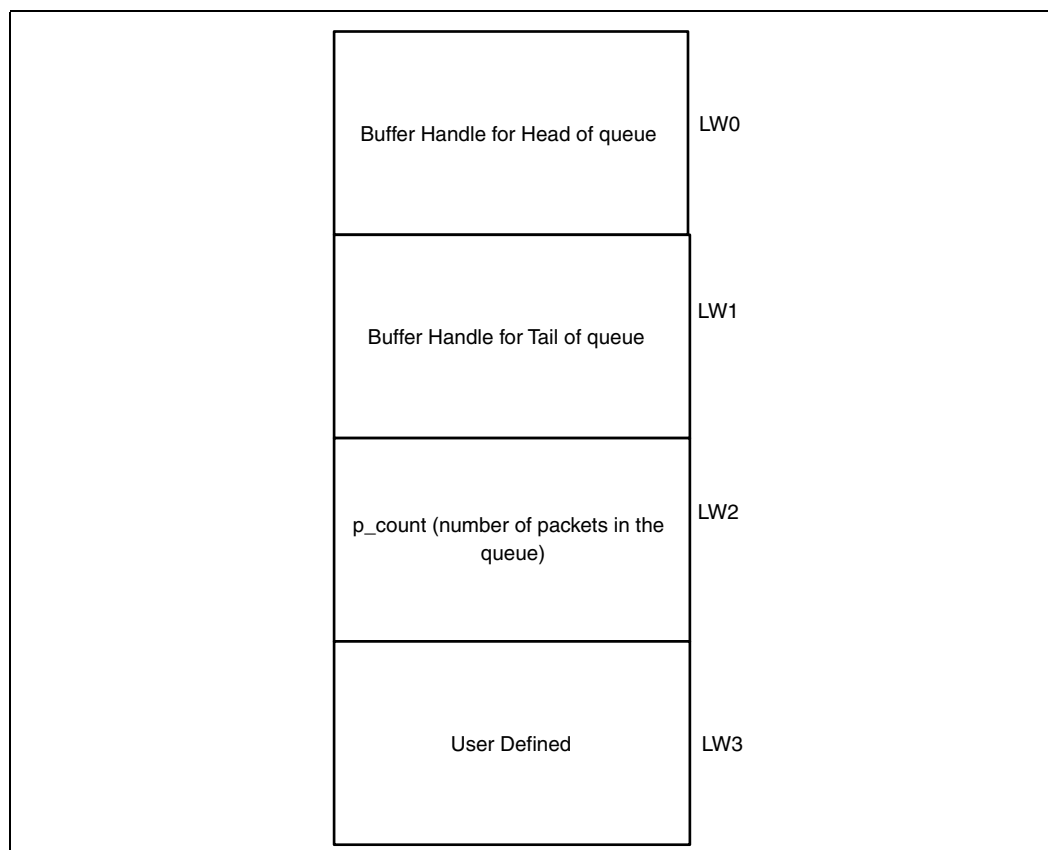


Figure 14-1 illustrates the format of queue the descriptor as stored in SRAM. When the queue is referenced, the Q-Array moves the first three words (or four depending on how the Q-Array is configured) into the SRAM controller. The Q-Array can cache up to 64 queue descriptors per SRAM channel. However for the Queue Manager uses only 16 entries. The remaining may be used for buffer free lists, SRAM rings etc.

Note: Even though SRAM is byte addressed, the Q-Array operations are all based on long words.

14.4 Build Switches

This section describes the compile time options used in the Queue Manager.

Table 14-1. Compile Time Options Used in the Queue Manager

Symbol	Description
PACKET_MODE	Set this build switch to use the Queue Manager in Packet Mode. Default is cell mode.

14.5 Design

The QM uses 8 threads running on a single microengine to handle OC-192 data rates. Every 57 cycles (min packet time for OC-192 POS) each thread handles in the worst case, one enqueue and one dequeue request received via a next neighbor ring from the scheduler.

The Queue Manager uses the Q-Array hardware in the SRAM controller to support link lists. The queue descriptors for the queues are maintained in SRAM. 16 queue descriptors are cached in the Q-Array in the SRAM controller. The Queue Manager uses the CAM to maintain this cache of queue descriptors. When an enqueue operation comes in, the queue manager checks the CAM to see if the queue descriptor for this queue is cached in local memory. If it is, then it simply performs the enqueue operation. If not, the LRU (Least Recently Used) queue descriptor is evicted from the Q-Array and written back to SRAM. Then the queue descriptor specified in the enqueue operation is read into the Q-Array cache and the enqueue operation is performed. A similar algorithm is used for the dequeue operation.

The Queue Manager supports both cell and packet mode as a compile time option. In packet mode the QM queues the packets using the packet_next field in the meta data. However the SRAM offset in the buffer handle received for the packet points to the start of the metadata, which is the buffer_next field. To work around this, the QM moves the offset in the handle to point to the packet_next field before the enqueue operation. When the packet is dequeued, the QM moves the offset in the handle to point back to the start of the metadata before sending it for transmit.

Figure 14-2 illustrates the basic functionality of the Queue Manager:

Figure 14-2. Basic Functionality of the Queue Manager

Initialize

Initialize Queue Descriptors and CSR's in thread 0.
All threads except thread 0, wait on previous thread signal

```
while (1)
{
    Signal next thread

    Read Enqueue and Dequeue Request From NN Ring

    if( valid enq request )
    {
        Do CAM lookup with enqueue q#;

        if( CAM hit )
        {
            issue enqueue
        }
        else // CAM miss
        {
            evict LRU from CAM. Read new Queue Descriptor from SRAM
            issue enqueue
        }
    }

    if( valid dequeue request )
    {
        check CAM;
        if( CAM hit )
        {
            issue dequeue
        }
        else // CAM miss
        {
            evict LRU from CAM. Read new Queue Descriptor from SRAM
            issue dequeue
        }
    }

    Wait for I/O operations to finish and previous thread signal
}
```

14.6 Differences between OC-48 and OC-192 QMs

The following are the significant differences between the OC-48 and OC-192 Queue Managers:

Table 14-2. Differences between the OC-48 and OC-192 Queue Manager

OC-48 Queue Manager	OC-192 Queue Manager
It receives enqueue requests from the packet processing code via a scratch ring and dequeue requests from the scheduler via another scratch ring.	It receives enqueue and dequeue requests from the Queue Manager via a Next Neighbor ring.
It keeps track of queue counts and detects enqueue/dequeue transitions and sends them to the scheduler via a Next Neighbor ring	It does not track queue counts or enqueue/dequeue transitions
It sends transmit requests to the Transmit Microengine via a scratch ring	It sends transmit requests to the Transmit Microengine via a Next Neighbor ring
Since it has scratch I/O operations it has 2 phases and the enqueue and dequeue operations have been spread between the two phases.	There is only phase that handles both the enqueue and dequeue operations.

14.7 Performance Analysis

The table below shows the worst cycle count estimate for the different phases of the QM microblock in the critical path.

The worst case for the QM occurs when all the following occur in one iteration of the loop:

- The enqueue stage has a CAM miss
- The dequeue stage has a CAM miss

Table 14-3. QM OC-192—Worse Cycle Count Estimate for Different Phases

Instruction budget	Worst Case Cycle Count for Packet Mode	Worst Case Cycle count for Cell Mode
57 cycles	55	51

The table below shows the I/O operations performed in the different phases of the QM microblock. The I/O latency available to a thread is 57*8 cycles for the two phases combined.

Table 14-4. IQM OC-192—I/O Operations Performed in the Different Phases

I/O operations	Number of words
Write queue descriptor from Q-Array to SRAM using sram[write_qdesc]	3 words
Read queue descriptor from SRAM to Q-Array using sram[read_qdesc]	3 words
Sram Enqueue operation	1
Sram Dequeue operation	1

14.7.1 Characterization Data

Table 14-5. OC-192 Queue Manager Microblock Characterization Data

Data	Value
General:	
Microblock Name	QM_UC
Microblock Version Number	1.0
Implementation Language	microcode
Configuration Options use to gather this set of data	1. PACKET_MODE 2. USE_TX_HELPER
Measurement Environment (tool settings)	SDK 3.5 Assembler optimizer enabled.
Performance: (cycle counts, latencies, bandwidths)	
Instruction cycle count per packet (common case packet / worst case cycle count)	Packet mode: 55 Cell mode: 51
Common-case packet/path assumptions to be documented here	Assumptions <ul style="list-style-type: none"> Queue Array entries 0..15 are used No number of queue entries information is maintained.
Scratch Memory	
# of longwords read (for bandwidth calculations)	0
# of longwords written (for bandwidth calculations)	0
# and type of each atomic operation performed (for bandwidth calculations)	0
SRAM	
# of longwords read	queue descriptor: 4
# of longwords written	queue descriptor: 4
# and type of each atomic operation performed (for bandwidth calculations)	0
DRAM	
# of quadwords read	0
# of quadwords written	0
Other I/O instructions (i.e. MSF, PCI, CAP, and Hash)	0
List of dependent I/O accesses in the longest latency path	0
Per-Microengine Resources:	
Control-Store Usage (# of instructions used)	packet mode: 130 cell mode: 122
Local Memory Footprint (# of long words used)	16

Table 14-5. OC-192 Queue Manager Microblock Characterization Data (Continued)

Data	Value
Local Memory Configuration (shared, or per-context pointer)	-
Local Memory - # of LM pointers used	0
GPR Usage – minimum, static usage (absolutes, static, globals)	
Transfer Reg. Usage – minimum, static usage	
Next Neighbor Reg. Usage – minimum, static usage	
Signal Usage – minimum, static usage	
CAM used? (yes or no)	Yes

Global Resources:	
Scratch footprint (# of longwords used) – constant or formula ...	0
SRAM footprint (# of longwords used) – constant or formula ...	0
DRAM footprint (# of quadwords used) – constant or formula ...	0
Q-Array usage - # of queues used and if they need to be cached	16
CRC Unit used?	no
Hash Unit used? (yes or no)	no

MSF Usage Information:	
Media Bus Configuration	-
RBUF, TBUF usage	-
CBus signals	-

Other Information:	
Critical Section Length (compute cycles + memory accesses)	0
# of phases	2
Packet Metadata - fields read	Packet mode: 1 Cell mode: 0
Packet Metadata - fields written	Packet mode: 1 Cell mode: 0
Header - fields read	-
Header - fields written	-

Documentation:	
Thread Ordering Requirements	Threads need to run in order.

Table 14-5. OC-192 Queue Manager Microblock Characterization Data (Continued)

Data	Value
OS dependencies	VxWorks, Linux
Chip/hardware dependencies (i.e. Crypto Unit in 2850)	IXP2800, IXP2850
Tested on which SDK Release(s)	IXA SDK 3.5
Tested on hardware? Which hardware configuration?	Yes, IXDP 2800
Tested in which applications (not an all inclusive list)	Several IXA SDK 3.5 applications
Possible Configuration Options	
Major limitations (i.e. Rx/Tx microblocks written to use entire ME)	None
Packet Sequencing Issues (esp. in POT applications)	None
Core Component or Interface requirements or dependencies	QM CC

Packet Queue Manager Microblock 15

15.1 Overview

The design of the Packet mode Queue Manager (QM) used in the POS and Ethernet egress pipelines is almost identical to the design of the Ingress QM described in [Chapter 13, “Queue Manager For OC-48 Microblock.”](#) Therefore this section captures only the differences between the two.

The following are the significant differences between the Ingress and Egress QM.

15.2 Q-Array Hardware Configuration

The QM for the POS/Ethernet egress pipeline works in packet mode while the ingress QM works in cell mode. This means that while the egress QM dequeues an entire packet and sends it to the transmit block, the ingress QM dequeues a cell in a packet and sends it for transmit.

The packet mode QM configures the SRAM Q-Array CSR's to ignore the cell count field in the buffer handle. The EOP bit still needs to be set for the Q-Array hardware to accurately keep track of the number of packets in a queue.

15.3 Hierarchical Queuing

The packet mode QM queues the packets using the `packet_next` field in the meta data. However the SRAM offset in the buffer handle for the packet points to the start of the metadata, which is the `buffer_next` field. To work around this, the QM moves the offset in the handle to point to the `packet_next` field before the enqueue operation. When the packet is dequeued, the QM moves the offset in the handle to point back to the start of the metadata before sending it for transmit.

15.4 Dequeue Response

The ingress cell mode QM sends a message to the scheduler only in the case of an invalid dequeue or if a queue transitions from empty to non-empty or vice-versa. The egress packet mode QM however sends a response message to the scheduler for every dequeue request. The response message contains the packet length and is combined with the transition messages if any. The packet length in the dequeue response is used by the scheduler for DRR.

15.5 Multiple Queues on Transmit

In MPHY-16 mode, the Packet Transmit block runs on two microengines, each of which has its own scratch ring for transmit requests. In MPHY-4 or SPHY 4x8 mode, the packet transmit block uses four scratch rings—one per port. The packet mode QM may be configured at compile time to support the different configurations.

15.6 Performance Analysis

Table 15-1 shows the worst cycle count estimate for the Egress Queue Manager.

Table 15-1. Egress Queue Manager Performance Analysis: Cycle Count

Component	Worst Case Cycle Count
Phase 1	56
Phase 2	25
Total	81 (budget is 97 cycles for OC48c POS min packet)

15.6.1 Characterization Data

Table 15-2. Packet Queue Manager Microblock Characterization Data

Data	Value
General:	
Microblock Name	QM_PACKET_UC
Microblock Version Number	1.0
Implementation Language	microcode
Configuration Options use to gather this set of data	EGRESS
Measurement Environment (tool settings)	SDK 3.1 Assembler optimizer enabled.
Performance: (cycle counts, latencies, bandwidths)	
Instruction cycle count per packet (common case packet / worst case cycle count)	81
Common-case packet/path assumptions to be documented here	Assumptions <ul style="list-style-type: none"> Queue Array entries 0..15 are used. Only SRAM linked list is supported.
Scratch Memory	
# of longwords read (for bandwidth calculations)	Enqueue: 3 Dequeue: 1
# of longwords written (for bandwidth calculations)	1

Table 15-2. Packet Queue Manager Microblock Characterization Data (Continued)

Data		Value
# and type of each atomic operation performed (for bandwidth calculations)		0
SRAM		
# of longwords read		4
# of longwords written		4
# and type of each atomic operation performed (for bandwidth calculations)		0
DRAM		
# of quadwords read		0
# of quadwords written		0
Other I/O instructions (i.e. MSF, PCI, CAP, and Hash)		0
List of dependent I/O accesses in the longest latency path		None

Per-Microengine Resources:

Control-Store Usage (# of instructions used)	282
Local Memory Footprint (# of long words used)	16
Local Memory Configuration (shared, or per-context pointer)	-
Local Memory - # of LM pointers used	0
GPR Usage – minimum, static usage (absolutes, static, globals)	
Transfer Reg. Usage – minimum, static usage	
Next Neighbor Reg. Usage – minimum, static usage	
Signal Usage – minimum, static usage	
CAM used? (yes or no)	Yes

Global Resources:

Scratch footprint (# of longwords used) – constant or formula ...	2
SRAM footprint (# of longwords used) – constant or formula ...	0
DRAM footprint (# of quadwords used) – constant or formula ...	0
Q-Array usage - # of queues used and if they need to be cached	16
CRC Unit used?	no
Hash Unit used? (yes or no)	no

MSF Usage Information:

Table 15-2. Packet Queue Manager Microblock Characterization Data (Continued)

Data		Value
Media Bus Configuration	-	
RBUF, TBUF usage	-	
CBus signals	-	

Other Information:		
Critical Section Length (compute cycles + memory accesses)	0	
# of phases	2	
Packet Metadata - fields read	1	
Packet Metadata - fields written	1	
Header - fields read	0	
Header - fields written	0	

Documentation:		
Thread Ordering Requirements	Threads need to run in order	
OS dependencies	VxWorks, Linux	
Chip/hardware dependencies (i.e. Crypto Unit in 2850)	2400	
Tested on which SDK Release(s)	IXA SDK PR-6	
Tested on hardware? Which hardware configuration?	Yes, IXDP 2400	
Tested in which applications (not an all inclusive list)	Several IXA SDK 3.1 applications	
Possible Configuration Options		
Major limitations (i.e. Rx/Tx microblocks written to use entire ME)	None	
Packet Sequencing Issues (esp. in POT applications)	None	
Core Component or Interface requirements or dependencies	QM CC	

This section describes the design and implementation of the ATM Queue Manager.

16.1 Overview

The ATM Queue Manager is responsible for performing enqueue and dequeue operations on the transmit queues which are implemented using the hardware SRAM link lists. It accepts enqueue requests from the packet processing microengines via a scratch ring. Dequeue requests come from the TM 4.1 scheduler microengine.

The Queue Manager uses the Q-Array hardware in the SRAM controller to support link lists. The queue descriptors for the queues are maintained in SRAM and sixteen queue descriptors are cached in the Q-Array in the SRAM controller. The Queue Manager uses the CAM to maintain this cache of queue descriptors. When an enqueue operation comes in, the queue manager checks the CAM to see if the queue descriptor for this queue is cached in local memory. If it is, then it simply performs the enqueue operation. If not, the LRU (Least Recently Used) queue descriptor is evicted from the Q-Array and written back to SRAM. Then the queue descriptor specified in the enqueue operation is read into the Q-Array cache and the enqueue operation is performed. A similar algorithm is used for the dequeue operation.

To maintain coherence, the threads on the QM microengine execute in strict order using local inter-thread signaling.

After every enqueue operation, the Queue Manager sends a message to the TM 4.1 Shaper via a Next Neighbor Ring. Also after every dequeue operation, the QM passes a transmit request via a scratch ring to the AAL5 TX microblock. For the 16 queue descriptors cached in the Q-Array, the QM keeps track of the number of packets in the queue using local memory.

The Queue Manager operates in a cell mode, that is, it enqueues packets and dequeues cells.

The Queue Manager can share one microengine with the TM 4.1 Shaper microblock. In this configuration microblocks work in the functional pipeline.

16.2 Assumptions/Dependencies

The Queue Manager assumes that Queue Array entries 0...15 are reserved for the QM cache.

Even though the same algorithm applies for SRAM rings also, this implementation supports only link lists.

16.3 Data Structures

This section describes the data structures relevant to the Queue Manager.

16.3.1 Queue Descriptor

Figure 16-1 shows the format of queue descriptor as stored in SRAM. When the queue is referenced, the Q-Array moves the first three words into the SRAM controller. The Q-Array can cache up to 64 queue descriptors per SRAM channel. However, the Queue Manager uses only 16 entries. The remaining may be used for buffer free lists, SRAM rings etc.

The Timestamp (L3 & LW4) fields contain information about the last access (enqueue or dequeues) to the appropriated queue (this value is necessary for example, for the WRED microblock).

Figure 16-1. Format of Queue Descriptor

Buffer Handle for Head of queue	LW0
Buffer Handle for Tail of queue	LW1
p_count (number of packets in the queue)	LW2
Timestamp Low	LW3
Timestamp High	LW4
User Defined	LW5
User Defined	LW6
User Defined	LW7

Note: Even though SRAM is byte addressed, the Q-Array operations are all based on long words.

16.3.2 Packet Counts in Local Memory

The Queue Manager stores the current packet count for the cached queues in local memory. 16 words are used, each representing the packet count of one entry in the Q-Array cache. The packet count is initialized when the entry is read into the Q-Array (the `sram[rd_qdesc]` instruction returns the value from the queue descriptor stored in SRAM). Subsequently, the packet count is incremented/decremented as appropriate during the enqueue/dequeue operation. The Queue Manager periodically updates the queue descriptor in SRAM with a value of the packet count from local memory.

16.4 External Interfaces

This section describes the interfaces between the ATM Queue Manager and other microblocks.

In most of the messages, there is a valid bit is used to prevent a value of zero from being enqueued on the scratch ring. Zero is used to detect a case where the scratch ring is empty. So the valid bit helps us distinguish between a zero value that was actually enqueued versus a case where the ring is empty.

16.4.1 LLC SNAP Encap and Cell QM

The interface between the LLC SNAP Encap microblock and the ATM Queue Manager is a scratch ring. Table 16-1 shows the format of each entry in the scratch ring, which is three long words.

Table 16-1. Scratch Ring (LLC SNAP Encap and ATM QM)—Three Long Words

LW	Bits	Size	Field	Description
0	31:0	32	SOP Buffer Handle	Buffer Handle for the SOP Descriptor
1	31:0	32	EOP Buffer Handle	Buffer Handle for EOP Descriptor (may be NULL)
2	31	1	Valid Bit	Must be 1
2	30:29	2	Reserved	Reserved
2	28:18	11	Packet cell count	Number of 48-byte cells in the entire packet
2	17:16	2	Reserved	Reserved
2	15:0	16	Queue Number	Queue Number

16.4.2 ATM QM and TM 4.1 Shaper

The interface between the ATM Queue Manager and the TM 4.1 Shaper is a Next Neighbor ring. Table 16-2 shows the format of each entry in the NN ring, which is two long words.

Table 16-2. NN Ring (ATM QM and TM 4.1 Shaper)—Two Long Words

LW	Bits	Size	Field	Description
0	31	1	Valid Bit	The enqueue word is valid only if this bit is set
0	30	1	Enqueue Transition	Notification that queue has gone from empty to non-empty
0	29	1	Reserved	Reserved
0	28:18	11	Cell count	Cell count provides the number of cells in the frame.
0	17	1	SOP	The Start of Packet (SOP) bit is used for GFR shaping algorithm set to "1" when transition bit is also set, otherwise "0"
0	16:0	17	Queue Number	Queue Number that was enqueued
1	31	1	Valid Bit	Must be 1
1	30	1	Dequeue Transition	Notification that queue has gone from non-empty to empty

Table 16-2. NN Ring (ATM QM and TM 4.1 Shaper)—Two Long Words (Continued)

LW	Bits	Size	Field	Description
1	28:18	12	Reserved	Reserved
1	17	1	SOP	The Start of Packet (SOP) bit is used for GFR shaping algorithm set to “1” message when QM has transmitted last cell from current packet and there is an another packet in the queue, otherwise “0”
1	16:0	17	Queue Number	Queue Number that was dequeued

16.4.3 TM 4.1 Scheduler to ATM QM

The interface between the TM 4.1 Scheduler and the ATM Queue Manager is a scratch ring. Table 16-3 shows the format of each entry in the scratch ring, which is one long word.

Table 16-3. Scratch Ring (TM 4.1 Scheduler and ATM QM)—One Long Word

LW	Bits	Size	Field	Description
0	31	1	Valid Bit	Must be 1
0	30	1	Reserved	Reserved
0	29:19	11	Output port number	Output port number
0	18:17	2	Reserved	Reserved
0	16:0	17	Queue Number	Queue Number

16.4.4 ATM QM and AAL-5 TX

The interface between the ATM Queue Manager and AAL-5 TX microblock is a scratch ring. Table 16-4 shows the format of each entry in the scratch ring, which is two long words.

Table 16-4. Scratch Ring (ATM QM and AAL-5 TX)—Two Long Words

LW	Bits	Size	Field	Description
0	31	1	Valid Bit	Must be 1
0	30	1	Reserved	Reserved
0	29:19	11	Output port number	Output port number
0	18:16	3	Reserved	Reserved
0	15:0	16	Queue Number	Queue Number
1	31:0	32	Buffer Handle	Buffer Handle currently being transmitted for queue

16.5 Build Switches

This section describes the compile time options used in the ATM Queue Manager.

Table 16-5. ATM QM Compile Time Switches

Symbol	Description
TM4_1	Use this when the Queue Manager is used with the TM 4.1 Shaper/Scheduler (always set in ATM QM microblock).
TX_PHY_MODE	Use this to set SPHY_1_32, SPHY_4_8, SPHY_4_8_SHARED, MPHY_4 or MPHY_16. In the SPHY_4_8 or MPHY_4 case, the QM writes the transmit request to 4 scratch rings one per port.
AAL5_TX_EVEN_ODD_PORTS	Use this to indicate to the QM, which two rings (instead of one) should be used for TX requests (for odd and even port numbers).
QUEUE_THRESHOLD	Use this switch to define the maximum number of packets that can be queued in the LBR VC queue at the same time (next packets are dropped). The default value is 16.
HBR_QUEUE_THRESHOLD	Use this switch to define the maximum number of packets that can be queued in the HBR VC queue at the same time (next packets are dropped). The default value is 64.
QM_REF_CNT	Use this to define how often (in packets) the queue descriptor in SRAM must be updated with value of the packet counter from local memory. This update is performed if the queue descriptor is not removed from Q-Array for a long time.
QM_SKIP_SETTING_TIMESTAMP	Use this switch to prevent QM from updating timestamp value in the queue descriptor (performance optimization when WRED microblock is not used in an application).
QM_WITH_SHAPER	Use this build switch to setup QM microblock to work in a functional pipeline with the TM 4.1 Shaper microblock on one microengine.
QM_USE_REG_INTF_TO_SHAPER	Use this switch to use GPRs in communication with TM 4.1 Shaper microblock instead of NN registers, when QM works with shaper microblock on one microengine. Configuring this switch requires the setting up of an appropriated build switch in the TM 4.1 Shaper microblock.

16.6 Design

The QM uses 8 threads running on a single microengine performing (in the worst case scenario) the following operations:

- One enqueue request from the packet processing code
- One dequeue request from the scheduler
- An enqueue transition
- A dequeue transition (or invalid dequeue).

The Queue Manager uses the Q-Array hardware in the SRAM controller to support link lists. The queue descriptors for the queues are maintained in SRAM. 16 queue descriptors are cached in the Q-Array in the SRAM controller. The Queue Manager uses the CAM to maintain this cache of queue descriptors. When an enqueue operation comes in, the queue manager checks the CAM to see if the queue descriptor for this queue is cached in local memory. If it is, then it simply performs the enqueue operation. If not, the LRU (Least Recently Used) queue descriptor is evicted from the

Q-Array and written back to SRAM. Then the queue descriptor specified in the enqueue operation is read into the Q-Array cache and the enqueue operation is performed. A similar algorithm is used for the dequeue operation.

The design has two phases and requires the threads to execute in strict order in each phase. The enqueue and dequeue requests are interleaved between the two phases to even out the I/O operations. So the enqueue request is handled in the first phase, while the dequeue request is handled in the second phase.

After every enqueue operation, the Queue Manager sends a message to the transmit scheduler via a Next Neighbor Ring. Also after every dequeue operation, the QM passes a transmit request via a scratch ring to the AAL5 TX microblock. For the 16 queue descriptors cached in the Q-Array, the QM keeps track of the number of packets in the queue using local memory.

Figures 16-2, 16-3, 16-4, 16-5 show the basic functionality in each phase:

Figure 16-2. Initialize

```
Initialize
Init memory and QArray's. All threads except thread 0, wait on previous thread signal. signal next thread;
read enqueue request from scratch ring. wait(enqueue scratch sig, previous thread signal);
```

Figure 16-3. Phase 1

```
Phase 1
    signal next thread
    read dequeue request from scratch ring.
    if( valid enq request )
    {
        if( ! enqueue request is for drop queue )
        {
            check CAM;
            if( CAM hit )
            {
                issue enqueue
                Wait(deq scratch signal, previous thread sig) ;
            }
            else // CAM miss
            {
                evict LRU from CAM. Read new Queue Descriptor from SRAM
                issue enqueue
                Wait(deq scratch signal, QD read signal, previous thread sig) ;
            }
        }
        else // its a drop queue enqueue request
        {
            enqueue to drop queue
            update local memory count.
            Wait(deq scratch signal, previous thread sig) ;
        }
    }
    else // no enqueue
        Wait( deq scratch signal, previous thread sig) ;
```

Figure 16-4. Phase 2

```

Phase 2
signal next thread
set up message for enqueue transition.
read enqueue request from scratch ring.
if( valid dequeue request )
{
    check CAM;
    if( CAM hit )
    {
        issue dequeue
        Wait(deq sig, enq scratch signal, previous thread sig) ;
    }
    else // CAM miss
    {
        evict LRU from CAM. Read new Queue Descriptor from SRAM
        issue dequeue
        Wait(deq sig, enq scratch signal, QD read signal, previous thread sig) ;
    }
}
else // no dequeue
{
    check local memory for drop queue, queue count;
    if( LM count != zero )
    {
        issue dequeue from drop queue
        decrement local memory count ;
        Wait( deq sig, enq scratch signal, previous thread sig) ;
    }
    else // no dequeue for drop queue
        Wait( enq scratch signal, previous thread sig) ;
}

```

Figure 16-5. Phase 3

```

Phase 3 (Actually start of Phase 1 of next round - loop unroll)

If( valid dequeue request )
{
    set up message for dequeue transition ;
    send the message to scheduler ;
    write the dequeued buffer to TX scratch ring;
}
else
{
    enqueue the dequeued buffer from drop queue to FREE LIST ;
}
jump to phase 1 ;

```

16.7 Flow Chart

Figure 16-6. Queue Manager Operations Algorithm (Page 1 of 5)

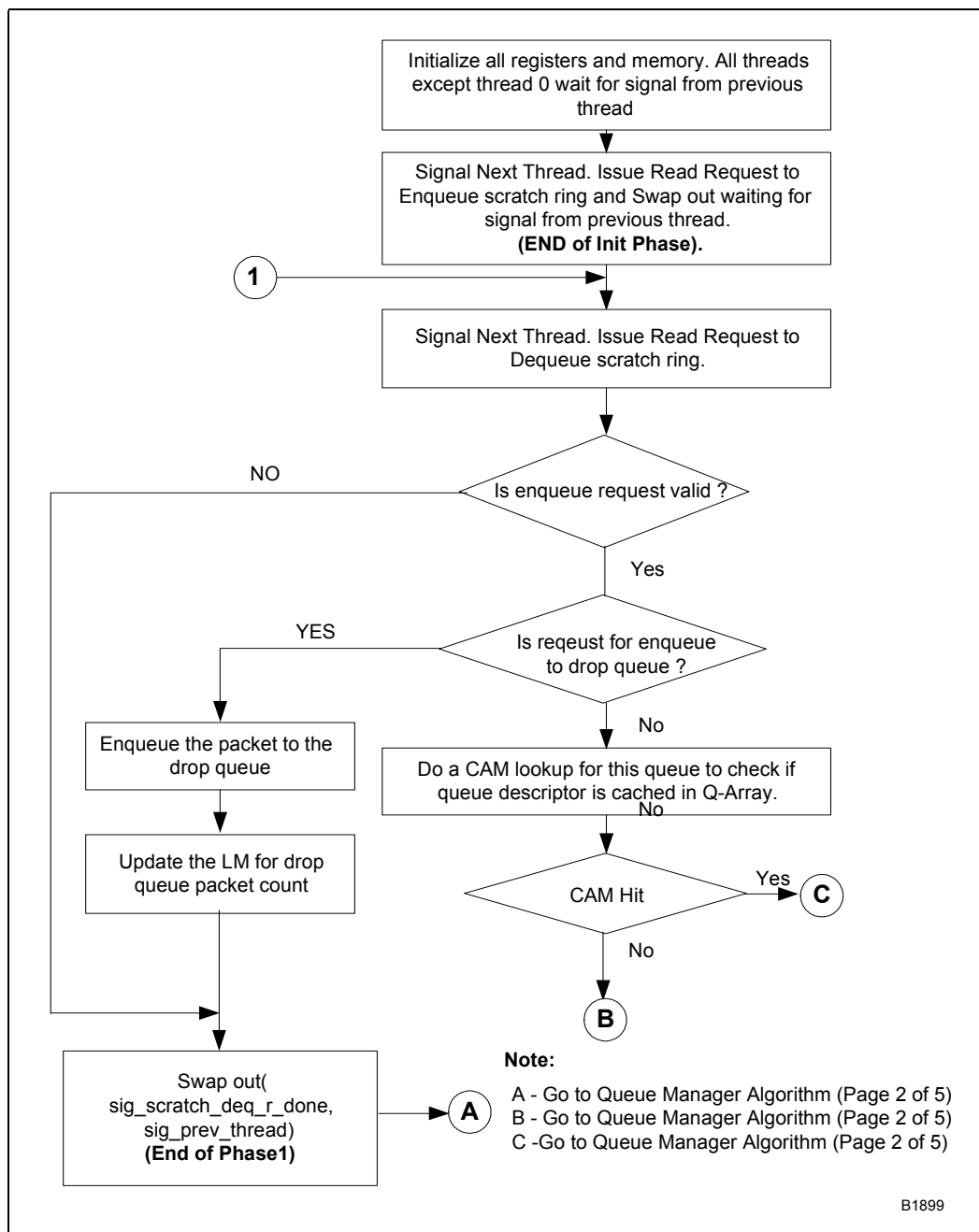


Figure 16-7. Queue Manager Operations Algorithm (Page 2 of 5)

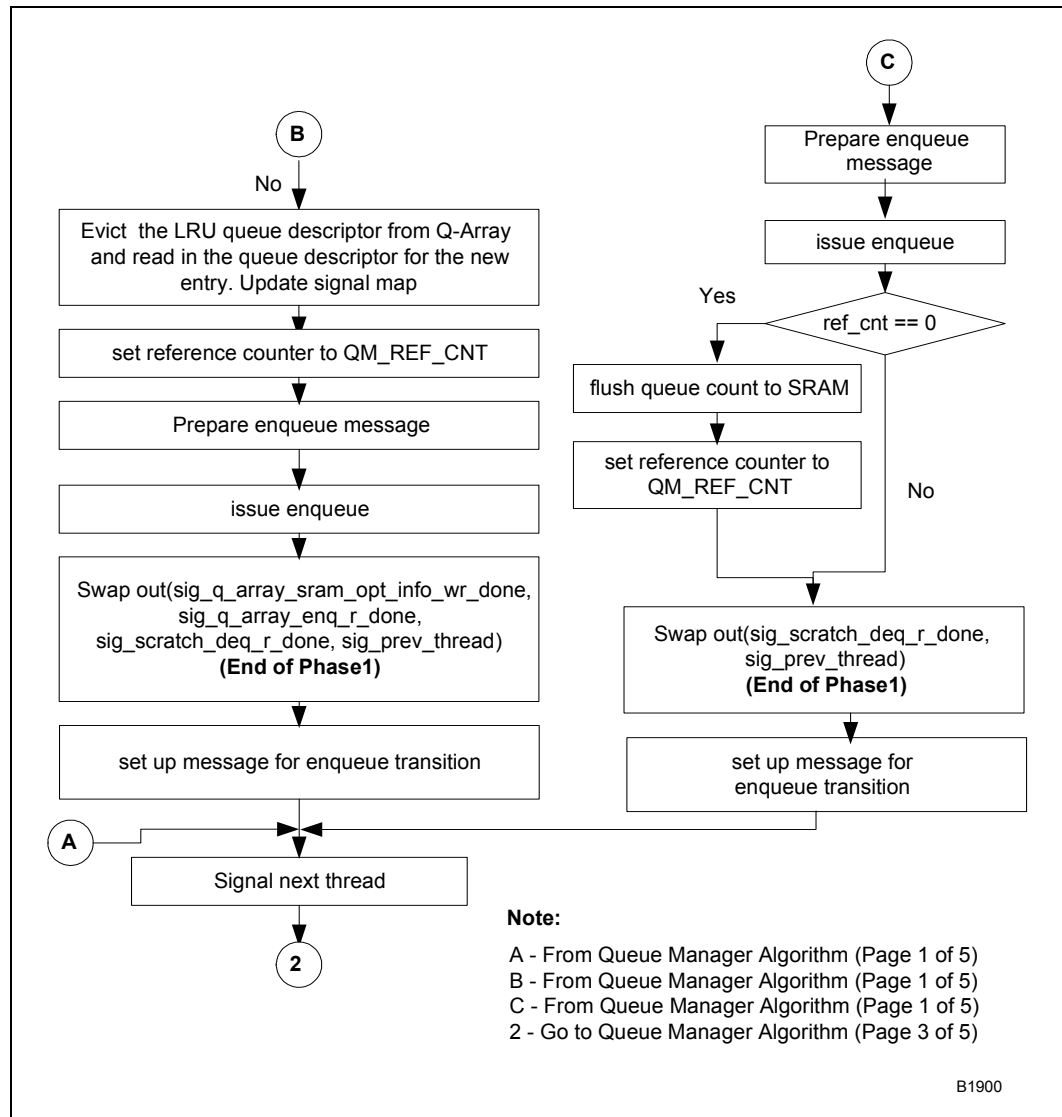


Figure 16-8. Queue Manager Operations Algorithm (Page 3 of 5)

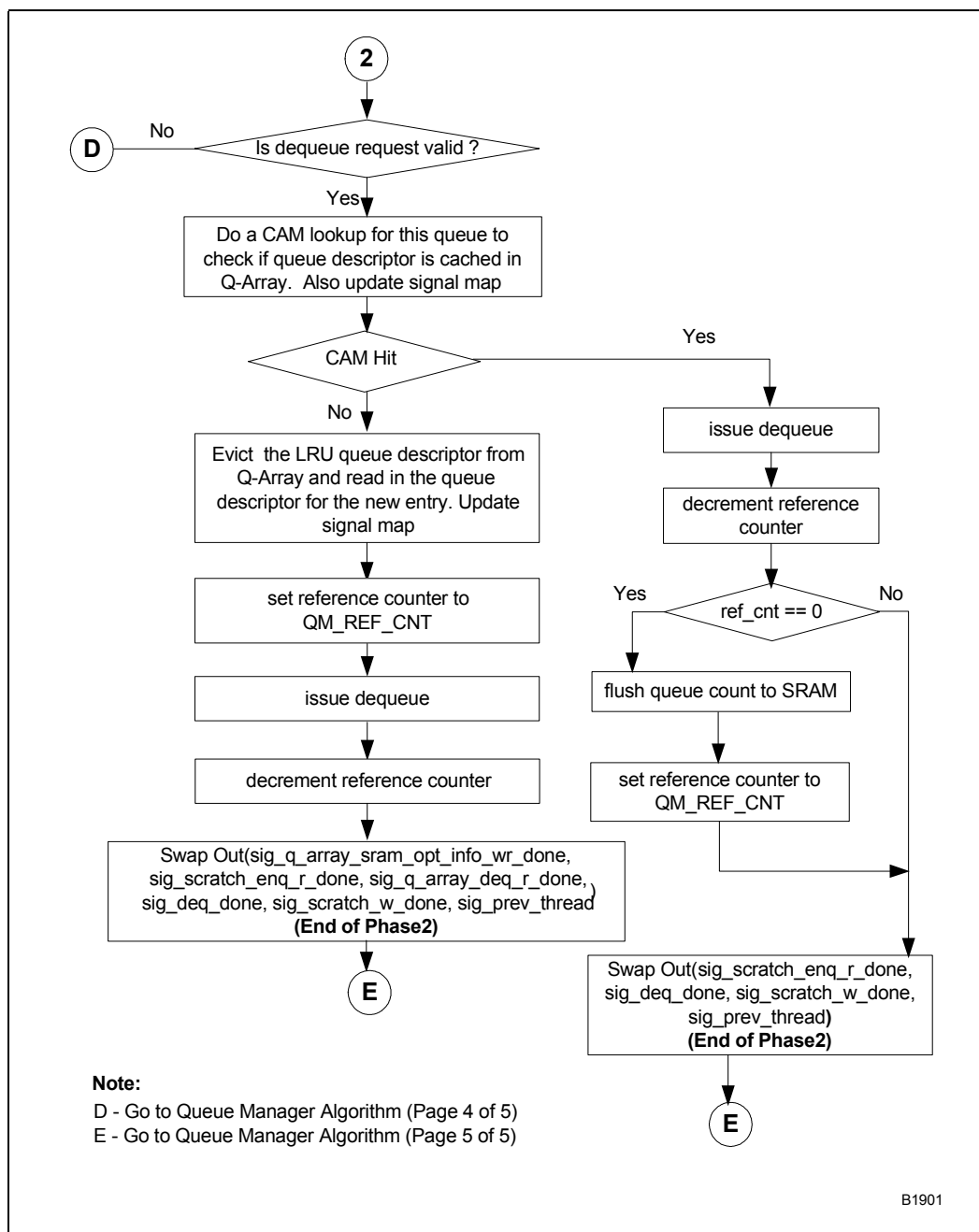


Figure 16-9. Queue Manager Operations Algorithm (Page 4 of 5)

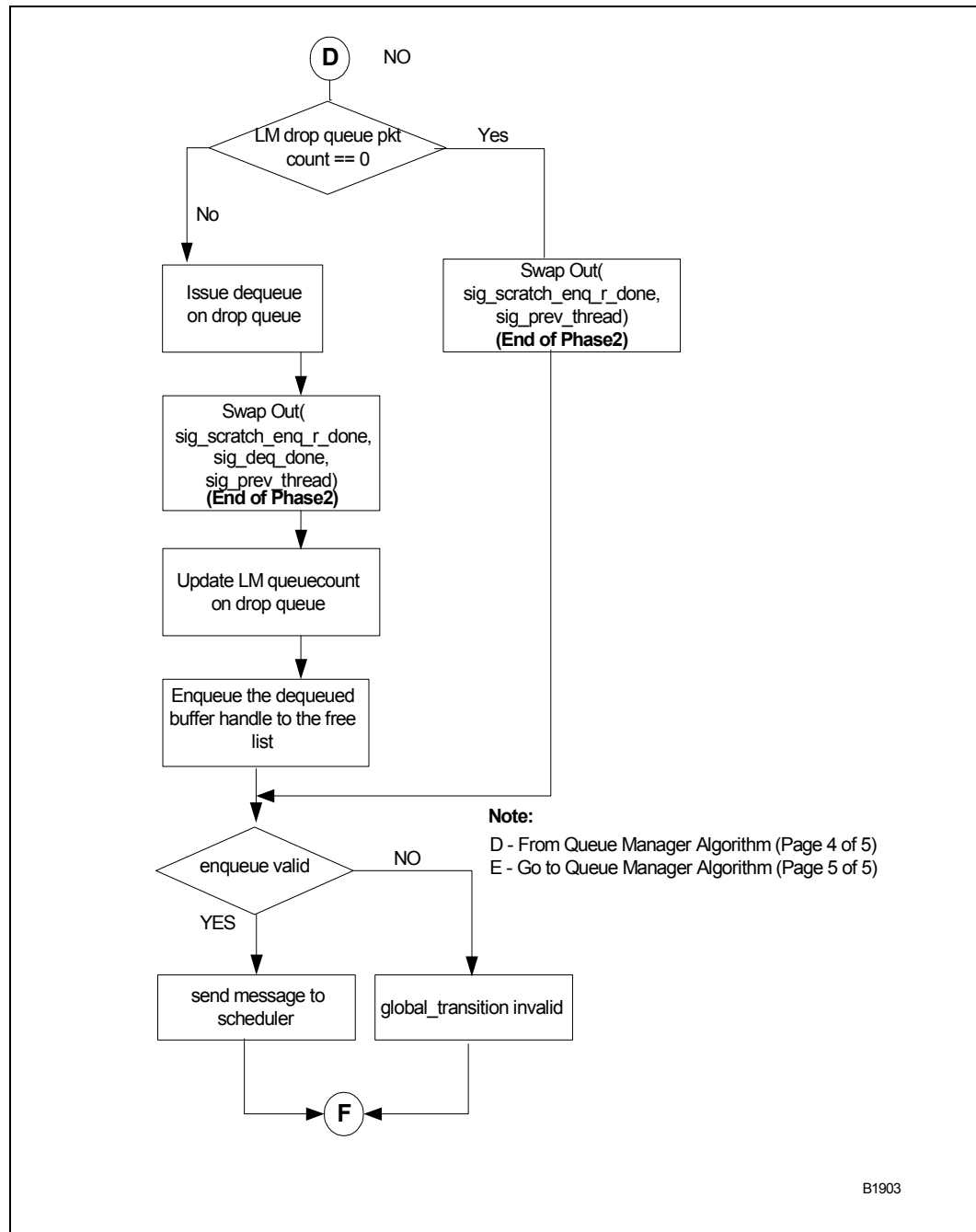
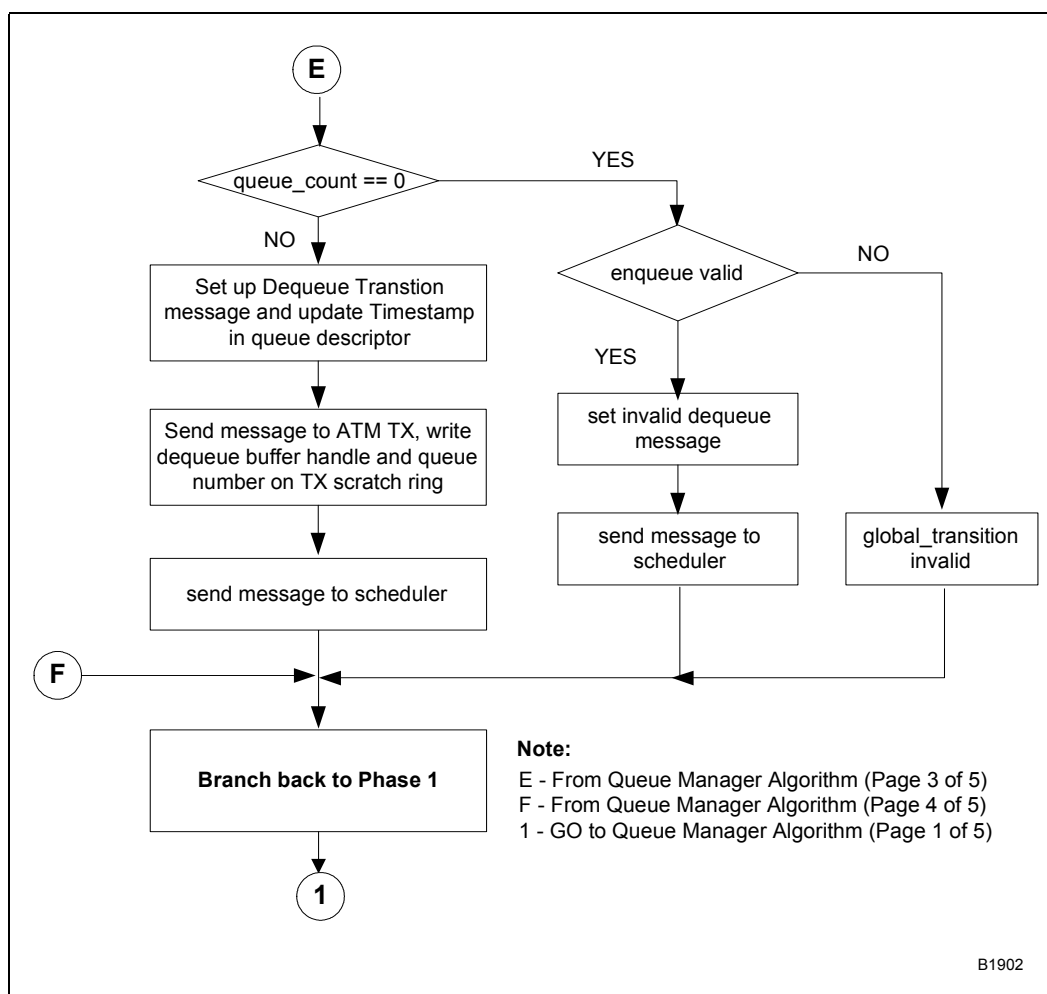


Figure 16-10. Queue Manager Operations Algorithm (Page 5 of 5)



16.8 Performance Analysis

Table 16-6 shows the worst and average cycle count estimate for the different phases of the QM microblock in the critical path.

The worst case for the QM occurs when all the following occur in one iteration of the loop:

- The enqueue stage has a CAM miss
- The dequeue stage has a CAM miss
- An valid enqueue message
- A valid dequeue message

Table 16-6. Cycle Count

Component	Average Case Cycle Count	Worst Case Cycle Count
Phase 1	26	38
Phase 2	65	65
Total	91	103

Since the available cycle count for the min packet case is 106 cycles, the QM microblock meets the performance requirements in terms of cycle count.

Table 16-7 shows the I/O operations performed in the different phases of the QM microblock. The I/O latency available to a thread is 106*8 cycles for the two phases combined.

Table 16-7. I/O Operations Performed in Different Phases of ATM QM

I/O operations	Number in the average case	Number in the worst case
Scratch read of enq request	1 (3LW)	1 (3LW)
Scratch read of deq request	1 (1LW)	1 (1LW)
Write queue descriptor from Q-Array to SRAM using sram[write_qdesc]	1	2
Read queue descriptor from SRAM to Q-Array using sram[read_qdesc]	1	2
Sram Enqueue operation	1	0
Sram Dequeue operation	1	1
Sram write operation (timestamp)	1 (2LW)	1 (2LW)
Write to TX scratch ring	1 (2lw)	1 (2lw)

16.8.1 Characterization Data

Table 16-8. ATM Queue Manager Microblock Characterization Data

Data	Value
General:	
Microblock Name	QM_CELL_UC
Microblock Version Number	1.0
Implementation Language	microcode
Configuration Options use to gather this set of data	
Measurement Environment (tool settings)	SDK 3.1 Assembler optimizer enabled.
Performance: (cycle counts, latencies, bandwidths)	
Instruction cycle count per packet (common case packet / worst case cycle count)	74

Table 16-8. ATM Queue Manager Microblock Characterization Data (Continued)

Data	Value
Common-case packet/path assumptions to be documented here	Assumptions <ul style="list-style-type: none"> Queue Array entries 0..15 are used. Only SRAM linked list is supported.
Scratch Memory	
# of longwords read (for bandwidth calculations)	Enqueue: 3 Dequeue: 1
# of longwords written (for bandwidth calculations)	2
# and type of each atomic operation performed (for bandwidth calculations)	0
SRAM	
# of longwords read	4
# of longwords written	4
# and type of each atomic operation performed (for bandwidth calculations)	0
DRAM	
# of quadwords read	0
# of quadwords written	0
Other I/O instructions (i.e. MSF, PCI, CAP, and Hash)	0
List of dependent I/O accesses in the longest latency path	None

Per-Microengine Resources:

Control-Store Usage (# of instructions used)	277
Local Memory Footprint (# of long words used)	16
Local Memory Configuration (shared, or per-context pointer)	-
Local Memory - # of LM pointers used	0
GPR Usage – minimum, static usage (absolutes, static, globals)	
Transfer Reg. Usage – minimum, static usage	
Next Neighbor Reg. Usage – minimum, static usage	
Signal Usage – minimum, static usage	
CAM used? (yes or no)	Yes

Global Resources:

Scratch footprint (# of longwords used) – constant or formula ...	2
SRAM footprint (# of longwords used) – constant or formula ...	0
DRAM footprint (# of quadwords used) – constant or formula ...	0

Table 16-8. ATM Queue Manager Microblock Characterization Data (Continued)

Data	Value
Q-Array usage - # of queues used and if they need to be cached	16
CRC Unit used?	no
Hash Unit used? (yes or no)	no
MSF Usage Information:	
Media Bus Configuration	-
RBUF, TBUF usage	-
CBus signals	-
Other Information:	
Critical Section Length (compute cycles + memory accesses)	0
# of phases	2
Packet Metadata - fields read	1
Packet Metadata - fields written	1
Header - fields read	None
Header - fields written	None
Documentation:	
Thread Ordering Requirements	Threads need to run in order
OS dependencies	VxWorks, Linux
Chip/hardware dependencies (i.e. Crypto Unit in 2850)	2400
Tested on which SDK Release(s)	IXA SDK PR-6
Tested on hardware? Which hardware configuration?	Yes, IXDP 2400
Tested in which applications (not an all inclusive list)	Several IXA SDK 3.1 applications
Possible Configuration Options	
Major limitations (i.e. Rx/Tx microblocks written to use entire ME)	None
Packet Sequencing Issues (esp. in POT applications)	None
Core Component or Interface requirements or dependencies	QM CC

Scheduler Microblocks

The Scheduler microblocks include the following:

- [Chapter 17, “Fabric Scheduler For OC-48”](#)
- [Chapter 18, “Fabric Scheduler For OC-192”](#)
- [Chapter 19, “OC-48 WRR/DRR Packet Scheduler”](#)
- [Chapter 20, “OC-192 DRR Egress Scheduler”](#)
- [Chapter 21, “Egress Queue Manager \(DiffServ\) Microblock”](#)
- [Chapter 22, “Egress Scheduler \(DiffServ\) Microblock”](#)
- [Chapter 23, “TM4.1 Shaper and Scheduler Microblock”](#)

Scheduling in IXP Network Processors is completely handled in software. This provides considerable flexibility in implementing the scheduling algorithm. Intel provides several implementations of various scheduling algorithms - Round Robin, Weighted Round Robin (WRR), Deficit Round Robin, Strict Priority etc. It is anticipated that users will modify these per their requirements. The schedulers described in this document may be broadly classified into

- Cell based schedulers for the switch fabric
- Packet based schedulers for POS and Ethernet media interfaces
- TM 4.1 Shaper and Scheduler for ATM interfaces

The table below summarizes the various schedulers supported on the SDK:

Scheduler	Description	Usage	Cycle Budget
OC-48 Fabric Scheduler	Cell mode, Enqueues packets, dequeues cframes. Supports 1024 VoQs	IXP2400 ingress on the CSIX fabric side for OC-48 and 4 Gigabit Ethernet data rates	97
OC-48 WRR/DRR Packet Scheduler	Packet mode, enqueues and dequeues packets. Supports 16 ports and 16 queues per port. WRR on ports and DRR on queues within the port	IXP2400 egress on the POS and Ethernet output interfaces	97
OC48 DiffServ Packet Scheduler	Extension to WRR/DRR scheduler. Supports a hierarchy of WRR on ports, strict priority between EF and AF classes and DRR for AF class	IXP2400 egress on the POS and Ethernet output interfaces	97
OC-48 TM 4.1 Shaper Scheduler	Implements the TM 4.1 specification. Runs on 2 microengines and includes Shaper, Scheduler and Write out components. May also be run at lower speeds on fewer microengines	IXP2400/IXP2800	105 cycles per microengine
OC-192 Fabric Scheduler	Cell mode, Enqueues packets, dequeues cframes. Supports 256 VoQs	IXP2800 ingress on the CSIX fabric	57

Scheduler	Description	Usage	Cycle Budget
OC-192 WRR/DRR Packet Scheduler	Packet mode, enqueues and dequeues packets. Supports 16 ports and an unlimited number of queues per (limited only by SRAM. WRR on ports and DRR on queues in port. Runs on 3 microengines for OC-192 data rates.	IXP2800 egress for POS and Ethernet media interfaces	57 per microengine
OC-192/10G RR Port Scheduler	Simple Round Robin Scheduler provided for applications that don't need DRR and want to use fewer microengines. Supports 16 ports	IXP2800 egress for POS and Ethernet media interfaces	57

17.1 Overview

The CSIX scheduler is a context pipe-stage that is implemented as a microblock that runs on one microengine. Since this is the only code running on the microengine and it does not process packets, there is no need for a dispatch loop.

The CSIX scheduler schedules c-frames to be transmitted to the CSIX fabric. The scheduling and transmit is cell-based where each cell is a c-frame. The scheduling algorithm implemented is Round Robin among the ports on the fabric and optionally Weighted Round Robin among the queues on a port. Since this is not a QoS application and there is only one queue per port, the Weighted Round Robin scheduling is compiled out. The scheduling and transmit is done a cframe at a time. The scheduler can support up to 64 ports on the fabric and 16 QoS classes per port. Each pair of <port, class> maps on to a VoQ (Virtual Output Queue) on the fabric.

The CSIX scheduler handles flow control messages from the fabric. These messages are sent by the fabric to the Egress IXP2400, which sends them on the c-bus to the Ingress IXP2400. If the fabric asserts Xoff on a particular VoQ, the scheduler stops scheduling for the queue.

The scheduler also handles queue transition messages from the queue manager. The QM sends an enqueue transition message when a queue goes empty to non-empty and a dequeue transition message when a queue goes from non-empty to empty. These messages are sent via an NN ring. A queue is scheduled only if there is data in the queue.

The scheduler also monitors how many packet cframes are in the pipeline and if it exceeds a certain threshold, it stops scheduling.

Based on the QM messages and fabric flow control messages, the scheduler keeps hierarchical bit vectors and uses the MEv2 FFS (Find First Bit) instruction to scan them efficiently and find an eligible queue.

17.2 Assumptions, Dependencies and Risks

The following assumptions are made in the design and implementation of the CSIX scheduler block:

- The QM sends queue transition messages to the scheduler via a NN ring. The latency associated with sending these messages to the scheduler implies that the scheduler could be operating with stale information. In such cases, it is possible that the scheduler schedules a queue that has no data or does not schedule a queue that actually has data. To work around this, we run the scheduler at a slightly faster heartbeat than the QM.
- The design currently assumes that the queue and port data structures used for scheduling are cached in local memory in the scheduler. This solution does not scale to a large number of queues and ports. Currently the design can support up to 64 ports and 16 classes per port.

17.3 Data Structures

The scheduler stores some data structures in local memory and some in registers.

17.3.1 Queue and Queue Groups

Currently the scheduler supports 1024 queues—64 ports on the fabric with 16 classes per port. The information for these is stored in 32 32-bit vectors and one parent vector. Each 32-bit vector is called a queue group. Each queue group has one bit representing it in the parent bit vector or root vector.

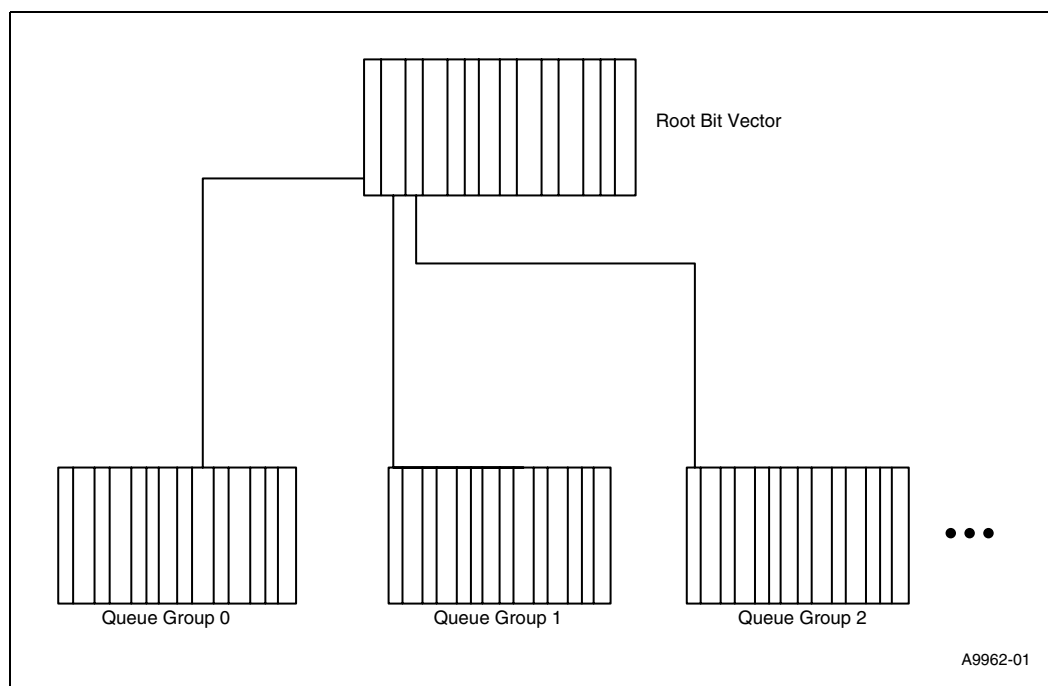
Ideally we would do Round Robin among ports and WRR on the queues in a port. This would require two 32-bit parent vectors, each with 32 16-bit vectors at the next level. This has been flattened out. One 32-bit vector is used for the parent and 32 32-bit vectors are used at the next level. This means that in each 32-bit vector on the leaf-level, there are queues from two ports. To keep the scheduling property the same as above, the following rules apply:

- The sum of the credit quantum for the 16 queues on each of the two ports must be the same. For example, if port one and two share the 32 bit vector, then the sum of the total credit for port one must equal the sum of total credit for port two.

This ensures using round robin among port one and port two.

The queue and queue group are combined to form a queue ID from 0..1023. In the case where we are not doing QoS, there is only one queue per port. So the weights for the queues are set to zero. The valid queue ID's are 0, 16, 32, 48, 64 ... and so on.

Figure 17-1. Hierarchical Bit Vector for Ingress Scheduler



17.3.2 Globals

Table 17-1 shows globals stored in absolute registers shared by all threads.

Table 17-1. Ingress Scheduler Globals

Global	Description
root_empty_vector	Root vector with one bit per queue group. If that bit is one, then the queue group has data.
root_flow_control_vector	Root vector with one bit per queue group. If that bit is one, then the queue group is not flow controlled.
packets_in_flight	Total number of c-frames, across all VOQ's, in flight—scheduled but not transmitted.
packets_scheduled	Total number of c-frames scheduled, across all VOQ's.

17.3.3 Queue Group Data Structure

For each queue group there are four words in local memory— $32 * 4 = 128$ words. The data structure has four unused bytes added to make the size a power of two thereby allowing for easy computation of the offset. The queue group information is stored at local memory offset zero.

Table 17-2. Queue Group Fields

Field	Description
queue_data_vector	Bit vector indicating which queues in the group have data.
queue_flow_control_vector	Bit vector indicating which queues in the group are not flow controlled. If a bit is set then it is NOT flow controlled.
mask	Mask used to Round Robin among the queues in the group.
Reserved	Reserved

17.3.4 Queue Data Structure

For each queue there are 2 bytes in local memory ($1024 * 1/2 = 512$ words), which hold the WRR credit information for a queue. The credit is in number of c-frames. The queue credit information is stored at local memory offset 128 (in words). Table 17-3 shows the fields of the queue.

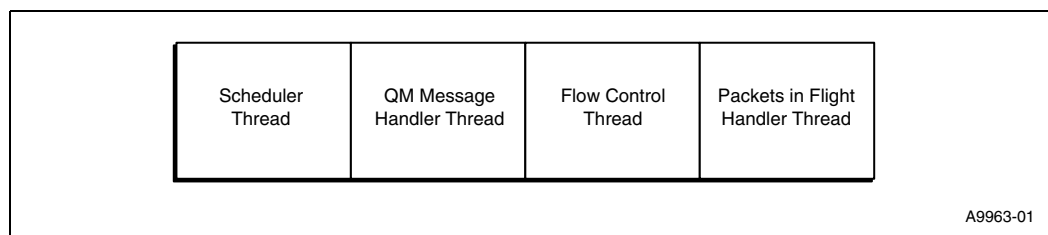
Table 17-3. Queue Fields

Field	Description
queueCurrentCredit (1 byte)	Current credit for the queue
QueueCreditIncrement (1 byte)	Credit Increment to be given to the queue at the end of every round

17.4 Design Decomposition

The scheduler contains four threads each of which is assigned a different task as shown in the figure below. All four threads run in parallel. Every 97 cycles—the minimum POS packet time—each thread is executed once.

Figure 17-2. Threads in the CSIX scheduler



17.4.1 Scheduler Thread

This thread is responsible for actually scheduling a queue and sending a dequeue request to the QM microengine. The thread runs a queue group scheduler and a queue scheduler. The queue group scheduler does Round Robin on the queue groups and finds a schedulable queue group that has at least one queue with data. The queue scheduler does WRR on the queues within the queue group and finds a schedulable queue that has data. Once the eligible queue is found, a dequeue request is sent by the scheduler thread to the QM over a scratch ring.

Every time the scheduler thread runs, it schedules an eligible queue and writes a dequeue request to the QM scratch ring before swapping out to let the other threads run.

17.4.2 QM Message Handler Thread

This thread handles messages coming back from the QM microengine. Every time a queue goes from empty to non-empty (enqueue transition), or vice-versa (dequeue transition), the QM sends a queue transition message to the scheduler via an NN ring. If the previous dequeue requested by the scheduler was to an empty queue, the QM also sends an invalid dequeue message. In every 88 cycle minimum packet time, the QM message handler thread, may in the worst case, receive an enqueue transition *and* either a dequeue transition message or an invalid dequeue message. The QM combines the enqueue and dequeue transition (or invalid dequeue) messages into a single 32-bit word. So every 88 cycles, the scheduler may receive one 32-bit message from the QM.

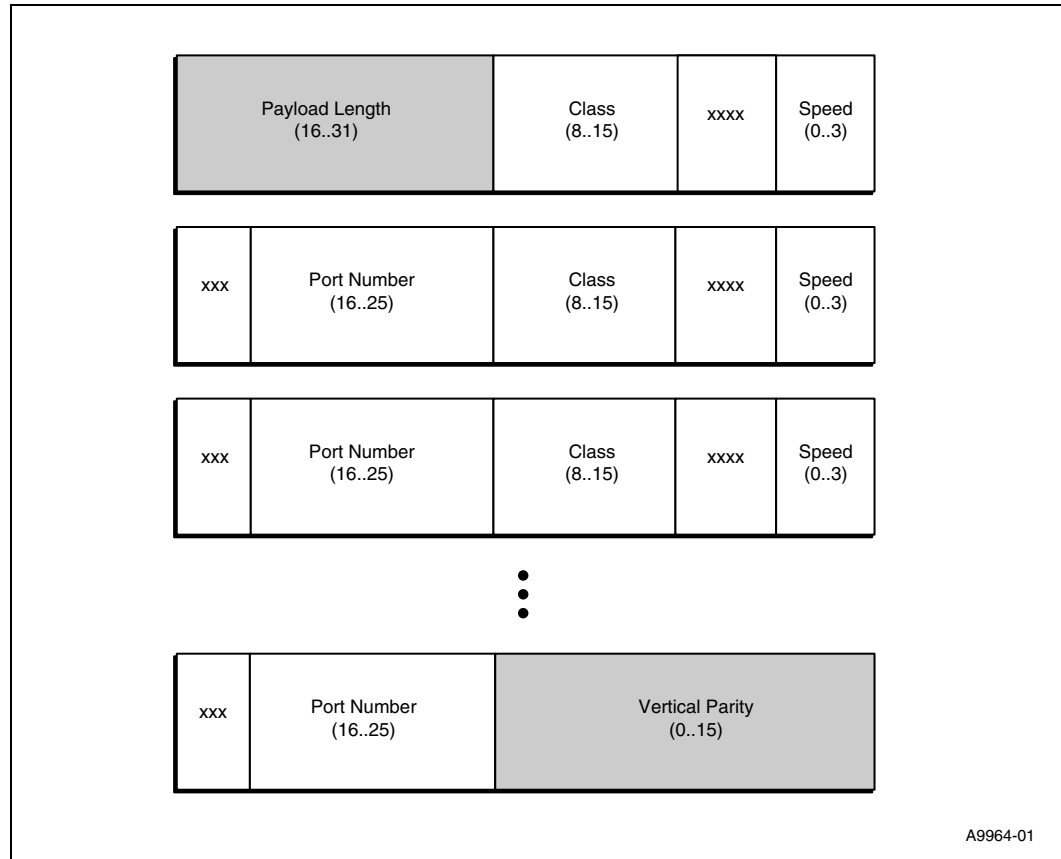
This thread updates the bit vectors indicating which queues have data based on the messages received.

17.4.3 Flow Control Handler Thread

This thread reads flow control frames from the Flow Control FIFO and updates the flow control bit vector accordingly. The format of the flow control frame is defined by the CSIX specification.

Each flow control frame read from the fabric has n messages of one word each. Additionally there is a 16-bit header, which contains the payload length and a 16-bit trailer, which contains the vertical parity. The figure below shows the format of the flow control frame.

Figure 17-3. Format of a CSIX Flow Control Frame



Only one bit of the speed parameter—bit zero is used. If bit zero is one then flow control is turned off and if bit zero is zero, then flow control is turned on. A maximum of four bits of the class field is used—supporting 16 classes—and 6 bits of the port number—supporting 64 ports.

Every time the thread runs, it handles one flow control message and then swaps out to let the other threads run.

17.4.4 Packets In Flight Handler Thread

This thread runs in an infinite loop and simply reads from the MSF the packets (c-frames) transmitted count. This is used to throttle the scheduler if the number of packets in flight—that is, the number of packets scheduled but not transmitted exceeds a certain limit.

There is a certain latency between the time flow control messages are sent by the fabric and the time that the scheduler receive/processes them. This implies that some cframes on flow-controlled queues have already been scheduled to be transmitted. The fabric has a finite amount of buffering to account for this. By using the packets in flight mechanism, we ensure that we do not overflow this buffering in the fabric.

17.5 Flow Chart for Scheduler Thread

Figures [17-4](#) and [17-5](#) show a two-page flowchart for the Scheduler thread.

Figure 17-4. Scheduler Thread: Flowchart Page 1 of 2

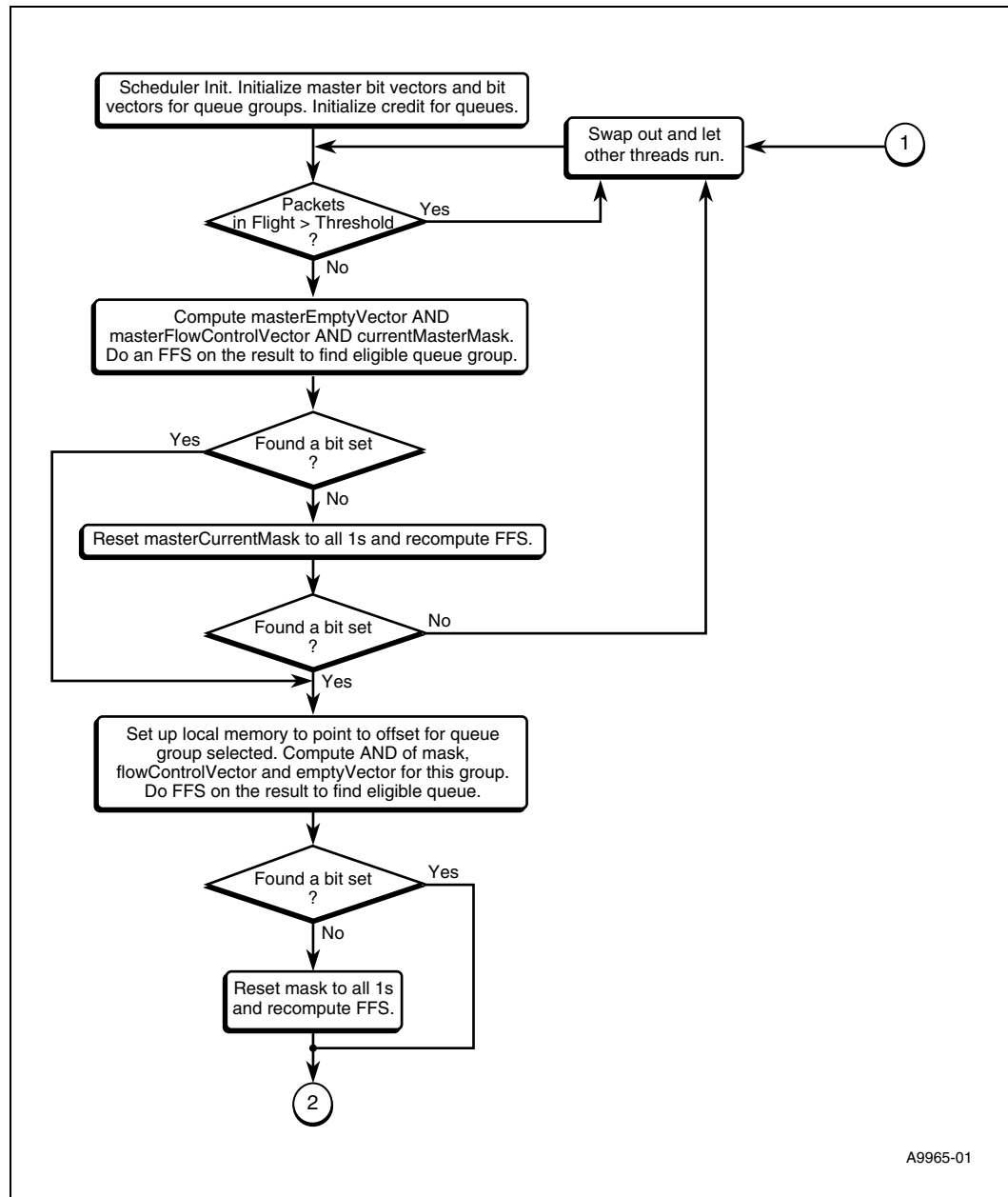
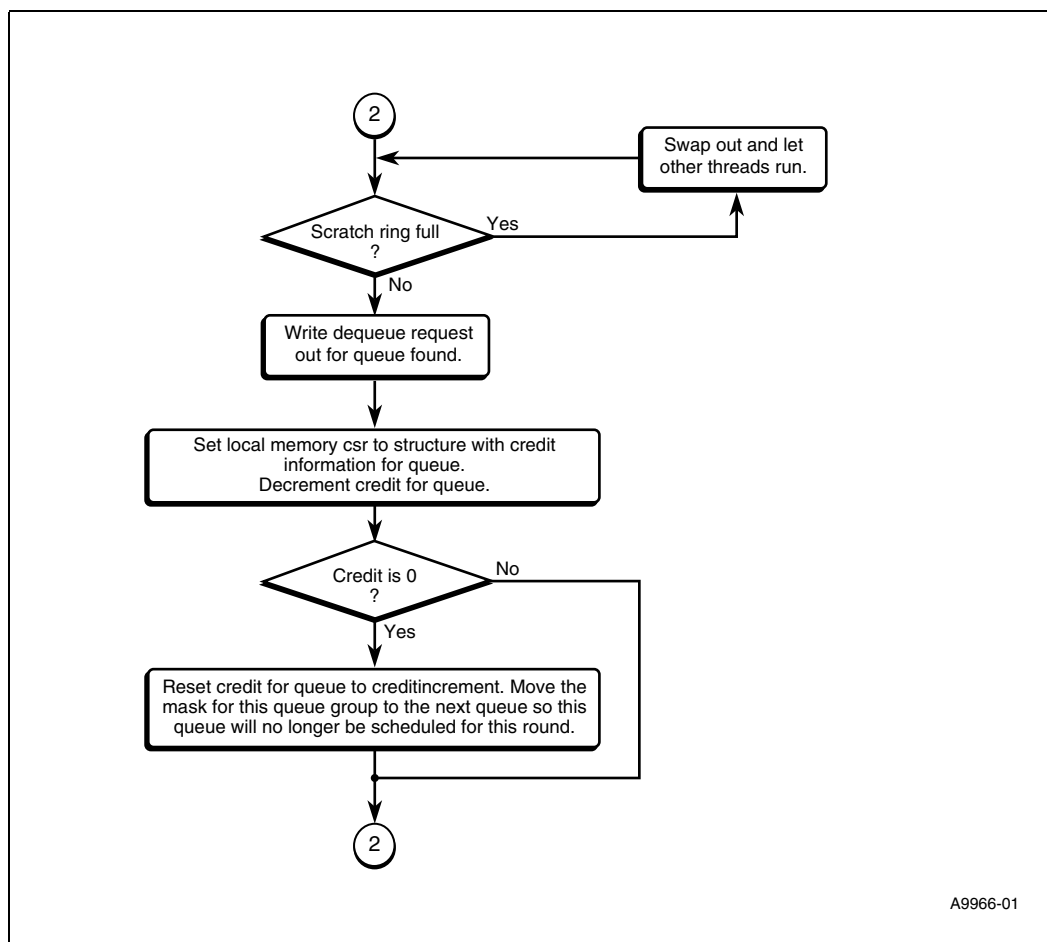
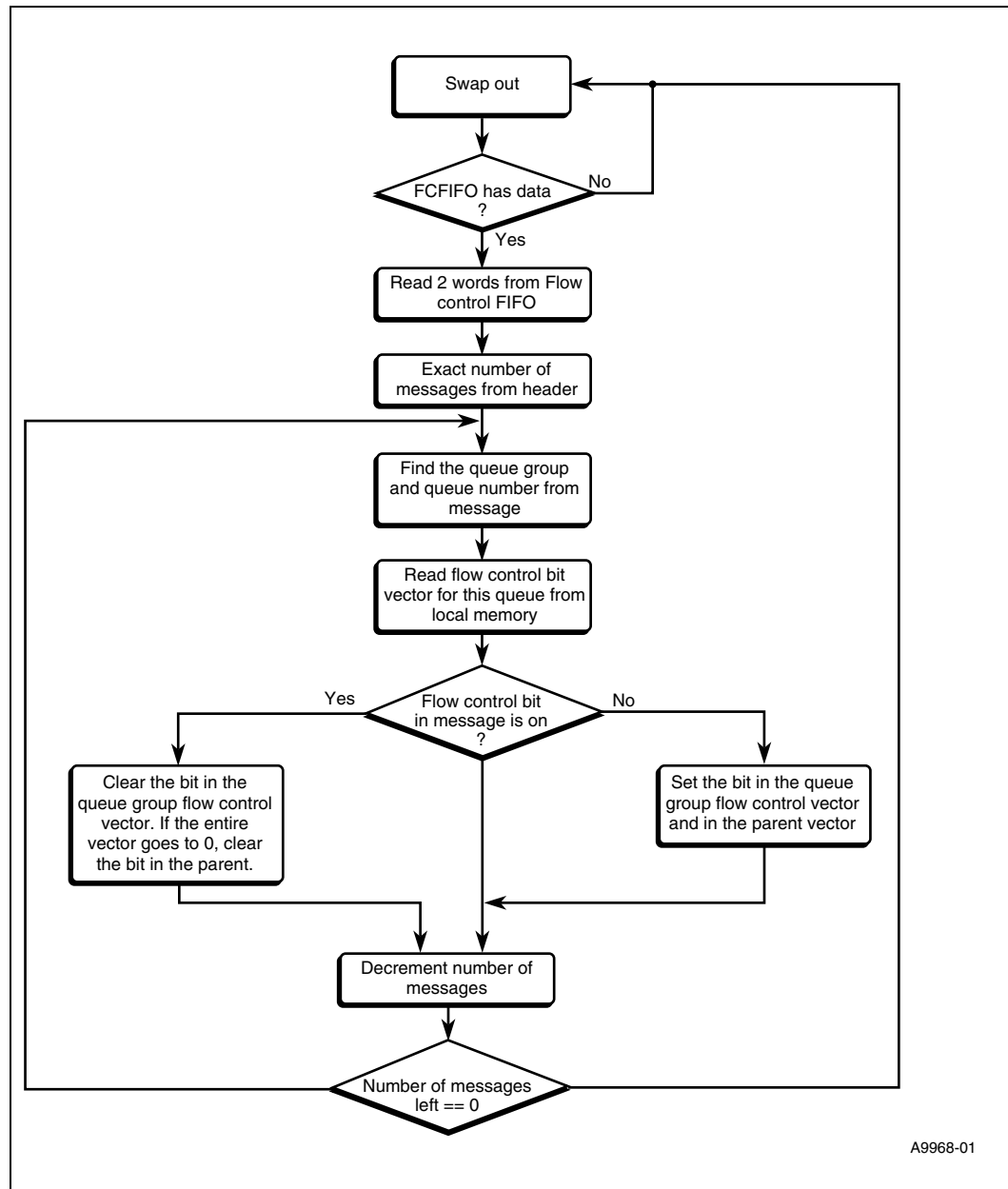


Figure 17-5. Scheduler Thread: Flowchart Page 2 of 2



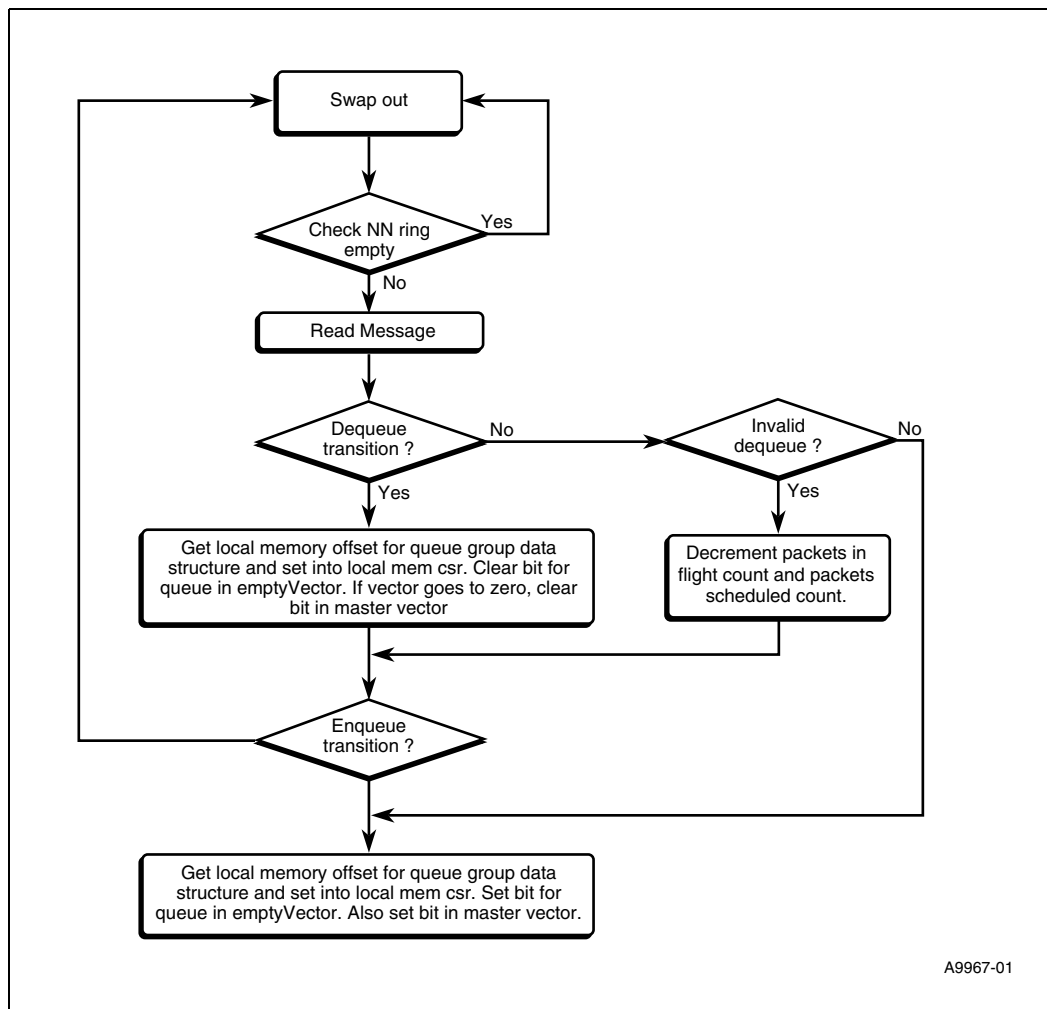
17.6 Flow Chart for Flow Control Thread

Figure 17-6. Flow Control Thread: Flowchart



17.7 Flow Chart for QM Message Handler Thread

Figure 17-7. QM Message Handler Thread: Flowchart



17.8 Performance Analysis

In the benchmark case, it is assumed that there are no flow control messages coming in from the fabric.

Assuming the above, in the worst case, in 97 cycles, the scheduler must be able to:

- Schedule one OC-48 min packet
- Handle one enqueue transition message
- Handle one dequeue transition message

The table below shows the estimated instruction cycle count for the different components of the scheduler.

Table 17-4 shows the instruction cycle estimates for the various components.

Table 17-4. Instruction Cycle Estimates for Scheduler Components

Component	Cycle Count Benchmark	Cycle Count Average	Cycle Count Worst Case
Scheduler Thread	43	35	43
QM Message Handling	31	25	31
Flow Control Message handling	7 (no flow control)	7 (no flow control)	26 for handling one FC message. (22 if messages are aggregated)
PacketsInFlight processing	7	7	7
Total	88 (97 budget)	74	107

As the table above shows, when the fabric asserts flow control, scheduling a c-frame every 97 cycles is not possible. This slows the scheduler and increases the queues size, which results in eliminating the queue transition messages. This in turn reduces the cycle count requirements.

Slowing down the scheduler operations decreases the transmit rate to the fabric which in turn eases the fabric congestion and reduces the flow control traffic from the fabric. Thus on the average, even with flow control turned on for certain queues, we should be able to keep up with min-packet line rate.

This section presents the low-level design and implementation of the Ingress CSIX Scheduler for the OC-192 design.

18.1 Overview

The CSIX scheduler is implemented as a microblock that runs on a single microengine.

Note: Since the scheduler does not run with other blocks in the same thread of execution, there is no need for a dispatch loop.

The CSIX scheduler schedules c-frames to be transmitted to the CSIX fabric. The scheduling and transmit is cell-based where each cell is a c-frame. The scheduler can support Weighted Round Robin (WRR) scheduling on up to 256 VoQ's (Virtual Output Queues) on the fabric. The VOQ's may be divided in different ways such as 64 ports with 4 classes per port or 32 ports with 8 classes per port where each pair of <port, class> maps on to a single VoQ.

The scheduler maintains a link list of active VoQ's (queues with data) in local memory. For each VoQ it tracks how many cells are currently present in the queue. The scheduler receives packet enqueue requests from the statistics microengine. It updates the VoQ information for the queue to which the packet is being enqueued and then passes the enqueue request to the Queue Manager. The scheduler also issues dequeue requests to the Queue Manager using the active list to determine which queues to schedule.

The CSIX scheduler handles flow control messages from the fabric. These messages are sent by the fabric to the egress IXP2800, which sends them on the c-bus to the ingress IXP2400. If the fabric asserts Xoff on a particular VoQ, the scheduler stops scheduling for the queue. The scheduler also monitors how many packet cframes are in the pipeline and if it exceeds a certain threshold, it stops scheduling.

18.2 Assumptions, Dependencies and Risks

The design currently assumes that the queue data structures used for scheduling are cached in local memory in the scheduler. This solution does not scale to a large number of queues and ports. Currently the design can support up to 256 VOQ's.

18.3 Data Structures

The scheduler stores some data structures in local memory and some in registers.

18.3.1 VoQ Data Structure

Currently the scheduler supports 256 VoQ's on the fabric. Table 18-1 shows each VoQ with 2 words in local memory ($256 * 2 = 512$ words).

Table 18-1. Two-word VoQ in Local Memory

LW	Bits	Size	Field	Description
0	31:31	1	Flow Control Flag	If set, the queue has flow control turned on
0	30:8	23	Cell Count	Total number of C-Frames in the queue
0	27:0	8	Next Index	Index (0..255) of the next active queue
1	31:16	16	weight	WRR weight allocated to the queue
1	15:0	16	credit	Current weight for the queue

18.3.2 Globals

Table 18-2 shows the globals stored in absolute registers shared by all threads.

Table 18-2. Globals Stored in Absolute Registers Shared by all Threads

Global	Description
active_list	Maintains information on the active list of queues
packets_in_flight	Total number of c-frames (across all VOQ's) in flight (scheduled but not transmitted)
packets_scheduled	Total number of c-frames scheduled (across all VOQ's).

Table 18-3 shows the format of the active list register.

Table 18-3. Format of the Active List Register

LW	Bits	Size	Field	Description
0	31:31	1	inactive_flag	If set, the list is empty
0	30:27	4	reserved	Reserved
0	26:16	11	prev_q_addr	11 bit index into previous active queue in the link list
0	15:11	5	Reserved2	WRR weight allocated to the queue
0	10:0	11	curr_q_addr	11 bit index into the next active queue in the link list

18.4 Design

The CSIX scheduler runs entirely in a single thread of execution. Every 57 cycles (min OC-192 POS packet time), one enqueue and one dequeue request is processed.

Table 18-1 shows the high level algorithm for the scheduler.

Figure 18-1. High Level Algorithm for the Scheduler

```
while (1)
{
    Issue read to MSF sequence number to get number of c-frames transmitted

    If (flow control message)
        Process flow control ()

    Process 8 enqueue and generate 8 dequeue requests

    If (MSF read has not finished)
        Wait for MSF read to finish

    Update the packets in flight counter
}
```

Tables 18-2 and 18-3 shows the processing of enqueue and dequeue requests.

Figure 18-2. Processing of Enqueue Requests

```
schedule() // This is called 8 times before flow control and msf read is
           handled
{
    If (enqueue request is present on the NN ring)
    {
        Read enqueue message - get queue number, total cell count
        Update the cell count for the queue in the VoQ data structure in local
        memory
        If (cell count was originally zero)
        {
            Add queue to the active list
            If (active list was originally empty)
                Clear the inactive flag and set up the active list register
        }

        If (the NN Ring to the QM is not full)
        {
            get_dequeue () --- get a queue to dequeue from
            Send enqueue and dequeue request to QM
        }
        else if the NN ring to QM is full
        {
            Wait till there is space in the ring and only send the enqueue
        }
    }
    else
    {
        get_dequeue() --- get a queue to dequeue from
        Send dequeue request to the QM
    }
}
```

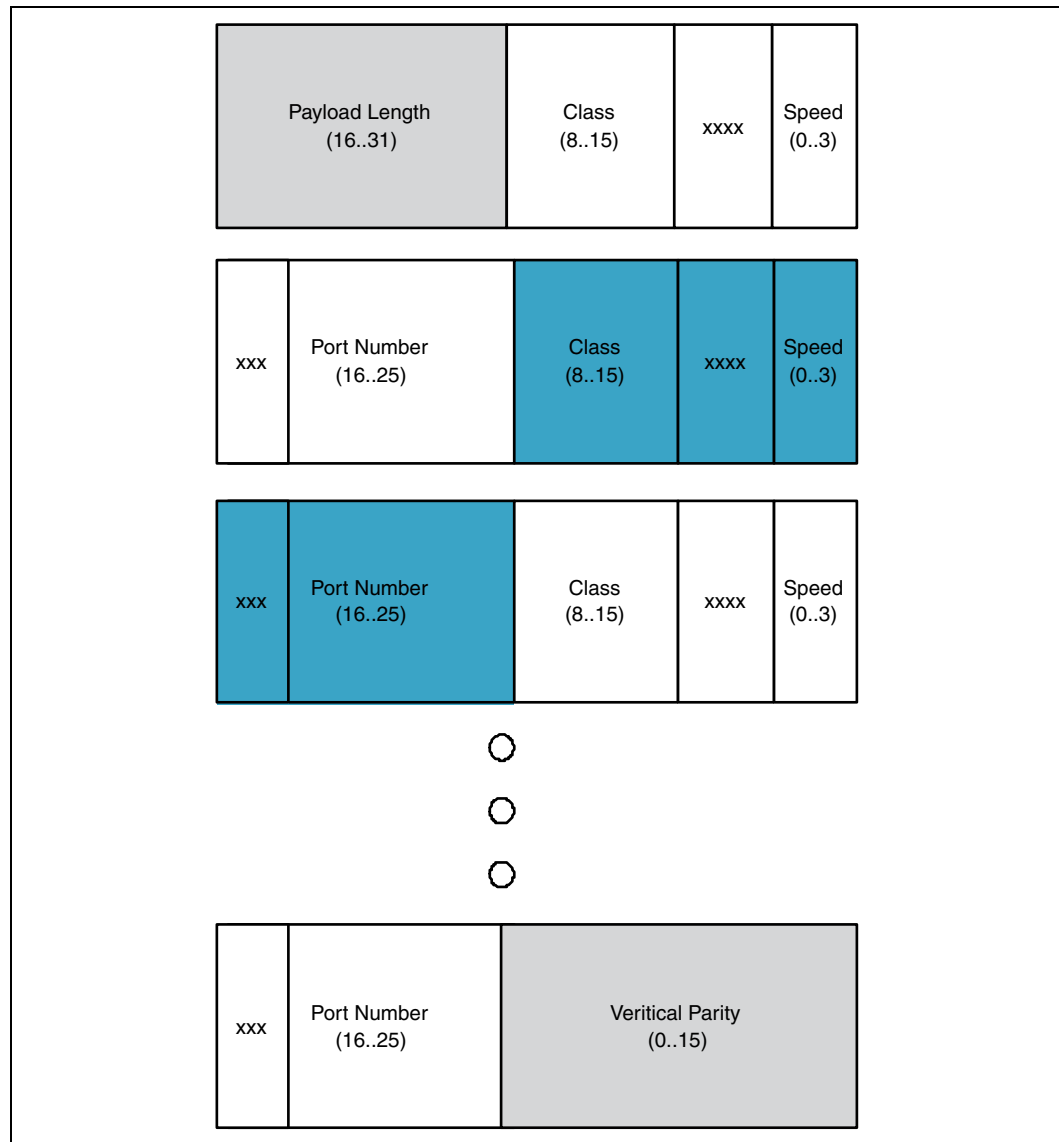
Figure 18-3. Processing of Dequeue Requests

```
get_dequeue() // get a queue to dequeue from
{
    If (packets in flight is greater than limit)
        return with no dequeue
    If (active list is empty)
        return with no dequeue
    Get the current queue from active list
    If (the queue is flow controlled)
        Remove queue from and return for this round with no dequeue
    Remove a cell from the queue
    Decrement the cell count
    If (cell count is now zero)
        Remove the queue from the link list
        If (list is empty)
            Set the inactive bit
        Decrement the credit for the queue
        If (credit goes to zero)
            Reload the credit for the queue and move to the next queue in the list
}
```

18.4.1 Flow Control Handling

After processing 8 enqueue/dequeue requests, the scheduler checks for flow control messages. If they are present, the thread reads flow control frames from the Flow Control FIFO and updates the flow control bit vector accordingly. The format of the flow control frame is defined by the CSIX specification.

Each flow control frame read from the fabric has n messages of 1 word each. Additionally there is a 16-bit header, which contains the payload length and a 16-bit trailer, which contains the vertical parity. The [Table 18-4](#) illustrates the format of the flow control frame.

Figure 18-4. Format of a CSIX Flow Control Frame


Only 1 bit of the speed parameter (bit 0) is used. If bit 0 is 1 then the flow control is turned off and if bit 0 is 0, then the flow control is turned on. A maximum of 4-bits of the class field (16 classes) and 6-bits of the port number (64 ports) are used.

18.4.2 Packets In Flight Handling

The scheduler issues a read to the MSF to get the number of tbufs transmitted. It checks for the completion of the read operation after 8 enqueue/dequeue operations. Based on the number of tbufs transmitted and that scheduled, the scheduler computes the number of c-frames in flight i.e. the number of cframes scheduled but not transmitted. If this exceeds a certain limit, then the scheduler stops scheduling.

There is a certain latency between the time flow control messages are sent by the fabric and the time that the scheduler receive/processes them. This implies that some cframes on flow-controlled queues have already been scheduled to be transmitted. The fabric has a finite amount of buffering to account for this. By using the packets in flight mechanism, we ensure that we do not overflow this buffering in the fabric.

18.5 Implementing a Hierarchical Scheduler

The design supports WRR on a flat set of 256 queues with each queue assigned a unique weight. To implement a hierarchical scheduler where the scheduling is Round Robin (RR) on ports and WRR on queues within a port, the following scheme may be used:

The weights must be assigned such that the sum of the weights for all the queues in a port must be the same for all ports.

For example, if we have 32 ports and 8 classes per port, then the total weight for each port could be 100 which is divided as needed among the 8 queues within the port. But by ensuring that all ports have the total weight of 100, the flat WRR scheduling ensures round robin scheduling on the ports.

Figure 18-5. Implementing a Hierarchal Scheduler (1 of 2)

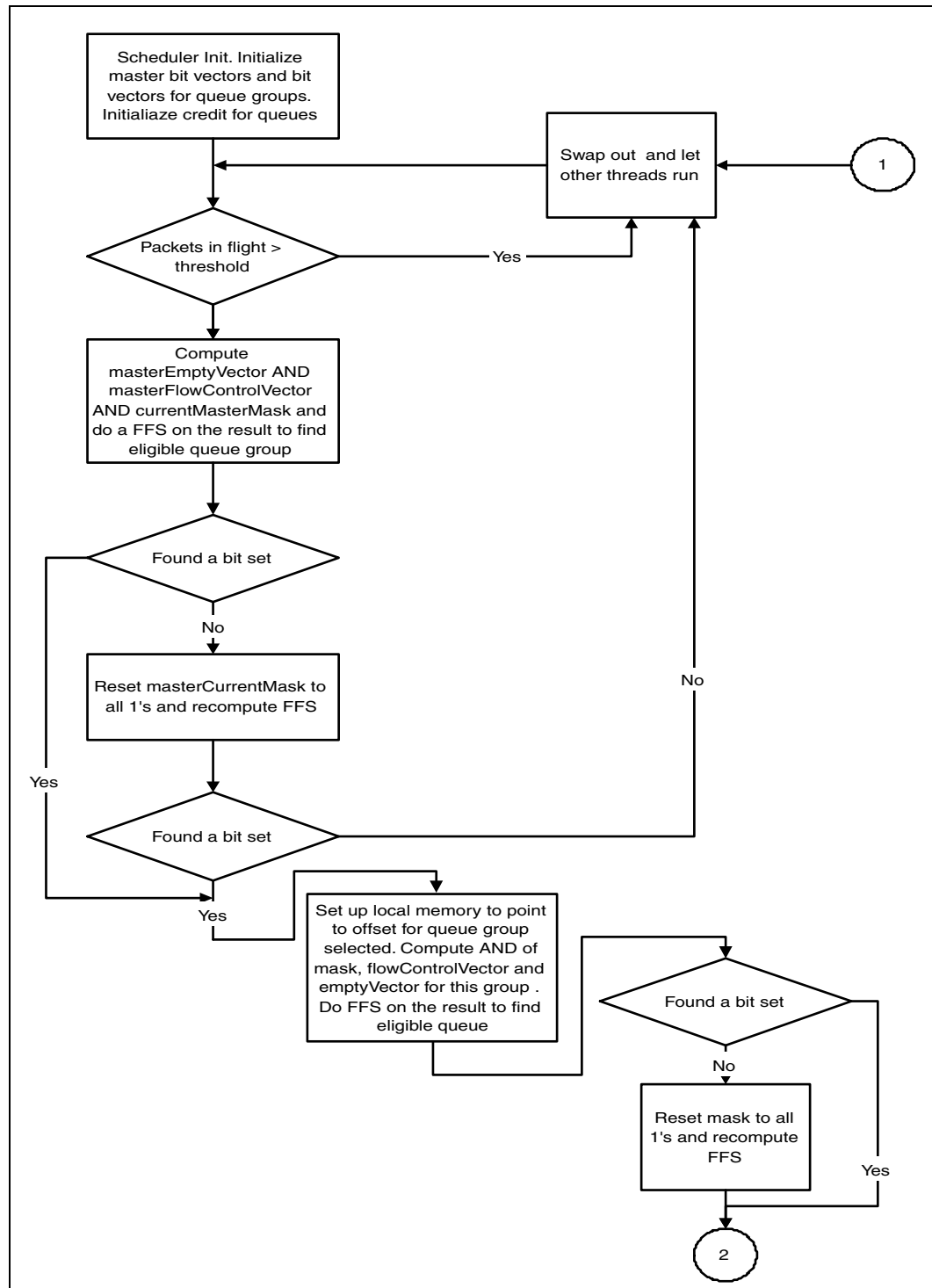
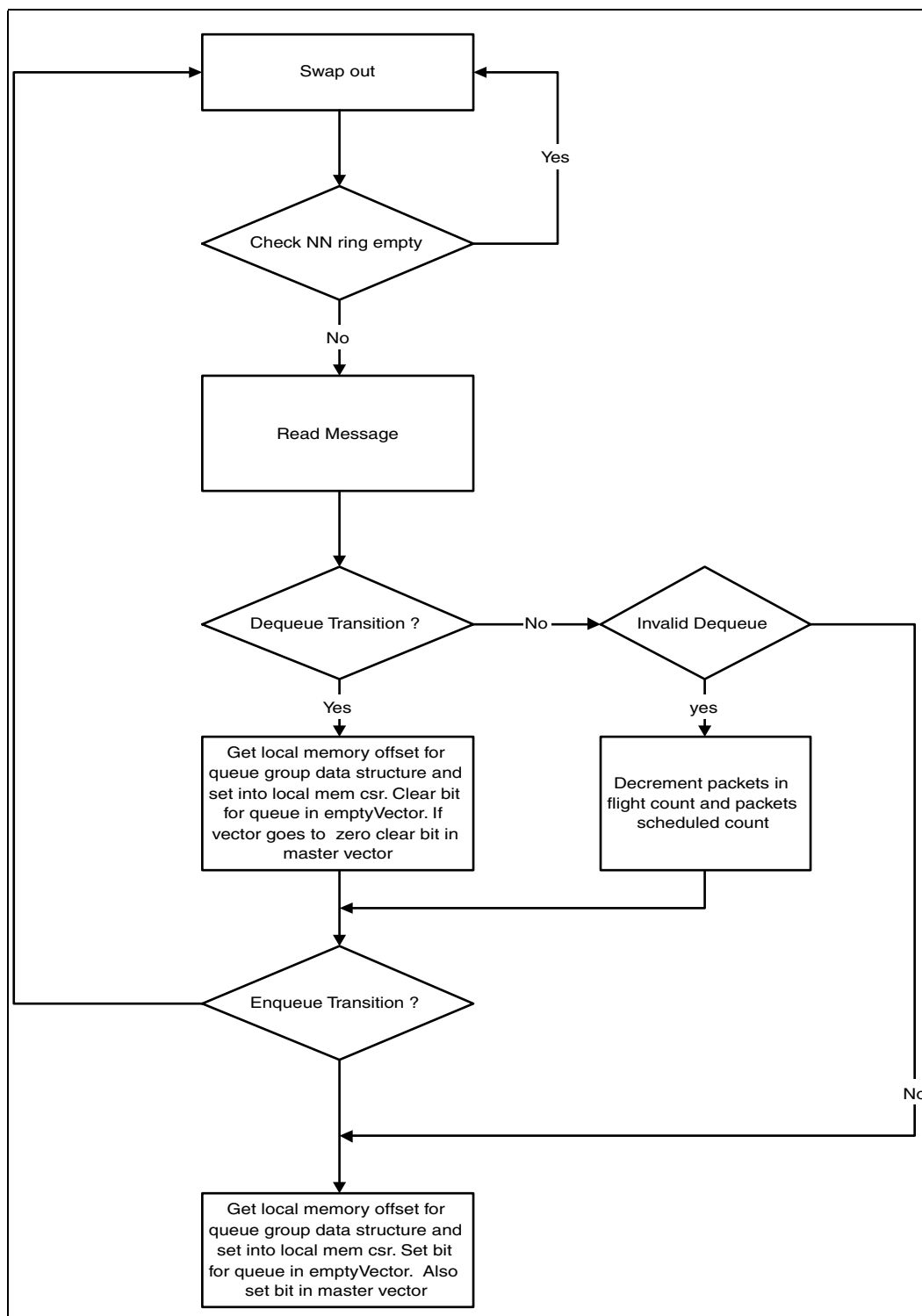


Figure 18-6. Implementing a Hierarchal Scheduler (2 of 2)



18.6 Performance Analysis

The benchmark case assumes that there are no flow control messages coming in from the fabric. Assuming the above, in the worst case, in 57 cycles, the scheduler must be able to process one enqueue and one dequeue message. [Table 18-4](#) shows the instruction cycle count for the scheduler

Table 18-4. Instruction Cycle Count for the Scheduler

Component	Actual Cycle count	Budget
Scheduler (no flow control)	52	57
Flow Control Message handling	31 for handling one FC message. (27 if messages are aggregated)	

It is not possible to schedule a c-frame every 57 cycles, when the fabric asserts flow control, which slows the scheduler operation. Slowing down the scheduler operation decreases the transmit rate to the fabric, eases the fabric congestion and reduces the flow control traffic from the fabric. Thus on an average, even with flow control turned on for certain queues, it is possible to keep up with the min-packet line rate.

18.6.1 Characterization Data

Table 18-5. Fabric Scheduler For OC-192 Microblock Characterization Data

Data	Value
General:	
Microblock Name	WRR_FLAT_SCHEDULER (2800 CSIX SCHEDULER)
Microblock Version Number	1.0
Implementation Language	microcode
Configuration Options use to gather this set of data	1. MICROENGINE 2. MICROCODE 3. CHIP_VERSION=IXP2800
Measurement Environment (tool settings)	SDK 3.5.
Performance: (cycle counts, latencies, bandwidths)	
Instruction cycle count per packet (common case packet / worst case cycle count)	Common case : 49 cycles Worst case: 56 cycles
Common-case packet/path assumptions to be documented here	Assumptions <ul style="list-style-type: none"> Queue data is in local memory Packet is forwarded in common case Message to next block is passed on NN-ring
Scratch Memory	
# of longwords read (for bandwidth calculations)	0
# of longwords written (for bandwidth calculations)	0

Table 18-5. Fabric Scheduler For OC-192 Microblock Characterization Data (Continued)

Data	Value
# and type of each atomic operation performed (for bandwidth calculations)	0
SRAM	
# of longwords read	0
# of longwords written	0
# and type of each atomic operation performed (for bandwidth calculations)	0
DRAM	
# of quadwords read	0
# of quadwords written	0
Other I/O instructions (i.e. MSF, PCI, CAP, and Hash)	0
List of dependent I/O accesses in the longest latency path	0

Per-Microengine Resources:

Control-Store Usage (# of instructions used)	202
Local Memory Footprint (# of long words used)	512 (2 long words per queue * 256 queues)
Local Memory Configuration (shared, or per-context pointer)	shared
Local Memory - # of LM pointers used	2
GPR Usage – minimum, static usage (absolute, static, globals)	1 Global GPR used as pointers to the linked list of queues with data and the linked list of queues that have completed the current WRR round
Transfer Reg. Usage – minimum, static usage	
Next Neighbor Reg. Usage – minimum, static usage	3
Signal Usage – minimum, static usage	0
CAM used? (yes or no)	No

Global Resources:

Scratch footprint (# of longwords used) – constant or formula ...	0
SRAM footprint (# of longwords used) – constant or formula ...	0
DRAM footprint (# of quadwords used) – constant or formula ...	0
Q-Array usage - # of queues used and if they need to be cached	0
CRC Unit used?	no
Hash Unit used? (yes or no)	no

MSF Usage Information:

Table 18-5. Fabric Scheduler For OC-192 Microblock Characterization Data (Continued)

Data	Value
Media Bus Configuration	CSIX
RBUF, TBUF usage	0
CBus signals	0
Other Information:	
Critical Section Length (compute cycles + memory accesses)	0
# of phases	1
Packet Metadata - fields read	None
Packet Metadata - fields written	None
Header - fields read	None
Header - fields written	None
Documentation:	
Thread Ordering Requirements	Only thread 0 is used
OS dependencies	VxWorks, Linux
Chip/hardware dependencies (i.e. Crypto Unit in 2850)	2800
Tested on which SDK Release(s)	IXA SDK 3.5
Tested on hardware? Which hardware configuration?	Yes, IXDP 2800
Tested in which applications (not an all inclusive list)	Ipv4_v6_forwarder/oc_192/ingress
Possible Configuration Options	None
Major limitations (i.e. Rx/Tx microblocks written to use entire ME)	None
Packet Sequencing Issues (esp. in POT applications)	None
Core Component or Interface requirements or dependencies	None

OC-48 WRR/DRR Packet Scheduler 19

This chapter describes the design for the packet scheduler block on the Egress IXP2400. The scheduler implements Weighted Round Robin (WRR) on ports and Deficit Round Robin (DRR) on the queues in a port.

There are significant challenges to implementing a DRR scheduler on IXP2400 because of the high memory latency and the multi-threaded (multi-microengine) programming model. This section describes how the traditional DRR scheduling algorithm was modified to work around some of these issues.

19.1 Overview

The packet scheduler is a context pipe-stage that is implemented as a microblock that runs on one microengine. Since this is the only code running on the microengine and it does not process packets, there is no need for a dispatch loop.

The packet scheduler is a frame-based scheduler. This means that it schedules a complete packet at a time. This is different from the CSIX ingress scheduler, which is a cell-based scheduler that schedules a cframe at a time.

The packet scheduler supports up to 16 virtual ports. Since these ports may have differing bandwidth requirements, the scheduler implements Weighted Round Robin (WRR) scheduling on the ports. This allows us to support different configurations—16 OC-3, 4 OC-12, 1 OC-48 and so on—simply by adjusting the weights for the ports in the scheduler.

For each port, the scheduler supports 16 QoS classes and one queue per class—total of $16 \times 16 = 256$ queues. It implements Deficit Round Robin (DRR) scheduling on the queues within a port.

Since there is no QoS requirement in the IPv4 forwarding application, we use only one of the classes per port. This means there is only one queue per port and the DRR scheduling is unused. However the same code can be reused in a QoS Diffserv application in which case the DRR scheduling is applicable.

Even though the scheduler is described in the context of a POS design, it can also be reused for Ethernet.

19.2 Assumptions, Dependencies and Risks

To implement DRR scheduling the scheduler needs to know the packet size of the packet at the head of a queue. We assume that the Queue Manager (QM) can return this to the scheduler via a next neighbor ring. Since the QM runs on a separate microengine, there is a relatively large latency in returning this to the scheduler. The DRR algorithm needs to be modified to handle this latency as explained in later sections. There is a risk that these modifications may not be as effective for bandwidth management as traditional DRR.

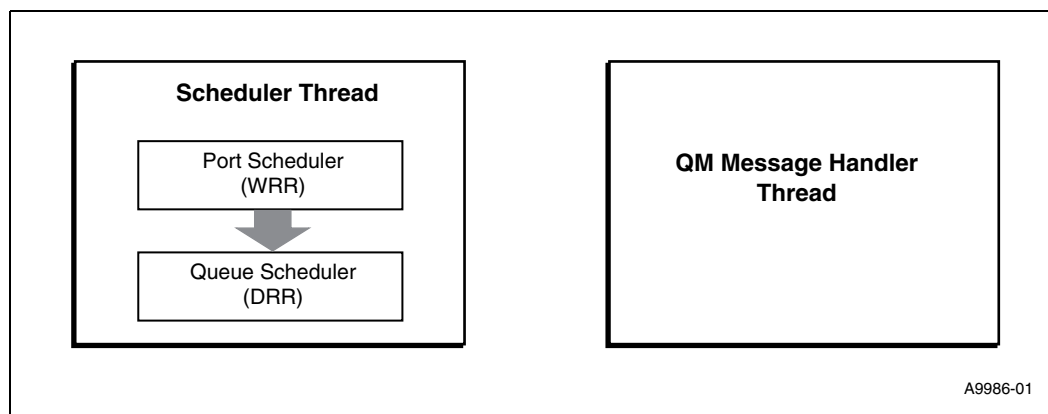
·To schedule a queue, the scheduler needs to know which queues have data. The QM sends queue transition messages to the scheduler, which indicate when a queue goes from empty to non-empty and vice versa. The latency associated with sending these messages from the QM to the scheduler implies that the scheduler could be operating with stale information. In such cases, it is possible that the scheduler schedules a queue that has no data or does not schedule a queue that actually has data. To work around this, we plan to run the scheduler at a slightly faster heartbeat than the QM. There is a risk that because of computation overhead, this is not possible.

·The design currently assumes that the queue and port data structures used for scheduling are cached in local memory in the scheduler. This solution does not scale to a large number of queues and ports since it is limited by the amount of local memory available.

19.3 Design Decomposition

The egress scheduler runs on a single microengine and has two threads: a scheduler thread and a QM message handler thread. The two share data structures stored in local memory and global (absolute) registers.

Figure 19-1. High Level Components in the Egress Scheduler



19.3.1 Scheduler thread

This thread is responsible for actually scheduling a queue and sending a dequeue request to the QM microengine. The thread runs a port scheduler and a queue scheduler. The port scheduler does WRR on the ports and finds a schedulable port that has at least one queue with data. The queue scheduler does DRR on the queues within the port and finds a schedulable queue that has data. Both schedulers use bit vectors to maintain information about which ports/queues have data and credit. Once the eligible queue is found, a dequeue request is sent by the scheduler thread to the QM.

19.3.2 QM Message Handler thread

This thread handles messages coming back from the QM microengine. The QM microengine receives dequeue requests from the scheduler. For each request, it sends a transmit message to the TX microengine and a dequeue response to the scheduler. This response has the length of the packet dequeued and an indication if the queue went from non empty to empty (dequeue transition). If the scheduler issued a dequeue to a queue that had no data (invalid dequeue), then the packet length returned is 0. The QM may also send an enqueue transition message to the scheduler when a queue goes from non-empty to empty. In every 88 cycle min packet time, the QM message handler thread, may in the worst case, receive an enqueue transition, a dequeue transition and a dequeue response. All these messages are combined into a single message consisting of 2 words. Therefore every 88 cycles, the scheduler may receive one two-word message from the QM.

This thread updates the bit vectors for credit and data based on the messages received from the QM.

19.4 Flow Control

The scheduler keeps track of the number of packets it has scheduled per port. The transmit microengine provides information to the scheduler as to how many packets were transmitted. The scheduler uses this to determine the number of packets in flight (scheduled but not transmitted) for any given port. If the number of packets in flight for a given port exceeds a pre-computed threshold, then the port is no longer scheduled until some packets are transmitted. The transmit microengine communicates the number of packets transmitted per port to the scheduler via a reflector write. 16 transfer registers are used (one per port) on the scheduler. The transmit microengine writes the number of packets transmitted into these transfer registers via the reflector bus.

19.5 Intel XScale® core Interface

An Intel XScale® core component specifies the credit information for each queue and the weight for each port. This is done via a control block shared between the XScale core and the scheduler microengine.

Note: The format for this block will be described in future revisions of the document.

19.6 Data Structures

The scheduler stores some data structures in local memory and some in registers. All the port specific data structures are stored contiguously in local memory. These are followed by queue specific data structures. Storing in this manner allows us to use both the local memory pointers LM_ACTIVE_0 and LM_ACTIVE_1 at the same time.

19.6.1 Port Specific Data Structure

The scheduler stores in local memory 8 words for each port. Thus the port data structure requires 128 words of local memory. The following data structure represents what is stored on a per port basis:

```
typedef structure __port {
    unsigned scheduleVector;    // Bit vector for queues with credit
    unsigned currentQueueMask;  // Used to round robin among queues in a port
    unsigned queueEmptyVector;  // Bit vector for queues with data for a port
    unsigned packetsScheduled;  // Packets scheduled for this port
    unsigned currentCredit;     // Current WRR credit for port
    unsigned weight;            // Weight for the port
    unsigned reserved[2];       // reserved
} port;
```

The port structure is aligned to be 8 words so that the local memory offset for a specific port may be computed from the port number using a simple shift operation.

Table 19-1. Egress Scheduler: Port-Specific Fields

Field	Description
scheduleVector	Bit vector that stores which queues in this port have positive credit. A queue with positive credit is schedulable
currentQueueMask	Bit Mask that is used to round robin through the queues in this port. It is initialized to all 1's and every time a queue is founded it is shifted left to mask out that queue for the next time. This is done until all schedulable queues are masked out. At this point we reset the bit mask to start from all 1's again.
queueEmptyVector	Bit vector that stores which queues for the port have data
packetsScheduled	Number of packets scheduled for this port
currentCredit	Current credit for WRR scheduling for this port. The credit is kept as number of packets.
weight	The weight (in number of packets) assigned to each port at the beginning of a WRR round
Reserved[2]	Reserved for aligning data structure to 32 bytes

19.6.2 Queue Specific Data Structure

The scheduler stores in local memory 2 words for each queue. Thus the queue data structure requires $256 \times 2 = 512$ words of local memory. The following data structure represents

```
typedef structure __queue {
    unsigned currentCredit;    // current credit for a queue
    unsigned creditIncrement;  // credit increment for a queue
} queue;
```

Table 19-2. Egress Scheduler: Queue-Specific Fields

Field	Description
currentCredit	Current credit for the queue in units of 128 bytes.
creditIncrement	Credit increment given to the queue at the beginning of a DRR round (in units of 128 bytes).

Since there are only 7 bits to encode the packet length, credit increments in units of the MTU/128 (rounded to the nearest power of 2) are used.

For example, for POS, if the MTU is 9k, the credit increments are in units of $9k/128 = 73$ bytes. Since dividing by 73 is not an easy operation in microcode, the credit increments are given in units of the closest power of 2, which is 128.

For Ethernet with MTU of 1518, the credit increment is in units of 16 bytes.

19.6.3 Data Structures in Registers

The data structures listed in Table 19-3 are stored in registers.

Table 19-3. Egress Scheduler: Registers for Data Structures

Register Name	Description
@PortEmptyVector	Absolute register that holds the bit vector for which ports have data.
PortCreditVector	GPR in the scheduler thread that holds a bit vector for which ports have credit
PortInitCreditVector	GPR in the scheduler thread that holds the bit vector that is used to re-initialize the credit vector at the start of every WRR round. This is useful since some of 16 ports may not be connected at all and may be simply omitted from scheduling by setting the associated bit in this vector to 0.
PortMask	GPR in the scheduler thread that is used to round robin among ports
\$\$txd_p0...\$txd_p15	16 transfer registers that hold the packet transmit count for each port. These are updated by the TX microengine via a reflector write.

19.7 Algorithm and Pseudo Code

This section explains the algorithm and provides pseudo code for the scheduler. It also details the various challenges in implementing the egress scheduler and appropriate workarounds.

19.7.1 Issues/Challenges in implementation

The following are the issues with implementing DRR:

- The packet size is not available for a queue when it is being scheduled. Once the dequeue is issued, the packet size is received N beats later where each beat is 88 cycles. Typically N is 8 beats.
- The bit vector with information on which queues have data may not be current. This could mean that a dequeue is issued on a queue that has no data.

- The scheduler schedules a packet every beat (88 cycles). This means that for large packets, the scheduler is running faster than the transmit microengine.
- The exact packet size is not available. The packet size is in multiples of `CHUNK_SIZE` which is `MTU/128`. The queue credit increments are also given in `CHUNK_SIZE` units.
- During a DRR round, a queue may go empty or come alive. While a queue is empty it should not receive credit. But queues that frequently go empty should not affect the bandwidth allocation.

The above issues are worked around with modifications to DRR as follows

- We use a scheme of negative credits. The criteria for a queue to be eligible to send are that it has data, flow control is off on the port and the credits for the queue are positive. A packet is transmitted from a queue if it meets the above criteria. Once the packet length is received, (N beats later), the packet length is decremented from the current credit of the queue. When the current credit of the queue goes negative, it can no longer transmit. When all the queues on a port go negative, one DRR round is over. Each queue gets another round of credit at this point. To ensure that all the queues are schedulable with one round of credit, we need to keep the minimum quantum for a queue as $(N * MTU)/CHUNK_SIZE$.
- If a dequeue is issued on a queue that has no data, then the QM returns the packet size as 0. This is treated as a special case. The scheduler runs slightly faster than the QM to allow it to make up for lost slots. The queue is not penalized in scheduling because no credit is decremented for the invalid dequeue.
- If the queue between TX and QM gets full due to large packets or because the scheduler is running slightly faster, then the QM does not dequeue the packet and instead return a 0 for the packet size.
- For each port the scheduler keeps a count of the number of packets in flight and not schedule a port if it exceeds a particular threshold.
- The algorithm will round robin among ports first and queues next—that is, if a queue i of port j is scheduled, the next queue scheduled is queue k of port $j + 1$. When the scheduler comes back to port j , the next queue scheduled in port j is port $i+1$. This increases the probability that the packet length is back to the original length by the time we come back to the queue
- While a queue is empty or flow control is on, its credit remains untouched. If a queue comes alive in the middle of a round, it is allowed to participate right away with the available credit. The other alternative would have been to not let the queue participate until the end of the DRR round. But that does not work well in our algorithm since we are setting a high value for the credit increment and the rounds are fairly long.

19.7.2 Scheduler Thread

This thread implements WRR on the ports and DRR on the queues within a port. The algorithm is as follows:

```
Scheduler()
{
    // Initialize all data structures
    InitScheduler()
While (1)
{
    // schedule a port
```

```

        portNumber = SchedulePort()

        // schedule a queue in that port
        queueNumber = ScheduleQueue(portNumber)

        // Issue a Dequeue Request for this queue number and port number
        RequestDequeue(portNumber, queueNumber)

        // Swap out
        SwapOut();
    }
}

// Implementation of WRR on ports
SchedulePort()
{
do {
    // AND the port empty vector with the port credit vector
    temp = @PortEmptyVector & PortCreditVector
    // AND in the Port Mask
    temp2 = temp & portMask
    // find the first bit set (ffs) in temp2
    portNumber = ffs(temp2)
    // Check if the ffs failed
    if (ffsFailed)
    {
        // Retry the ffs on temp
        portNumber = ffs(temp)
        // check if the ffs failed
        if (ffsFailed)
        {
            // port empty vector is 0. So we swap out and
            // try later. Credit vector cannot be empty because every time
            // it is updated we check if it is 0. If it is 0, we reset it.
            SwapOut();
            // go back to the top of the loop
            Continue;
        }
    } // end of ffs failed
    // We update the port mask
    PortMask = ~(1 << portNumber) - 1)
    // Set up the local memory pointer to point to the data structure for this
    //port
    port = (port*) &localmemory[portNumber << 5]
    // Now decrement the credit for the port
    port->currentCredit--;
    // Check if the current credit has gone to 0
    if (port->currentCredit == 0)
    {
        // give the port the new round of credit
        port->currentCredit = port->weight;
        // clear the appropriate bit in the PortCreditVector
    }
}

```

```

        PortCreditVector &= ~(1 << portNumber)
        // Check if there is no schedulable port
        if ((PortCreditVector & @PortEmptyVector) == 0)
        {
            // Start a new round of WRR
            PortCreditVector = InitPortCreditVector;
        } // end if credit vector is 0
    } // end if current credit for port is 0

    // Now check if packets in flight is within range. Xfer contains for each port
    // how many packets have been transmitted. It is written by a reflector write
    // by the transmit ME
    if (port->packetsScheduled - xfer[portNumber] > MAX_IN_FLIGHT)
    {
        // The port needs to skip this turn. It loses its credit for the turn
        // In this beat we will not send out any packet. We then move to the next
        // port
        SwapOut()
        Continue;
    }

    // If we get here we have a port that has met all our criteria. So we break out
    // of the while loop

    break;

} // end of the while loop

port->packetsScheduled++;

} // end of schedule port function

// Schedule a Queue within a port
ScheduleQueue()
{
    // AND the schedule and empty vectors. Empty vector must be nonzero or
    // we would not have selected the port. The schedule vector also
    // cannot be zero because the QM message handler thread resets it
    // when it goes to 0
    temp = port->scheduleVector & port->emptyVector;
    // AND the queue mask in
    temp2 = temp & port->currentQueueMask
    // find the eligible queue
    queueNumber = ffs(temp2);
    // check if no bit is set, in that case we need to recompute the ffs with
    the
    // mask reset
    If (ffsFailed)
    {
        // Do an ffs on temp alone. This cannot fail
        queueNumber = ffs(temp);
    }
}

```



```

        // Now recompute the queue mask
        port->currentQueueMask = ~((1 << queueNumber) - 1)
    }

```

19.7.3 QM Message Handling thread

This thread handles messages from the QM microengine. Based on these messages it updates the credit for the queues and various bit vectors.

```

While(1)
{
    // Check that ring has data
    if (NNRing has no data)
    {
        SwapOut();
        Continue;
    }
    // Read the message from the next neighbor ring
    message = *n$index++;

    // get packet length
    packetLength = *n$index++;

    // Check if it is an enqueue transition.
    If (QM_ENQ_TRANSITION(message))
    {
        // Get enq port number and queue number from message
        portNumber = QM_GET_ENQ_PORT_NUMBER(message)
        queueNumber = QM_GET_ENQ_QUEUE_NUMBER(message)

        port = (port*) localMemory[portNumber << 5]

        // Set the bit in the queue empty vector and in the parent for the port
        port->queueEmptyVector |= (1 << queueNumber)
        @PortEmptyVector |= (1 << portNumber)
    }

    // Get deq port number and queue number from message
    portNumber = QM_GET_DEQ_PORT_NUMBER(message)
    port = (port*) localMemory[portNumber << 5]
    queueNumber = QM_GET_DEQ_QUEUE_NUMBER(message)

    // A zero packet length signifies an invalid dequeue.
    If (QM_INVALID_DEQUEUE(message))
    {
        // decrement packets scheduled for the port
        port->packetsScheduled--;
        // swap out and then return to top of the loop
        SwapOut()
        continue;
    }
}

```

```

    }
    // Now check if the queue went empty
    if (QM_DEQUEUE_TRANSITION(message))
    {
        // clear the bit in the empty vector for queue
        port->queueEmptyVector &= ~(1 << queueNumber)
        // if the vector is 0, then mark the port as empty in the parent vector
        if (port->queueEmptyVector == 0)
            @PortEmptyVector &= ~(1 << portNumber)
    } // end if dequeue transition

    // get queue structure
    queue = (port*) localMemory[queueNumber << 3] + BASE_FOR_QUEUE_STRUCTURES

    // Now adjust the credit
    queue->currentCredit -= packetSize;

    // Check if the credit has gone negative
    if (queue->CurrentCredit <= 0)
    {
        // Give the queue the next round of credit
        queue->CurrentCredit = queue->CreditIncrement;
        // clear the bit in the schedule vector
        port->scheduleVector &= ~(1 << queueNumber)
        // if no queue is schedulable, then the DRR round is over, so reset the
        // schedule vector to all 1's
        if ((port->scheduleVector & port->queueEmptyVector) == 0)
            port->scheduleVector = 0xffffffff;
    }

    SwapOut()

} // end while loop

```

19.7.4 Flow Chart for Scheduler Thread

Figure 19-2. Scheduler Thread: Flowchart Page 1 of 2

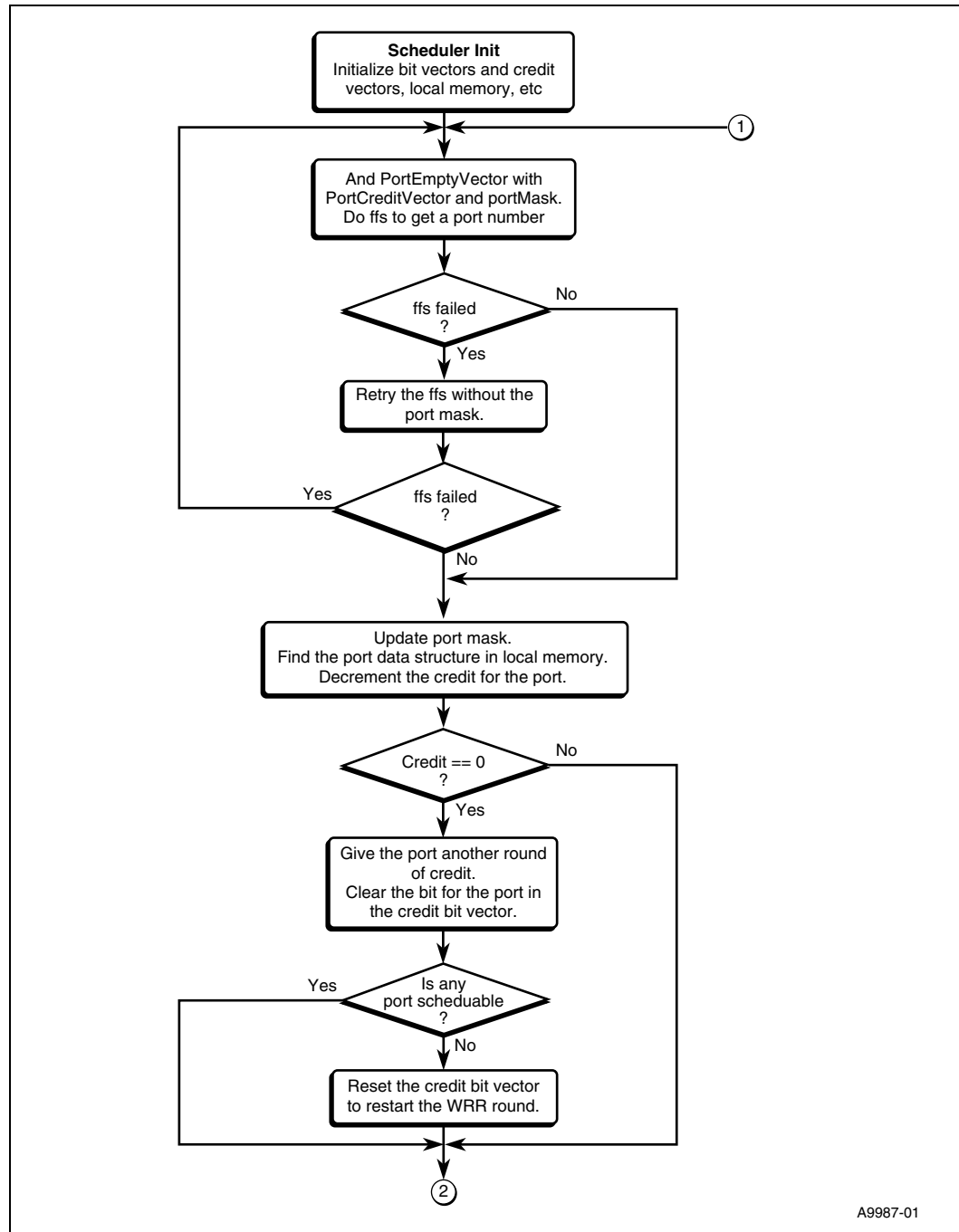
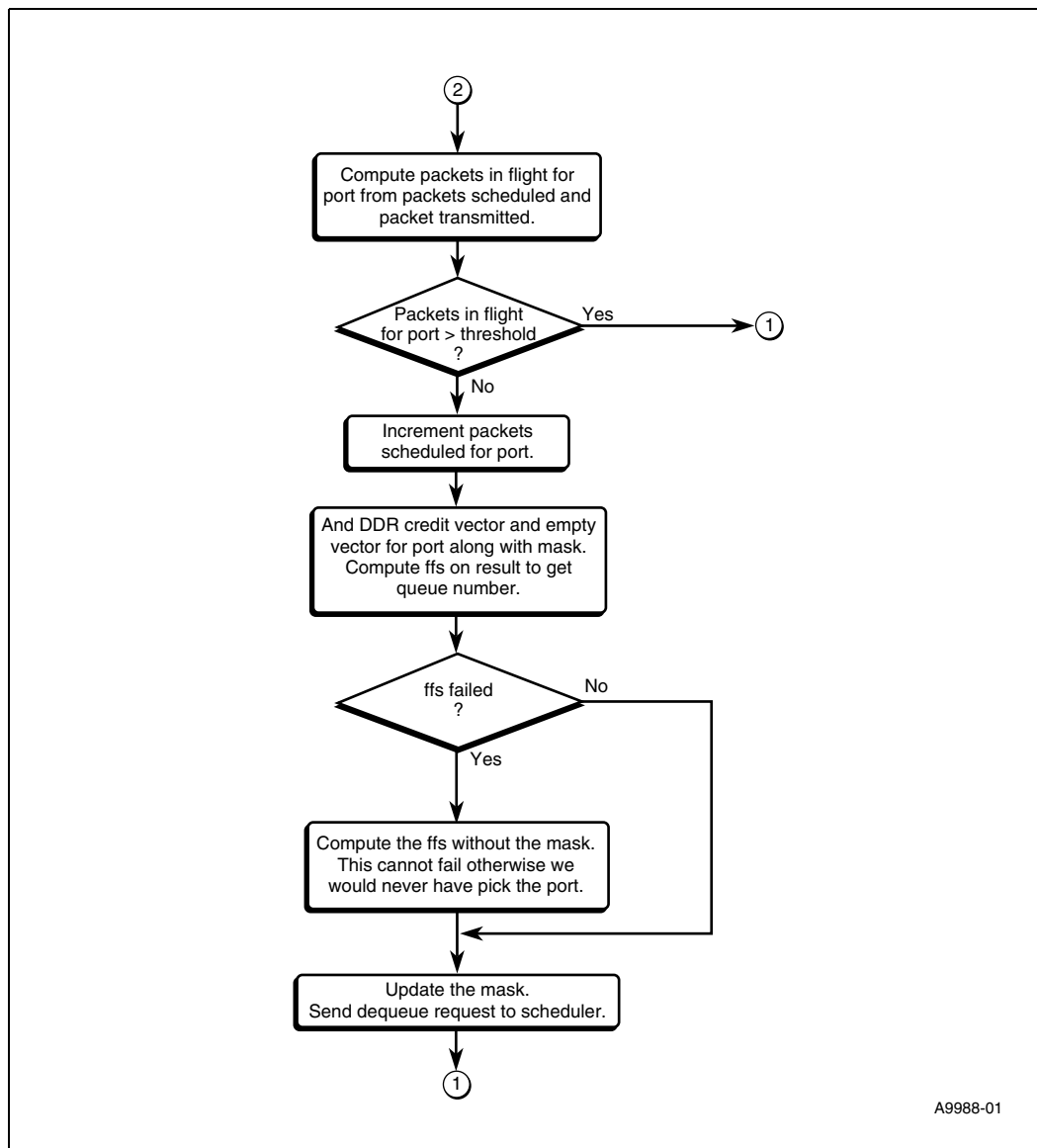


Figure 19-3. Scheduler Thread: Flowchart Page 2 of 2



19.7.5 Flow Chart for QM Message Handler Thread

Figure 19-4. Queue Manager Message Handler Thread: Flowchart Page 1 of 2

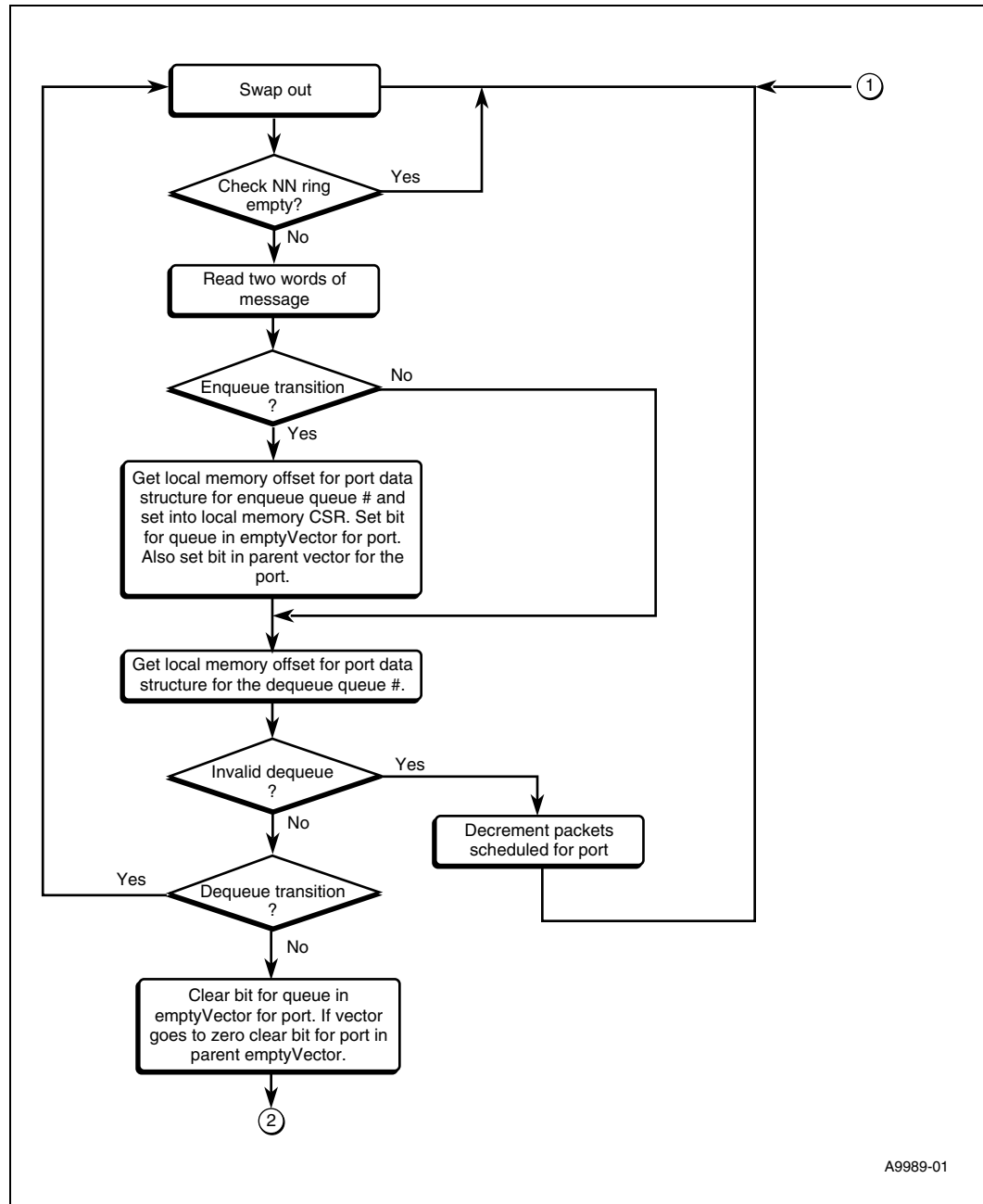
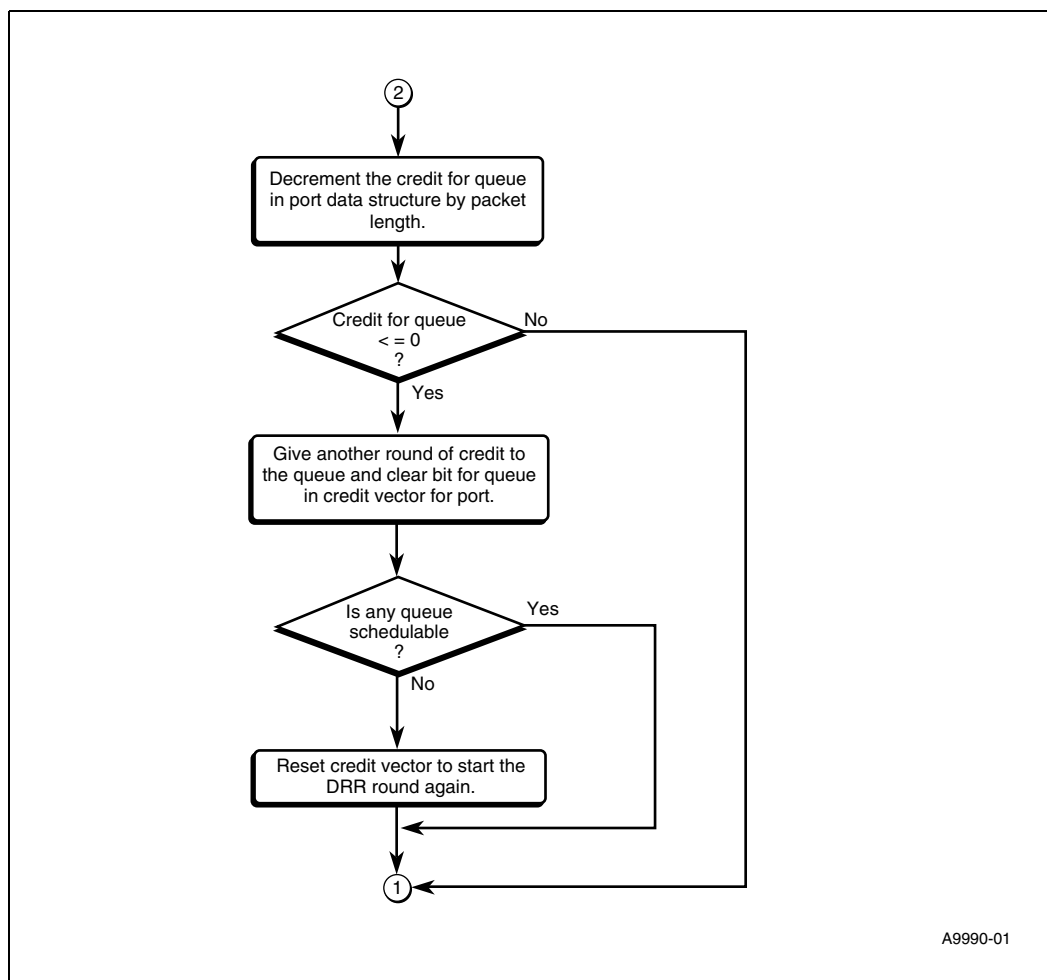


Figure 19-5. Queue Manager Message Handler Thread: Flowchart Page 2 of 2



19.8 Performance Analysis

In the worst case, the scheduler must

- Schedule 1 OC-48 min POS packet every 97 cycle beat
- Handle 1 enqueue transition message per 97 cycle beat
- Handle 1 dequeue response per 97 cycle beat

Table 19-4 shows the instruction cycle estimates for the various components.

Table 19-4. Egress Scheduler Performance Analysis: Cycle Counts

Component	Cycle Count worst case
Port Scheduler	20
Queue Scheduler	12
Packets in flight check	6
Issue Request Dequeue	6
QM Dequeue Response	23
QM Enqueue Transition Handling	10
QM Dequeue Transition	10
Total	87 (Instruction budget is 97 cycles)

Apart from a scratch write, there are no significant I/O operations in the scheduler block. So I/O latency is not an issue.

This section describes the design for the packet scheduler block on the Egress Intel® IXP2800. The scheduler implements Weighted Round Robin (WRR) on ports and modified Deficit Round Robin (DRR) on the queues in a port.

There are significant challenges to implementing a DRR scheduler for OC-192 data rates because of the high memory latency, the tight instruction budget, and the multi-threaded (multi-microengine) programming model. This section describes how the traditional DRR scheduling algorithm was modified to work around some of these issues.

20.1 Overview

The packet scheduler is implemented as a microblock that runs on 3 microengines in a pipeline. This microblock will be referred to as the Scheduler microblock, and the individual microengines the Class Scheduler block, the Count block and the Port Scheduler block respectively.

The Scheduler microblock communicates with other microblocks via Next Neighbor Ring. Within the Scheduler microblock, the blocks communicate via Next Neighbor Ring and Scratch ring.

The Scheduler microblock is a frame-based scheduler. This means that it schedules a complete packet at a time. This is different from the CSIX ingress scheduler, which is a cell-based scheduler that schedules a cframe at a time.

The Scheduler supports up to 16 virtual ports. Since these ports may have differing bandwidth requirements, it implements Weighted Round Robin (WRR) scheduling on the ports. This allows us to support different configurations (16 OC-3, 4 OC-12, 1 OC-48 etc) simply by adjusting the weights for the ports in the scheduler.

For each port, the Scheduler supports an unlimited number of queues per port (limited only by the size of SRAM). The current version supports 256 queues per port. The Scheduler implements a modified version of Deficit Round Robin (DRR) scheduling on the queues within a port.

Even though the scheduler is described in the context of a POS design, it can also be reused for Ethernet.

20.2 DRR Algorithm

20.2.1 Traditional DRR

Deficit Round Robin (DRR) scheduling specifies per queue processing to determine the queuing order for a set of active queues. Persistent data for each queue includes a Deficit Counter and a per-round Quantum.

A round is defined as one round-robin iteration over the backlogged (non-empty) queues. Conceptually the algorithm is simple:

- As rounds progress, each active queue is examined in turn. Its Quantum is added to its residual Deficit Count to produce a new Deficit Count.
- The new Deficit Count is compared with the size of the packet at the head of the queue. If the packet size is smaller than the Deficit Count, the packet is de-queued and the Deficit Count is decreased by the packet size.
- If the queue is still not empty, the packet size of the new packet at the head of the queue is compared with the Deficit Count. If smaller than the Deficit Count, the packet is de-queued and the Deficit Count is decreased by the packet size. This continues until the packet size at the head of the queue is greater than the Deficit Count. At this point, the count is stored and the round moves to the next active queue in order.
- If the queue becomes empty, then zero is stored as the residual Deficit Count, the queue is removed from the list of active queues and the round moves to the next active queue in order.

20.2.2 The Pre-Sorted DRR algorithm

The difficulties in implementing DRR as defined above include the need to know the packet sizes of the packets at the head of the active queues at dequeue time. Accessing the packet size needs a relatively large latency, and a list of active queues needs to be maintained. Besides, it takes too many cycles to access packet size, examine queue credit, and then find out the Deficit Count, which may not yet be sufficient to schedule the packet at the head of the queue.

To solve these problems, the Scheduler implements a modified DRR algorithm, namely the Pre-Sorted DRR algorithm. The Pre-Sorted DRR algorithm examines packets as they are en-queued and makes scheduling decisions that are recorded in a Pre-Sort Scheduling Array.

Using this algorithm, the dequeue order of packets closely approximates the order determined by the “traditional” DRR algorithm. The advantage is the number of cycles required is much smaller since the need to read in packet size and to manage a list of active queues is eliminated.

The Pre-sorted DRR algorithm decides the order in which packets are dequeued right at enqueue time. A few concepts that related to the algorithm are:

- Round: A round is one round-robin iteration through all the queues with packets (active queues) and enough credit to send packets. Each round is implemented as a link list in which the packet buffer handles of all packets to be dequeued in that round are kept. The Scheduler supports up to 4K rounds for EACH port; therefore, a round is uniquely identified by a pair of a port number and a round number.
- Round's link list: In this DRR Scheduler implementation, the Queue Manager doesn't manage the queues in which packets are to be enqueued—that is, the queues specified by packet handles. Instead, the Queue Manager enqueues and dequeues packets handles from and to the rounds' link list structures mentioned in the Round definition above.
- Credit Quantum: each queue is given a credit quantum. This credit indicates the total number of bytes the queue can transmit per round.

Every queue has pre-assigned parameters that define its transfer rate. In DRR schemes every active queue collects credit as time progress and uses these credits as packets arrive. If there is not enough credit when a packet arrives, that packet is scheduled to be transmitted in some later round when its queue collects enough credit.

In this Scheduler, the way the DRR scheme is accomplished is by defining a credit quantum that is assigned to each queue. For example, there are 2 active queues: queue X can transmit 10 bytes per round, queue Y can transmit 20 bytes per round, and their last packets were scheduled to be

transmitted in the current round. If queue X and Y each receive a 40-byte packet, then the packet in queue X will have to wait for 4 rounds and the one in queue Y will have to wait for 2 rounds from the current round. Assuming that current round is round 1, so the packet handle of the queue X's packet is placed in the link-list used by round 5 and the packet handle of queue Y's packet will be placed in the link-list used by round 3. These rounds are defined separately for every port. Depending upon the transmission of packets from different ports, packets are drained from the round link lists and sent to the Transmit functional block for transmit.

When a packet is enqueued, the Scheduler calculates the number of rounds this packet has to wait until it can be transmitted. The number of rounds to wait is dependent on the credit quantum of its queue, the amount of credit that the queue has already used in the current round, and the packet size. When a new packet is enqueued, the packet counter of the queue is examined. If the packet count is zero, the current round is used as the base for calculating the final round. If the existing packet count is not zero, the last round in which a packet from this queue is inserted is used as the base. The final round in which to insert the packet is the number of rounds to wait plus the base (either current round or last scheduled round). The handle to the packet will then be put in the link-list of the final round.

With this Scheduler design, the Queue Manager actually manages the rounds' link-lists. The queue into which the packets are to be enqueued can be thought of as “virtual” queues since the data structures that the Queue Manager actually works with are the rounds' link-lists.

When the Queue Manager gets a dequeue message from the Scheduler, it extracts the round number from the message, then go to the round's link list and dequeues a packet.

20.3 Assumptions and Dependencies

The following are the design assumptions and dependencies:

- If a queue is empty, the queue history is reinitialized because we do not want to accumulate credit for empty queues. Therefore, to update queue credit, the Scheduler needs to know whether a queue is empty. For each queue, there is an enqueued packets counter and a dequeued packets counter to keep tracks of the number of packets have been enqueued and dequeued, respectively. Both of these counters are kept in SRAM and they are updated at different times by different microblocks; the enqueue counter is updated by the Scheduler microblock and the dequeue counter by the Transmit Helper microblock. The Scheduler reads and compares the counters, and if they have equal values, the queue is empty. Since there is latency for SRAM write, the dequeue counter may lag the enqueue counter—that is, the write to the dequeue counter is pending when the Scheduler reads the counters for comparison. The assumption we used is that at most 16 rounds can pass while the write to update a dequeue counter is pending. Therefore, if a dequeue counter and an enqueue counter are not equal, and there are less than `MAX_ROUND_FOR_PENDING_COUNTER_UPDATE` rounds have passed, the Scheduler assumes that the dequeue counter is being updated and the queue is treated as empty. `MAX_ROUND_FOR_PENDING_COUNTER_UPDATE` is a defined constant that can be changed.
- The design currently assumes that the port data structures used for port scheduling are cached in local memory in the Port Scheduler block. This solution does not scale to a large number of ports since it is limited by the amount of local memory available.
- Rounds are implemented as link list in SRAM. The Scheduler supports up to 4K rounds per port. When a port is under flow control and no packets can be transfer for that port, the number of rounds on that port will keep increasing since packets sent to that port would need to wait

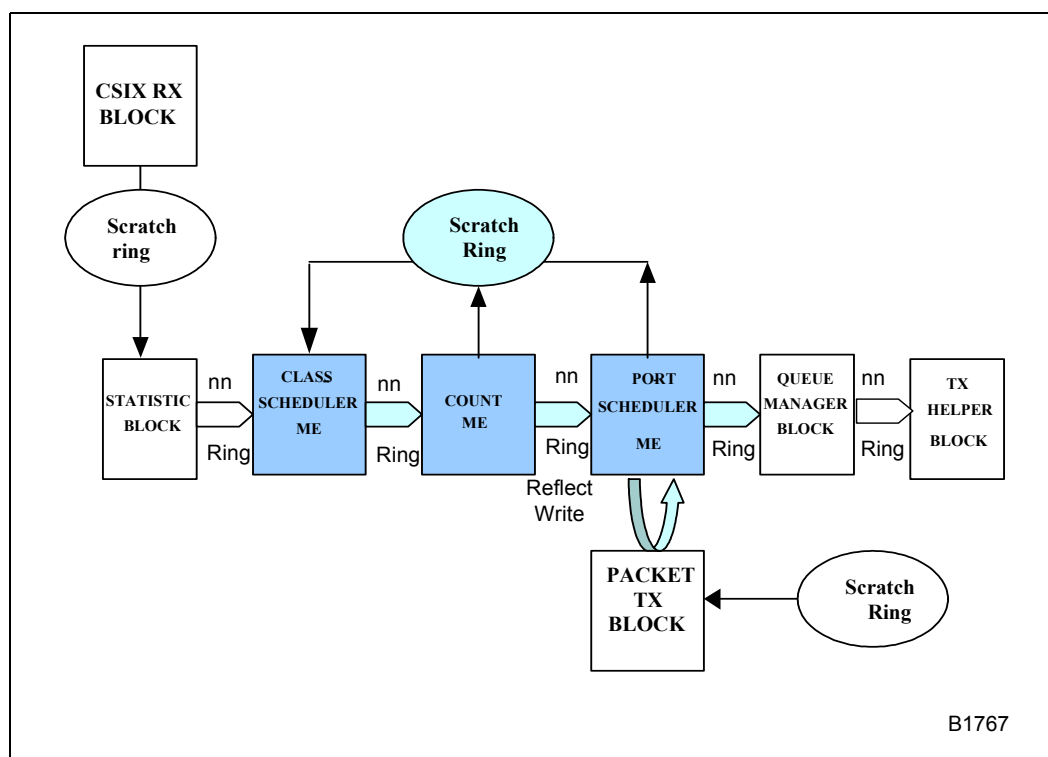
for longer and longer time. To prevent rounds overflow, other flow control mechanisms such as WRED will kick in and start dropping packets for that port.

- This scheduler keeps all the queue data structures in SRAM. It is capable of supporting a read of this structure from SRAM per packet at OC-192 data rates. This allows it to support any number of queues per port limited only by the size of SRAM.

20.4 Design Decomposition

The OC192 Egress scheduler runs on 3 MEs (Microengines), namely the Class Scheduler ME, the Count ME, and the Port Scheduler ME. Each ME communicates with the ME next to it via the Next Neighbor Ring. The Count ME and the Port Scheduler ME communicate with the “upstream” Class Scheduler block ME via a Scratch ring.

Figure 20-1. High Level Components in the Egress Scheduler



20.4.1 Enqueuing

- The Class Scheduler block ME figures out when a packet can be dequeued and sends the port and round number to the Count ME together with the SOP of the packet. The Count ME forwards the data to the Port Scheduler ME. The Port Scheduler ME sends this data to the Queue Manager who actually enqueues the packet's handle to the FIFO of that particular round.

20.4.2 Dequeueing

- Upon receiving the first packet for an empty round (a round is identified by a pair of a port number and a round number), the Port Scheduler knows that round now has packet to dequeue. It sends the port number in Next Dequeue Round Request message to the Class Scheduler block on the Scratch ring. Upon receiving the request, the Class Scheduler block sends the current round of that port in the Next Dequeue Round Response to the Count ME then increments the current round. This mechanism ensures that once the dequeuing from the FIFO of a round has started, no more packets can be scheduled to that round.
- Since the Class Scheduler block doesn't have cycles, it doesn't verify that the current round has packets before sending that round number in the Next Dequeue Round Response. The Count ME is responsible for this checking. The Count ME extracts the round number, reads the Enqueued Packets Counter for that round from SRAM. If the counter has non-zero value, the Next Dequeue Round Response is sent to the Port Scheduler block. The Count block issues a SRAM write to zero-out the Packets Counter because once a round number is sent for dequeuing, all packets in that round will be dequeued (this write does not pose any conflict with the packets counter of the enqueue round since the round was marked unavailable for enqueue as soon as it was sent as next dequeue packet round). If the counter is zero, the Next Dequeue Round Request message is sent back to the Class Scheduler block via the Scratch ring and an INVALID Next Dequeue Round Response is sent to the Port Scheduler.
- The Port Scheduler block keeps a data structure for each of the 16 ports in its local memory. It also keeps a linked list of active ports (ports that have packets ready for dequeuing). When the Port Scheduler block receives a round (identified by a port number and a round number), it's certain that the round is ready for dequeue. The Port Scheduler block updates the port data structure, and adds the port into the active ports linked list if it's not already enlisted. Dequeueing will be done from the linked list starting from the port at the head of the list.
- In the per-port data structure, the Port Scheduler block keeps 2 rounds: the current dequeue round and the next dequeue round. When a port is dequeued, all packets from its current dequeue round will be dequeued. When the packets count of the current dequeue round falls below a LOW_WATER_MARK, the Port Scheduler block sends a Next Dequeue Round Request for that port to the Class Scheduler block. If the response comes back before the current dequeue round becomes empty, the data will be kept in the next dequeue round. This “back-up” data in the next dequeue round will be moved to the current dequeue round once the current dequeue round becomes empty. If the current dequeue round becomes empty and the next dequeue round is also empty (no “back-up” data), the port will be removed from the active ports linked list. By tuning the LOW_WATER_MARK number, there will be very little waiting as long as packets keep coming to the ports.

20.4.3 Class Schedule Block

20.4.3.1 Interface

The Class Schedule block run on all 8 threads. It takes 3-longwords message from the Statistic ME. The message is passed on the Next Neighbor ring. The format of the message is as follows:

31	0
Packet size [31:0]	
Port number [31:1]	Queue number [15:0]
SOP handle [31:0]	

20.4.3.2 Pseudo Code

The following is the pseudo code for Phase 1 and Phase 2:

Phase 1

Read Scratch ring for Next Dequeue Round Request from Port Scheduler block or Count block

Read message on NN-ring

Extract packet length, queue number, port number, SOP from the message

Calculate queue address in SRAM. Use Queue address as tag to look up CAM.

If (CAM miss)

SRAM write to write back the LRU

SRAM read for Queue Data Structure and Dequeued Packets Counter of the new queue

Else (CAM hit)

SRAM read the Dequeued Packets Counter of the new queueEndif

Phase 2

If CAM miss

Move new queue data to lmem

Endif

If (Enqueue Packets Counter == Dequeue Packets Counter)

Reinitialize queue credit used and last round schedule

Else If (number of rounds have passed <

MAX_ROUNDS_FOR_PENDING_COUNTER_UPDATE)

Assume that the dequeue counter lags the enqueue counter, and the queue is actually empty.

Reinitialize queue credit used and last round schedule

Endif

Compute the round when this packet will have to wait can be dequeued.

If (valid Next Dequeue Round Request from Scratch ring)

Extract port number from Next Dequeue Round Request

Find the location of the Current Round of that port

Next Dequeue Port and Round = current round

Increment current round (this make the current round no longer available for enqueueing)

Else

Set INVALID bit in Next Dequeue Port and Round response

Endif

Send message on NN-ring to Count block

20.4.4 Count Block

20.4.4.1 Interface

The Count ME runs on all 8 threads. It gets a 3-longword message from the Class Schedule block ME. The message is passed on the Next Neighbor ring. The format of the message is as follows:

31				0		
SOP handle [31:0]						
Valid	Reserved	Enqueue Port #	Enqueue Round #[31:31]	[30:20]	[19:16]	[15:0]
Valid	Reserved	Next Dequeue Port #	Next Dequeue Round #[31:31]	[30:20]	[19:16]	[15:0]

20.4.4.2 Pseudo Code

```

Phase 1
  //First, process previous next dequeue data
  If (previous dequeue round packets count == 0)
    Extract the port from previous dequeue port and round
    Send port number to Class Scheduler Block again on Scratch ring to request
    another next dequeue round
  Set invalid bit in previous dequeue port and round so that the next block
  (Port_Scheduler) doesn't have to waste time checking empty round
  Else
    SRAM write to zero out the Packets Counter
  Endif
  //Second, process new message
  Read message from nn-ring
  If (valid enqueue message)
    Extract enqueue port and round from enqueue message
    Calculate SRAM offset of the Packets Counter for that round
    Use SRAM offset as a tag to look up CAM
    If CAM miss
      SRAM write back the LRU
      SRAM read to get Packets Counter
    Endif
  Endif
Phase 2
  If CAM miss
    Move Packets Counter value of enqueue round to local memory
  Endif
  Increment Packets Counter of enqueue round in local memory
  Endif
  Write message on nn-ring to Port Scheduler block
  //Third, process next dequeue port and round message
  Extract next dequeue port and round from Next Dequeue Port and Round message
  Calculate SRAM offset of the Packets Counter for the next dequeue round
  Use SRAM offset as a tag to look up CAM
  If CAM hit
    Save the packets count from CAM entry to previous next dequeue pkts count
    Invalidate the entry because the round will be sent for dequeue and its
    packets count will be zeroed out
  Else
    SRAM read to get Packets Counter for next dequeue round
    Save the read value to previous_next_dequeue_pkts_count
  Endif

```

20.4.5 Port Schedule Block

20.4.5.1 Interface

The Port Schedule ME run on only 1 thread. It gets a 4-longword message from the Counter ME. The message is passed on the Next Neighbor Ring. The format of the message is as follows:

31		0	
Valid[31:31]	Reserved [30:16]	Next [15:12] Port #	Next [11:0] Round #
SOP handle [31:0]			
Valid[31:31]	Reserved [30:16]	Next [15:12] Dequeue Port #	Next [11:0] DequeueRound #
Next Dequeue Round Packets Count [31:0]			

The Port Schedule ME sends a 3-longword message to the Queue Manager ME. The message is passed on the Next Neighbor Ring. The format of the message is as follows:

31		0	
Next [15:12] Dequeue Port #	Next [11:0] DequeueRound #	Enqueue [15:12] Port #	Enqueue [11:0] Round #
SOP handle [31:0]			
EOP handle [31:0]			

20.4.5.2 Pseudo Code

```
//Part 1: Process Enqueue Message
  Read message from nn-ring
  If (valid enqueue message)
    Calculate location of the Port Data Structure of the enqueue port in local
    memory
    Increment Packet Enqueued Counter in Port Data Structure
    If port is not in ports_with_packets list
      Send a Next Dequeue Round request to CLASS_SCHEDULE block via Scratch ring.
    Endif
  Endif
```



```

//Part 2: Process Dequeue Message
    If (valid next_dequeue_port_and_round)
Calculate location of the Port Data Structure of the next dequeue port in
local memory
        If (the port has zero packet in its current round)
            Add packets count to the Current Dequeue Round
            Add port in the ports_with_packets list as the last entry.
        Else
Port must have been added to the ports_with_packets list already. Just save
round number to Next Dequeue Round and packets count to the Next Dequeue
Round Packets Count in Port Data Structure.
        Endif

//Part 3: Create a dequeue message
Get the Port Data Structure pointed to by at pointer at the head
ports_with_packets list
If ((current packets count < LOW_WATER_MARK) && no pending request)
Send a Next Dequeue Round Request to CLASS_SCHEDULE block via Scratch ring.
    Endif
    If (port packet in flight > MAX_PACKET_IN_FLIGHT)
        If list only has more port
            Advance to the next port
    Endif
        If valid enqueue message
Write nn-ring message with valid enqueue and invalid dequeue port and round
    Endif
        Decrement port credit.
        If (port credit== 0)
            If (list has more ports)
                Reload credit
                Advance to the next port
            Else
                Reload credit
            Endif
        Endif
        Decrement Enqueued Packets Count
        Increment Packets Scheduled For Dequeue
        Decrement Current Round Packets Count
        If (packets in Current Round == 0 && packets in Next Round == 0)
            Remove port from list
        Else if (packets in Current Round == 0 && packets in Next Round > 0)
            Move Next Round to Current Round
        Endif

Write message to nn-ring to Queue Manager

```

20.5 Other Features

20.5.1 Flow Control

The scheduler keeps track of the number of packets it has scheduled per port. The transmit microengine provides information to the scheduler as to how many packets were transmitted. The scheduler uses this to determine the number of packets in flight (scheduled but not transmitted) for any given port. If the number of packets in flight for a given port exceeds a pre-computed threshold, then the port is no longer scheduled until some packets are transmitted. The transmit microengine communicates the number of packets transmitted per port to the scheduler via a reflector write. 16 transfer registers are used (one per port) on the scheduler. The transmit microengine writes the number of packets transmitted into these transfer registers via the reflector bus. This mechanism accomplishes flow control by not allowing scheduler to schedule much more packets in a port than transmit microengine can transmit.

20.5.2 WRED Support

The Scheduler also maintains the queue depth for WRED as the difference between the number of enqueues and dequeues issued to a “virtual” queue. The number of enqueues to a virtual queue is easily tracked on the enqueue path.

The number of dequeues is tracked as follows—when a packet is dequeued, the Tx_Helper ME looks into the packet descriptor and determine the “virtual” queue number from the port and class of the packet. We then increment the dequeue counter. When the dequeue counter and the enqueue counter have equal value, the virtual queue is empty.

20.6 Data Structures

The Scheduler works with data structures in local memory and in SRAM.

20.6.1 Queue Specific Data Structure

Queue specific data structures are stored in SRAM. Up to 16 queue structures can be cached in local memory of the Class Schedule block ME using the CAM.

Each virtual queue has 4 longwords data structure

- Lw0—current_credit_used
- Lw1—last_round_scheduled_and_quantum_shift
- Lw2—enqueued_packets_count
- Lw3—dequeued_packets_count

Table 20-1. virtual Queue with 4 longwords Data Structure

Field	Description
Current_credit_used	A value reflects how much credit of the current round has been used
last_round_scheduled_and_quantum_shift	[19:16] port [11:0] round. The last round number in which packet(s) from this queue was scheduled. When a new packet from this queue comes, the round in which the new packet will be scheduled will be calculated starting from this last round. [31:24] N where 2^N is credit quantum per round
enqueued_packets_count	The number of packets enqueued to this queue
dequeued_packets_count	For each queue, there's a 1 long word counter to keep track of dequeued packets. This longword is maintained in SRAM and ALWAYS read/written from SRAM each time it's accessed because these counters are updated (written) by Tx_Helper ME and used (read) by Class Schedule block ME.

20.6.2 Port Specific Data Structure

All the port specific data structures for 16 ports supported by the Scheduler are stored contiguously in local memory of the Port Scheduler ME. The Port Scheduler ME stores in local memory 8 words for each port. Thus the port data structure requires 128 words of local memory.

The port structure is aligned to be 8 words so that the local memory offset for a specific port may be computed from the port number using a simple shift operation. The following data structure represents what is stored on a per port basis.

- Lw0—port_packets_enqueued
- Lw1—port_credit_quantum_and_current
- Lw2—port_pkts_scheduled
- Lw3—port_next_round_req_status_next_port
- Lw4—port_next_and_curr_rnd_pkts_cnt
- Lw5—port_next_and_curr_deq_rounds
- Lw6—reserved
- Lw7—reserved

Table 20-2. Data Structure Stored on Each Port

Field	Description
port_packets_enqueued	Total number of pkts have been enqueued for the port.
port_credit_quantum_and_current	WRR credit quantum for the port and current credit the port has[31:16] credit quantum[15:0] current credit
port_pkts_scheduled	Number of packets scheduled to be dequeued for the port

Table 20-2. Data Structure Stored on Each Port

Field	Description
port_next_round_req_status_next_port	<p>[31]: round_request_pending bit.</p> <p>When the first packet is enqueued to an empty round, a Next Dequeue Round Request is sent to the Class Scheduler block.</p> <p>When the packets count in dequeue round is less than LOW_WATER_MARK, Port Scheduler block sends a Next Dequeue Round Request to Class Scheduler block. The round_request_pending bit is set to signal a request has been send and the response has not come back yet.</p> <p>[30]: got_request_response bit.</p> <p>When the response comes back, the packets count will be saved in [31:16] of port_next_and_curr_rnd_pkts_cnt. The port and round number will be saved in [31:16] of port_next_and_curr_deq_rounds.</p> <p>The got_request_response bit is set and the request_pending bit will be cleared.</p> <p>[29]: port_active bit</p> <p>When a port has been added to the active_port list, this bit is set.[0:16]: link to the structure of the next port in the active ports linked list.</p>
port_next_and_curr_rnd_pkts_cnt	[15:0] current round number and port[31:16] next round number and port
port_next_and_curr_deq_rounds	[15:0] packets count of current round[31:16] packets count of next round number and port
Reserved [2]	Reserved for aligning data structure to 32 bytes

20.6.3 Packets Counter for Each Round

A packets counter is kept for each round. The Scheduler supports up to 64K rounds (16 ports * 4K rounds per port). These per-round packets counters are kept in SRAM. Up to 16 counters can be cached in local memory of the Count ME using the CAM.

20.6.4 Current Round for Each Port

The Class Scheduler block also keeps 1 longword per port to record the current round where the packets from that port are being scheduled. This data is kept in local memory of the Class Scheduler block ME.

20.6.5 Data Structures in Registers

Table 20-3 shows the data structures are stored in registers.

Table 20-3. Data Structures Stored in Registers

Register Name	Description
\$\$txd_p0...\$\$txd_p15	16 transfer registers that hold the packet transmit count for each port. These are updated by the TX microengine via a reflector write.

20.7 Performance Analysis

Table 20-4. OC-192 DRR—Worse Cycle Count Estimate

Component	Worst Case Cycle count
Class Schedule block	57
Count	55
Port Schedule	59

20.7.1 Characterization Data

Table 20-5. OC-192 DRR Microblock Characterization Data

Data	Value
General:	
Microblock Name	CLASS_SCHEDULER, COUNT, PORT_SCHEDULER (3 blocks of the DRR OC-192 packet scheduler)
Microblock Version Number	1.0
Implementation Language	microcode
Configuration Options use to gather this set of data	1. MICROENGINE 2. MICROCODE 3. CHIP_VERSION=IXP2800
Measurement Environment (tool settings)	SDK 3.5.
Performance: (cycle counts, latencies, bandwidths)	
Instruction cycle count per packet (common case packet / worst case cycle count)	Common case: 55 cycles Worst case: 56 cycles
Common-case packet/path assumptions to be documented here	Assumptions <ul style="list-style-type: none"> Per virtual Queue data structures in SRAM, Per Round packets count data is in SRAM, Per Port data structures in local memory of Port_scheduler ME Packet is forwarded in common case Message to next block is passed on NN-ring
Scratch Memory	
# of longwords read (for bandwidth calculations)	1 (SCRATCH put)
# of longwords written (for bandwidth calculations)	1 (SCRATCH get)
# and type of each atomic operation performed (for bandwidth calculations)	0
SRAM	

Table 20-5. OC-192 DRR Microblock Characterization Data (Continued)

Data	Value
# of longwords read	Class_scheduler ME: CAM hit: 1/ CAM miss: 4 Count ME: CAM hit 0/ CAM miss: 2 Port_scheduler ME: 0
# of longwords written	Class_scheduler: CAM hit: 0/ CAM miss: 3 Count: CAM hit: 1/ Cam miss: 1 Port_scheduler : 0
# and type of each atomic operation performed (for bandwidth calculations)	0
DRAM	
# of quadwords read	0
# of quadwords written	0
Other I/O instructions (i.e. MSF, PCI, CAP, and Hash)	0
List of dependent I/O accesses in the longest latency path	2 independent I/O accesses: <ul style="list-style-type: none"> Class_scheduler: Read SRAM to find number of enqueued and dequeue packets. Calculate the round number based on the number of packets. Count: Read SRAM to find the packets enqueued for that round.

Per-Microengine Resources:

Control-Store Usage (# of instructions used)	<ul style="list-style-type: none"> Class_scheduler: 79 Count: 94 Port_scheduler: 134
Local Memory Footprint (# of long words used)	<ul style="list-style-type: none"> Class_scheduler: 64 longwords used to cache 16 virtual queue data (4 longwords per virtual queue * 16 virtual queues) Count: 16 longwords used to cache 16 rounds (1 longword per round) Port_scheduler: 64 longwords used to store data for all 16 ports (4 longwords per port * 16 ports)
Local Memory Configuration (shared, or per-context pointer)	Shared
Local Memory - # of LM pointers used	2
GPR Usage – minimum, static usage (absolute, static, globals)	
Transfer Reg. Usage – minimum, static usage	Port_Scheduler: 1 SRAM read register. Packet Tx reflect-writes the number of packets in flight into this register.
Next Neighbor Reg. Usage – minimum, static usage	<ul style="list-style-type: none"> Class_scheduler: 3 Count: 4 Port_scheduler: 3
Signal Usage – minimum, static usage	0
CAM used? (yes or no)	Yes

Table 20-5. OC-192 DRR Microblock Characterization Data (Continued)

Data	Value
Global Resources:	
Scratch footprint (# of longwords used) – constant or formula ...	1 longword: [31] valid bit [30:24] reserved [3:0] port number
SRAM footprint (# of longwords used) – constant or formula ...	0
DRAM footprint (# of quadwords used) – constant or formula ...	0
Q-Array usage - # of queues used and if they need to be cached	0
CRC Unit used?	no
Hash Unit used? (yes or no)	no
MSF Usage Information:	
Media Bus Configuration	POS
RBUF, TBUF usage	0
CBus signals	0
Other Information:	
Critical Section Length (compute cycles + memory accesses)	0
# of phases	<ul style="list-style-type: none"> • Class_scheduler: 2 phases • Count: 2 phases • Port_scheduler: 1 phase
Packet Metadata - fields read	None
Packet Metadata - fields written	None
Header - fields read	None
Header - fields written	None
Documentation:	
Thread Ordering Requirements	<ul style="list-style-type: none"> • Class_scheduler: 8 threads running in order • Count: 8 threads running in order • Port_scheduler: 1 thread
OS dependencies	VxWorks, Linux
Chip/hardware dependencies (i.e. Crypto Unit in 2850)	2800
Tested on which SDK Release(s)	IXA SDK 3.5
Tested on hardware? Which hardware configuration?	Yes, IXDP 2800
Tested in which applications (not an all inclusive list)	Ipv4_v6_forwarder/oc_192/ingress
Possible Configuration Options	None

Table 20-5. OC-192 DRR Microblock Characterization Data (Continued)

Data	Value
Major limitations (i.e. Rx/Tx microblocks written to use entire ME)	Use 3 MEs
Packet Sequencing Issues (esp. in POT applications)	None
Core Component or Interface requirements or dependencies	None

Egress Queue Manager (DiffServ) Microblock

21

21.1 Overview

The Egress Queue Manager for DiffServ is the same as the Packet Queue Manager described in [Chapter 15, “Packet Queue Manager Microblock,”](#) except for the following added features:

- Queue Manager keeps up-to-date packet count in SRAM Queue Descriptor.
- Queue Manager writes last queue idle timestamp in a Queue Descriptor.

21.2 Assumptions and Dependencies

In addition to the assumptions listed in [Chapter 15, “Packet Queue Manager Microblock”](#):

- Queue Manager does not need to check for superfluous dequeue requests. The modified Scheduler never requests a dequeue from an empty queue.
- Queue Manager sends transition messages to Scheduler on every enqueue/dequeue event.

New dependencies:

- Queue Descriptor table is accessed (read-only) by a WRED microblock.

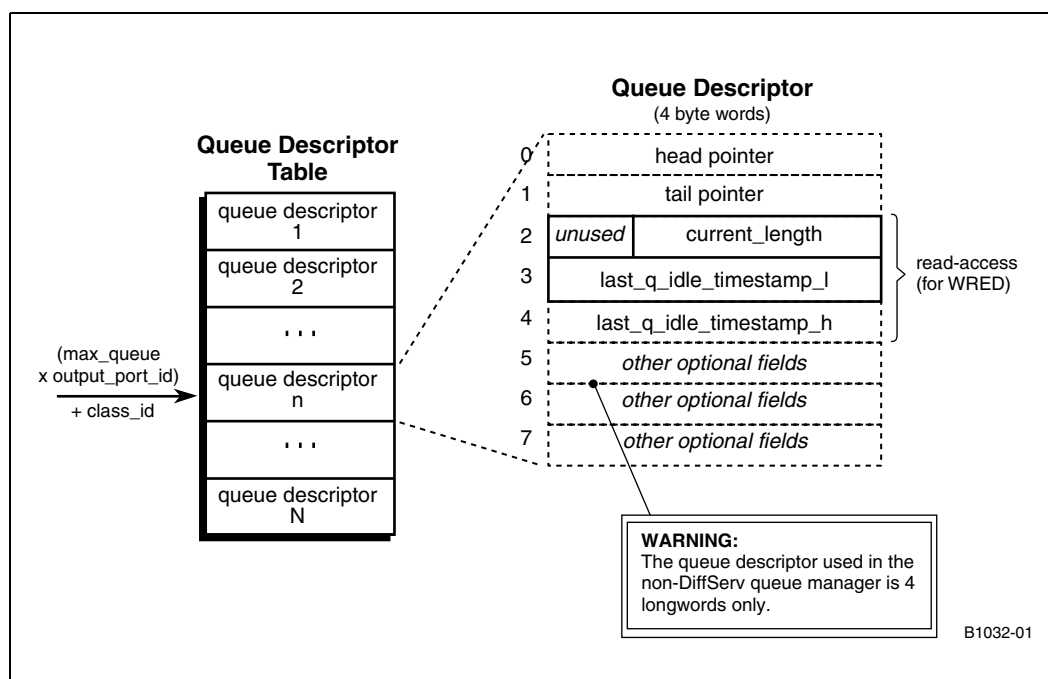
21.3 Microblock Interfaces

Both the input and output microblock variables are the same as for the Packet Queue Manager described in [Chapter 15, “Packet Queue Manager Microblock.”](#)

21.4 Data Structures

The format of a queue descriptor is compatible with SRAM Cache Array, which provides hardware acceleration for enqueue and dequeue operations. The SRAM controller uses first three 32-bit long words of a queue descriptor, and thus these fields are fixed. In particular, the lower 24 bits of the third word store the instantaneous queue length (`current_length`).

Figure 21-1. Queue Data Structures in SRAM (maintained by Queue Manager)



The remaining long words can contain any optional queue attributes. The classical RED93 algorithm needs to know a timestamp when the queue went idle last time. Since the MEv2 timestamp occupies 64 bit, two long words should be used. However, this implies extending the queue descriptor size to 8 long words. RED93 is “Random Early Detection Gateways for Congestion Avoidance” (<http://citeseer.nj.nec.com/floyd93random.html>).

21.5 Flow Chart

21.5.1 Synchronization

The synchronization algorithm is the same as for the Packet Queue Manager detailed in Chapter 15, “Packet Queue Manager Microblock.”

21.5.2 Algorithm

The following extensions to the existing algorithm are needed:

- On enqueue or dequeue operation, the QM flushes the updated packet count from Q-Array to the Queue Descriptors table in SRAM. This prevents WRED from operating on stale information.

To minimize SRAM controller load, QM should flush packet count only on CAM hits. Such approach saves SRAM accesses at cost of acceptable WRED inaccuracy. On CAM miss, the updated packet count differs only by one from the value stored in SRAM. This is acceptable for WRED, since it estimates the long-term average.

- The Queue Manager no longer needs to track the number of dequeue request issued.
- On every dequeue operation, which results in a queue becoming empty, the QM saves the current timestamp in the Queue Descriptors table in SRAM memory. This allows WRED to estimate the average queue length on queue empty condition.
- The QM sends a message to egress Scheduler on every enqueue/dequeue event. This allows egress Scheduler to know the current queue length. In this way, the Scheduler never sends invalid dequeue requests.

21.6 Micro-code Budget

21.6.1 Performance Analysis

Table 21-1. Cycle Count Table (including unfilled defers)

Phase	Worst case
Existing QM implementation	81 ¹
Enhancements	+5
Total	86

1. See Chapter 13, "Queue Manager For OC-48 Microblock"

Table 21-2. I/O Latency Analysis Table

Operation	I/O type	Accesses	Transfer size (in 32-bit long words)
Read enqueue request	scratch read	1	2
Read dequeue request	scratch read	1	1
Load queue descriptor from SRAM to Q-Array	SRAM rd_qdesc	2	3
Evict queue descriptor from Q-Array to SRAM	SRAM wr_qdesc	2	3
Flush packet count (only if queue descriptor cached)	SRAM wr_qdesc_count	2	1
Save last idle timestamp for a queue (if a queue goes empty)	SRAM write	1	2
SRAM enqueue operation	SRAM enqueue	1	-
SRAM dequeue operation	SRAM dequeue	1	-
Write transmit request to scratch ring	scratch write	1	1

Table 21-3. Memory Access Summary Table

I/O type	Accesses
SRAM	5
Scratch	3

21.6.2 Memory Footprint Analysis

Table 21-4. SRAM Footprint

SRAM Data Structures	Size (in bytes)
Queue Descriptor table (entry is 8 long words)	8 kB (256 queues)
Total	8 kB

Table 21-5. Scratchpad Footprint¹

Scratchpad Data Structures	Size
Dequeue requests	2 kB
Enqueue requests	2 kB
Total	4 kB

1. The table shows only scratch rings, where QM acts as a message consumer

Table 21-6. Code Store Footprint

Code Store	Size (instruction)
Existing Implementation	298
Enhancements	267

Egress Scheduler (DiffServ) Microblock

22

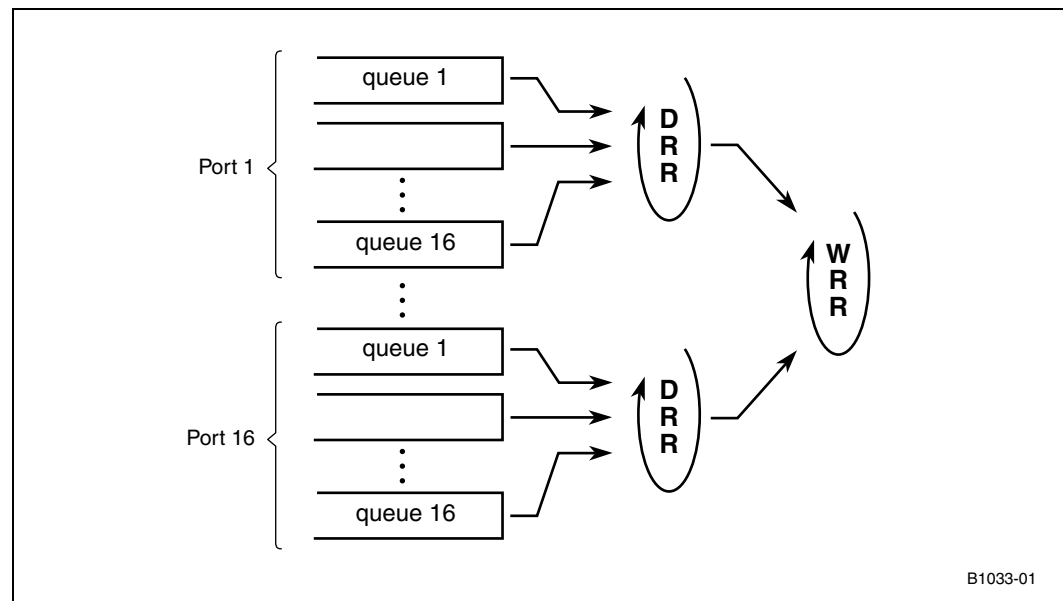
22.1 Overview

This section describes enhancements made to the egress packet scheduler depicted in [Chapter 19](#), “OC-48 WRR/DRR Packet Scheduler.” The high-level design primarily focuses on new elements. However, it also briefly recalls existing design, when needed, to understand the architecture concepts.

The existing egress Scheduler handles up to 16 virtual ports. It implements a packet-based scheduler. This means that it schedules a complete packet at a time. This is different from the ingress CSIX scheduler which is cell-based and schedules a cframe at a time.

Since the output ports may have different bandwidth requirements, the scheduler employs Weighted Round Robin (WRR) scheduling on the ports. By adjusting WRR weights, different configurations—16 OC-3, 4 OC-12, 1 OC-48, and others—can be easily supported. For each port, the scheduler distinguishes 16 QoS classes and one queue per class (total of $16 \times 16 = 256$ queues). It approximates Deficit Round Robin (DRR) scheduling on the queues within a port.

Figure 22-1. Existing implementation of WRR/DRR egress scheduler

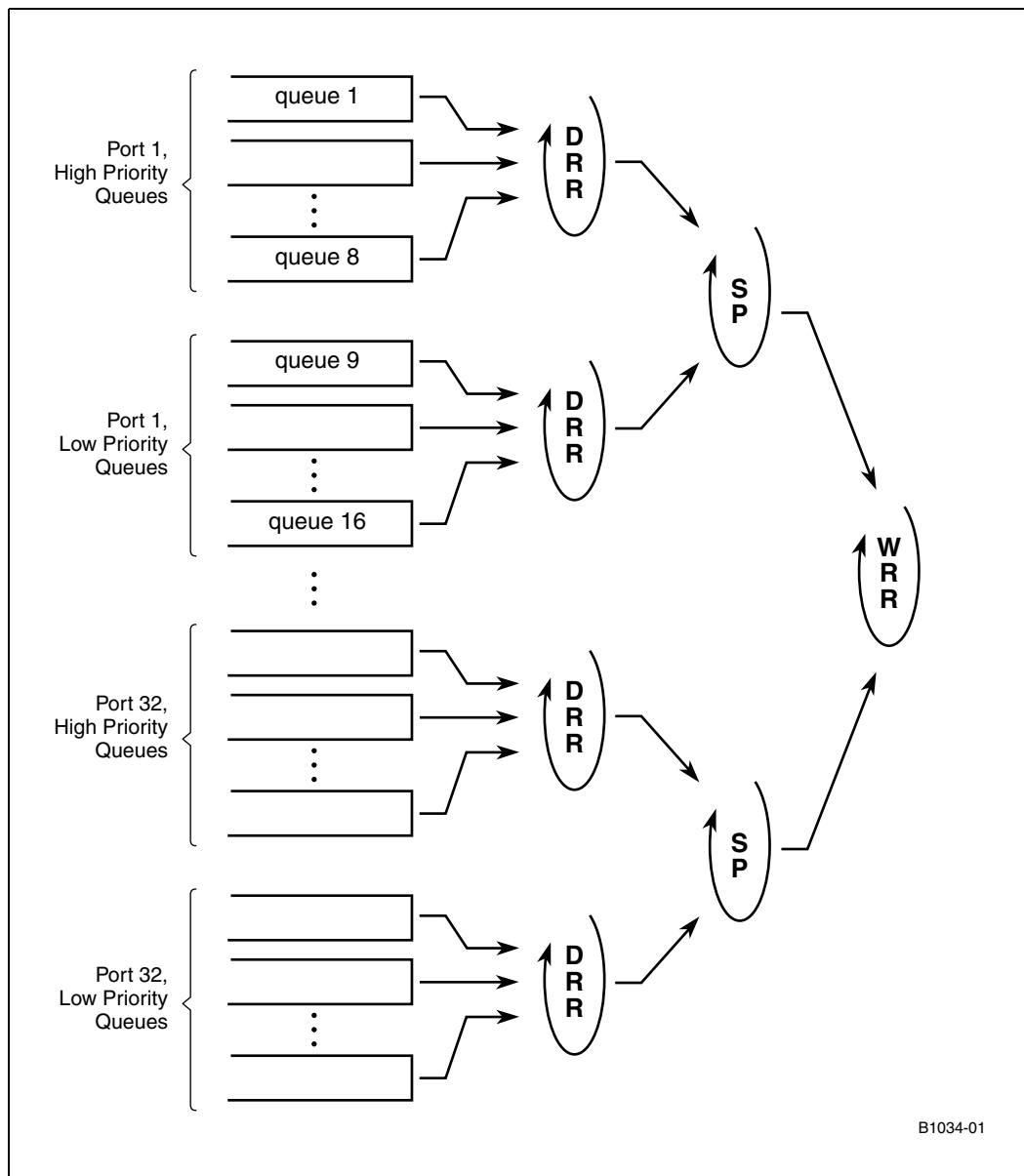


The extended design adds a third level of scheduling: Strict Priority (SP). On each port, the scheduler distinguishes two sets of queues: 8 high priority queues and 8 low priority queues. Two priority levels enable “canonical” implementation of DiffServ Expedited Forwarding PHB. The enhanced, three-level hierarchic scheduler is illustrated in [Figure 22-2](#):

- Weighted Round Robin (WRR) between 16 ports

- Strict Priority (SP) between 2 queue groups within a port
- Deficit Round Robin (DRR) between 8 queues within a queue group

Figure 22-2. Enhanced Egress Scheduler with Hierarchic WRR/SP/DRR



In addition, the enhanced design overcomes some inaccuracies of DRR approximation. In particular, the Scheduler:

- Does not generate dequeue requests for queues that are empty.
- Minimizes delay between queue status change and scheduler reaction to this change.

22.2 Assumptions and Dependencies

As noted in the [Chapter 19, “OC-48 WRR/DRR Packet Scheduler,”](#) there are significant challenges to implement a DRR scheduler on the IXP2400 Network Processor. A scheduler does not have direct access to the first packet in a queue, so it cannot check the packet length before it issues a dequeue request. The scheduler receives a packet length in a dequeue response message from the Queue Manager microblock. Unfortunately, there is high communication latency between Queue Manager and Scheduler blocks.

[Chapter 19, “OC-48 WRR/DRR Packet Scheduler”](#) introduces a solution of “predictive” DRR which first schedules packets and then checks if a scheduling decision was allowable. This brings in several issues:

- A scheduling decision can concern an empty queue.
The “predictive” DRR continues scheduling a queue until Queue Manager responds with a queue empty transition. Due to latency in receiving feedback, a scheduler can send more dequeue requests than the actual number of packets in a queue.
In [Chapter 19, “OC-48 WRR/DRR Packet Scheduler,”](#) an indented workaround is that the scheduler does round robin on all non-empty queues so that intervals between successive references to the same queue are as long as possible. This increases the probability that dequeue request feedback is received on time.
Unfortunately, this workaround does not work well when combined with priority queuing. Basically, a scheduler tries to exhaust a priority queue as fast as possible, so it generates dequeue requests one after another. For instance, if there is one port and one high priority queue with one packet, then a scheduler would issue at least 8 dequeue requests to the high priority queue before it switched to low priority queues.
- A scheduler can skip a queue (considering it empty), even if it has packets.
According to [Chapter 19, “OC-48 WRR/DRR Packet Scheduler,”](#) the egress scheduler does not handle enqueue transitions from QM right away but allows them to backlog in a NN ring. As long as the scheduler runs faster than the line rate, the backlogging is only temporarily. Nevertheless, with present design a scheduler can select low-priority packet even if there is high-priority enqueue waiting on the NN ring.
The above effect can be neglected if only DRR algorithm is used. However, the latency incurred on the NN ring can become significant and not acceptable for delay-sensitive traffic such as Expedited Forwarding.
- A scheduling decision can be unfair—that is, it can exceed the credit granted to a queue.
The “predictive” DRR schedules a queue until it receives a dequeue response that causes a queue credit to drop below zero. Then, it resets the queue credit to the quantum increment and stops scheduling a queue in this DRR round. Because of feedback latency, the scheduler can issue more dequeue requests than it is necessary to exhaust the queue credit. The excessive requests are not counted in a DRR balance, thus leading to unfair behavior.
To minimize the amount of unfair scheduling decisions, [Chapter 19, “OC-48 WRR/DRR Packet Scheduler”](#) assumes fairly large DRR credits (at minimum $N \times \text{MTU}$ bytes, where a typical N is 8). In a POS scenario with MTU set to 9kB, the minimum credit is 72 kB.

The subsequent sections describe how the design in [Chapter 19, “OC-48 WRR/DRR Packet Scheduler”](#) was modified to solve the above problems. In particular, the following design assumptions were made:

- Scheduler keeps track of the number of packets in a queue. In this way, it can avoid referring to an empty queue.

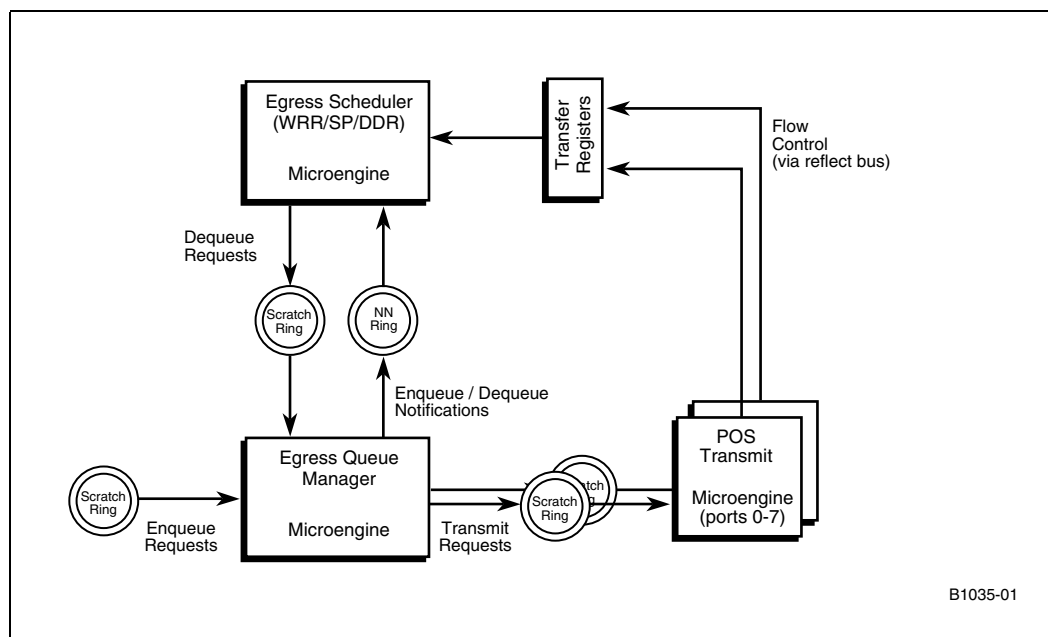
- Scheduler reads all enqueue/dequeue messages pending on the NN ring, so it does not operate on stale information.
- Scheduler allows at most 12 outstanding dequeue requests (that is dequeue requests sent to QM for which no response has been received). This is to bound maximum communication latency between Scheduler and QM.
- Before starting a next DRR round, scheduler penalizes a queue with the amount of unfairly consumed credits in the previous DRR.

22.3 Microblock Interfaces

The external interfaces of a scheduler microblock are almost the same as described in [Chapter 19, “OC-48 WRR/DRR Packet Scheduler.”](#) This section recalls them for completeness and highlights modifications. The scheduler microblock runs as a context pipeline on one microengine. All external interfaces are described in terms of messages exchanged between microengines.

As illustrated in [Figure 22-3](#), the Queue Manager sends enqueue/dequeue transition messages to Scheduler, through a Next Neighbor ring. Scheduler uses transition messages to update queue status, and to generate dequeue requests on that basis. Dequeue requests are sent back to Queue Manager by a scratch ring. As egress Scheduler operates in a packet mode, it may happen that the number of issued dequeue requests exceeds transmission capacity at a moment. To avoid head of the line blocking, Scheduler monitors a number of scheduled but not transmitted packets on each port. The flow control status is obtained from POS transmitter via transfer registers.

Figure 22-3. Interfaces between Scheduler and Collaborating Microengines



22.3.1 Input Microblock Variables

The egress Scheduler microblock receives input from two sources:

- Enqueue/dequeue transition messages from Next Neighbor ring
- Flow control information from transfer registers

Table 22-1 shows the layout of a transition message. The message format is slightly modified, so that `packet_size` is aligned to a byte boundary. Moreover, some original fields are not used and marked as reserved.

Table 22-1. Transition Message Format (NN ring between QM and Scheduler)

LW	Bits	Size	Field	Description
0	31	1	reserved	Not used
	30	1	enqueue_event	If set to 1, one packet has been enqueued.
	29:16	14	reserved	Not used
	15:0	16	enq_queue_id	Absolute queue identifier for enqueue event (queue_id = port_id * QUEUES_PER_PORT + class_id)
1	31	1	reserved	Not used
	30	1	dequeue_event	If set to 1, one packet has been dequeued.
	29:24	6	reserved	Not used
	23:16	8	packet_size	Packet size in 128-byte chunks
	15:0	16	deq_queue_id	Absolute queue identifier for dequeue event (queue_id = port_id * QUEUES_PER_PORT + class_id)

The POS transmit block communicates the number of packets waiting for transmission to the scheduler via a reflector write. The scheduler uses 16 transfer registers (one per port). This interface is not modified.

Table 22-2. Flow Control Message Format (Scheduler xfer registers)

LW	Bits	Size	Field	Description
0	31	1	\$\$txd_p0	Number of packets waiting for transmission on port 0
...
15	31	1	\$\$txd_p15	Number of packets waiting for transmission on port 15

22.3.2 Output Microblock Variables

The egress Scheduler generates only one type of messages: dequeue requests to QM. The message format is not modified, as compared with *Intel® Internet Exchange Architecture Portability Framework Developer's Manual*.

Table 22-3. Dequeue Message Format (scratch ring between Scheduler and QM)

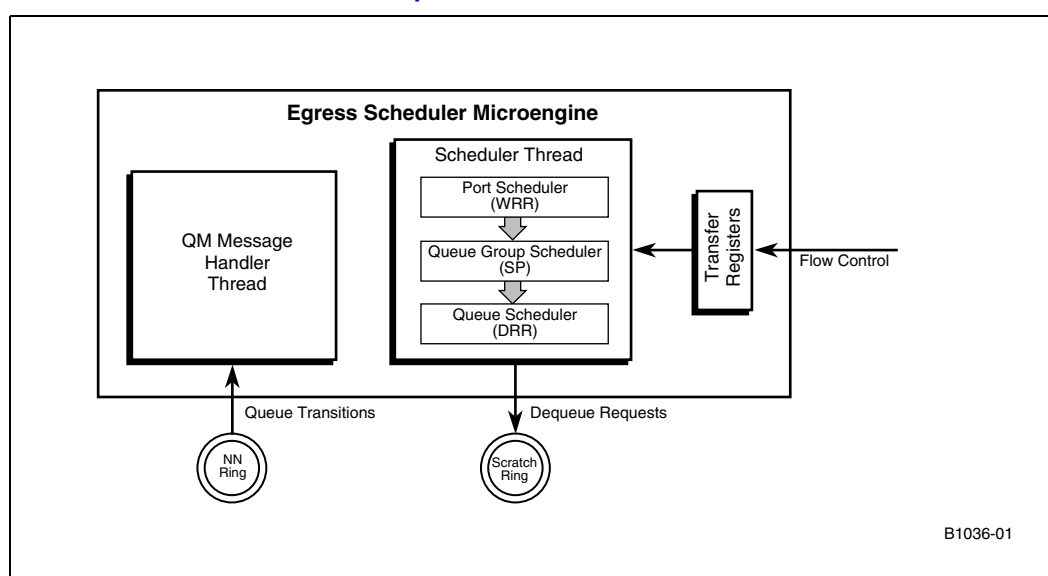
LW	Bits	Size	Field	Description
0	31	1	Valid Bit	Must be 1
0	30:16	15	Reserved	Currently not used
0	0:15	16	Queue ID	Absolute queue identifier for dequeue request (queue_id = port_no * QUEUES_PER_PORT + queue_no)

22.4 Design Decomposition

The Egress Scheduler uses two threads:

- A Queue Manager message handler thread, which processes enqueue/dequeue events
- A Scheduler thread, which selects a queue for transmission and handles flow control

Figure 22-4. Scheduler Microblock Decomposition



22.5 Data Structures

All threads share data structures stored in local memory and global (absolute) registers. Data structures are the same as specified in *Intel® Internet Exchange Architecture Portability Framework Developer's Manual*, including:

- 16 port records in Local Memory, each record occupies 8 long words
- 256 queue records in Local Memory, each record occupies 2 long words
- 2 global GPR register

The extended scheduler design adds new fields to these structures, and rearranges their layout. All the modified parts are indicated with a bolded text in tables below.

Figure 22-5. Common Data Structures, Shared between Two Scheduler Threads

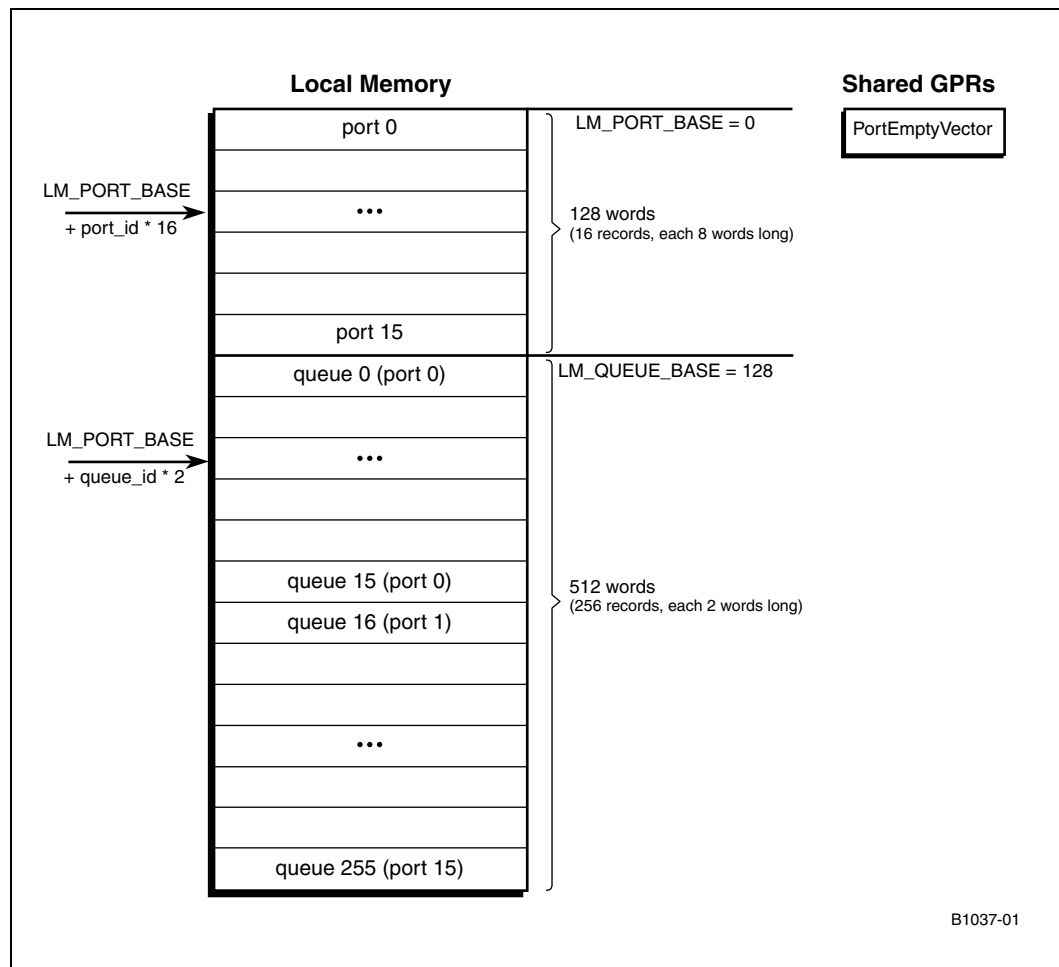


Table 22-4. Port Record

LW	Bits	Size	Field	Description
0	0...15	16	scheduleVector	Bit vector that stores which queues in this port have positive credit. Only a queue with positive credit is schedulable.
1	0...15	16	queueMaskHi	Bit Mask used to round robin through the 8 high priority queues in this port. In each turn, it is shifted left to mask out the currently serviced queue. Once all queues are served (mask is 0), the mask is set to 0x00FF.
2	16...31	16	queueMaskLo	Bit Mask that is used to round robin through the 8 low priority queues in this port. In each turn, it is shifted left to mask out the currently serviced queue. Once all queues are served (mask is 0), the mask is set to 0xFF00.
3	0...15	16	queueEmptyVector	Bit vector that indicates which queues in this port have data.

Table 22-4. Port Record

LW	Bits	Size	Field	Description
4	0...31	32	packetsScheduled	Number of packets scheduled for this port.
5	0...31	32	currentCredit	Current credit for WRR scheduling for this port. The credit is kept as number of packets.
6	0...31	32	weight	The weight (in number of packets) assigned to each port at the beginning of a WRR round
7	0...31	32	Reserved	Aligning data structure to 8 words boundary

Table 22-5. Queue Record

LW	Bits	Size	Field	Description
0	0...15	16	creditIncrement	Credit Increment given to the queue at the beginning of a DRR round (in units of 128 bytes)
0	16...31	16	currentCredit	Current credit for the queue in units of 128 bytes.
1	0...23	24	packetCount	Number of packets stored in a queue.
1	24...31	8	Reserved	Not used.

Table 22-6. Global Register

Register Name	Description
@PortEmptyVector	Bit vector indicating which ports have data.
@PendingRequest	Number of dequeue requests issued to Queue Manager, for which no response is received yet.

In addition, the scheduler thread uses several GPR registers to implement WRR among ports. They are described in [Section 22.6.3, “Scheduler Thread Algorithm”](#) on page 348.

22.6 Flow Chart

22.6.1 Inter-thread synchronization

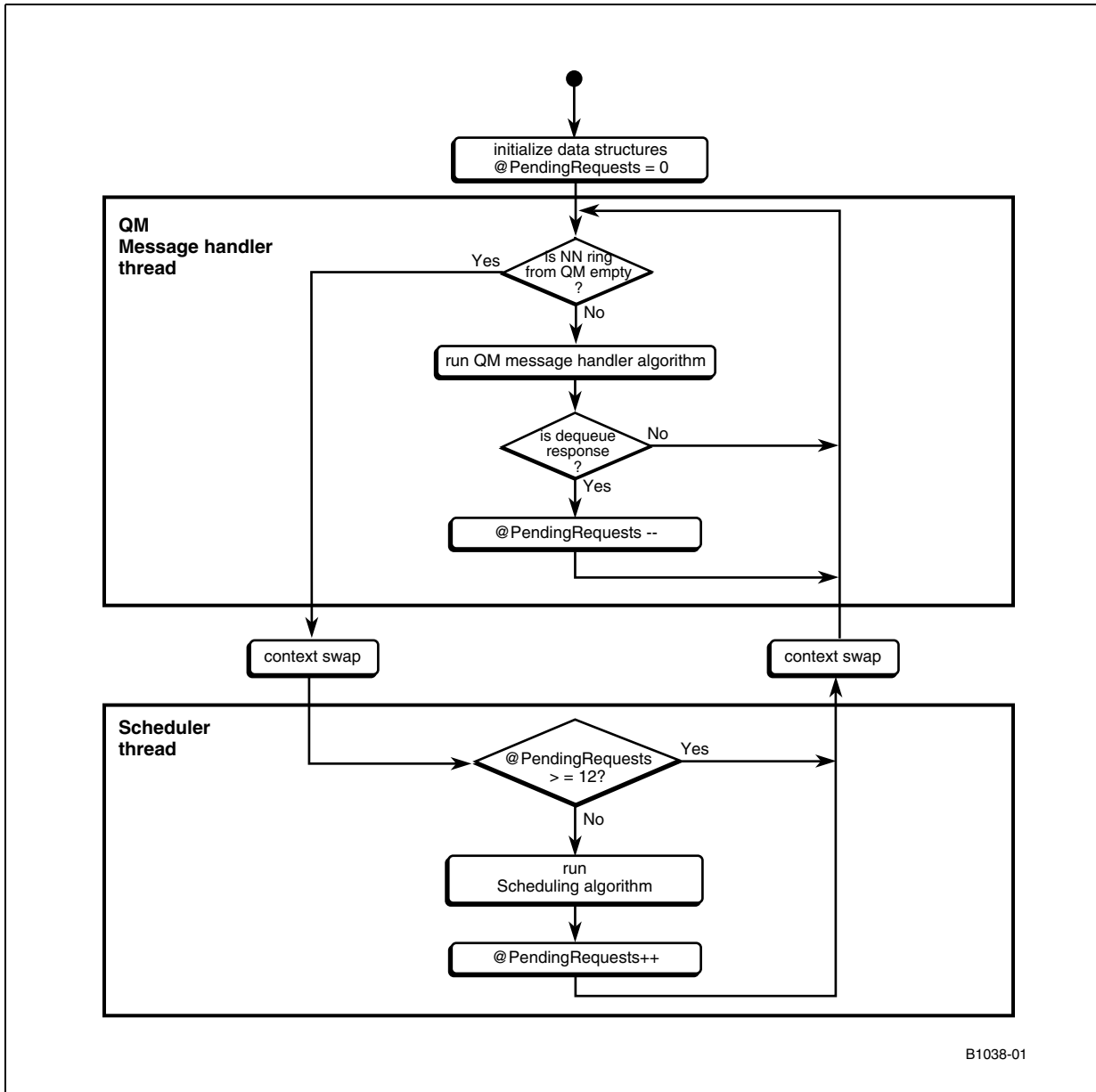
The two operational threads—that is, excluding configuration thread—should be synchronized in a way that ensures low latency between Scheduler and QM microengines. In particular:

- A Queue Manager message handler thread shall empty the NN ring as fast as possible, preferably in an exhaustive manner. Such solution ensures that a Scheduler thread does not operate on stale information.
- A Scheduler thread shall avoid saturating the scratch ring with dequeue requests. Since the Queue Manager microengine employs 8 threads, the number of concurrently processed dequeue messages never exceeds 8. However, when the QM threads are finished with currently processed dequeue messages they immediately attempt to get new messages from the scratch ring. From QM algorithm analysis, it can be observed that the scheduler should allow at minimum 12 pending request to keep QM threads constantly fed up with dequeue messages.

There is no point in posting more messages to scratch ring, as it would increase the inter-ME communication latency.

Figure 22-6 illustrates a synchronization algorithm that fulfills the above goals.

Figure 22-6. Synchronization between scheduler threads



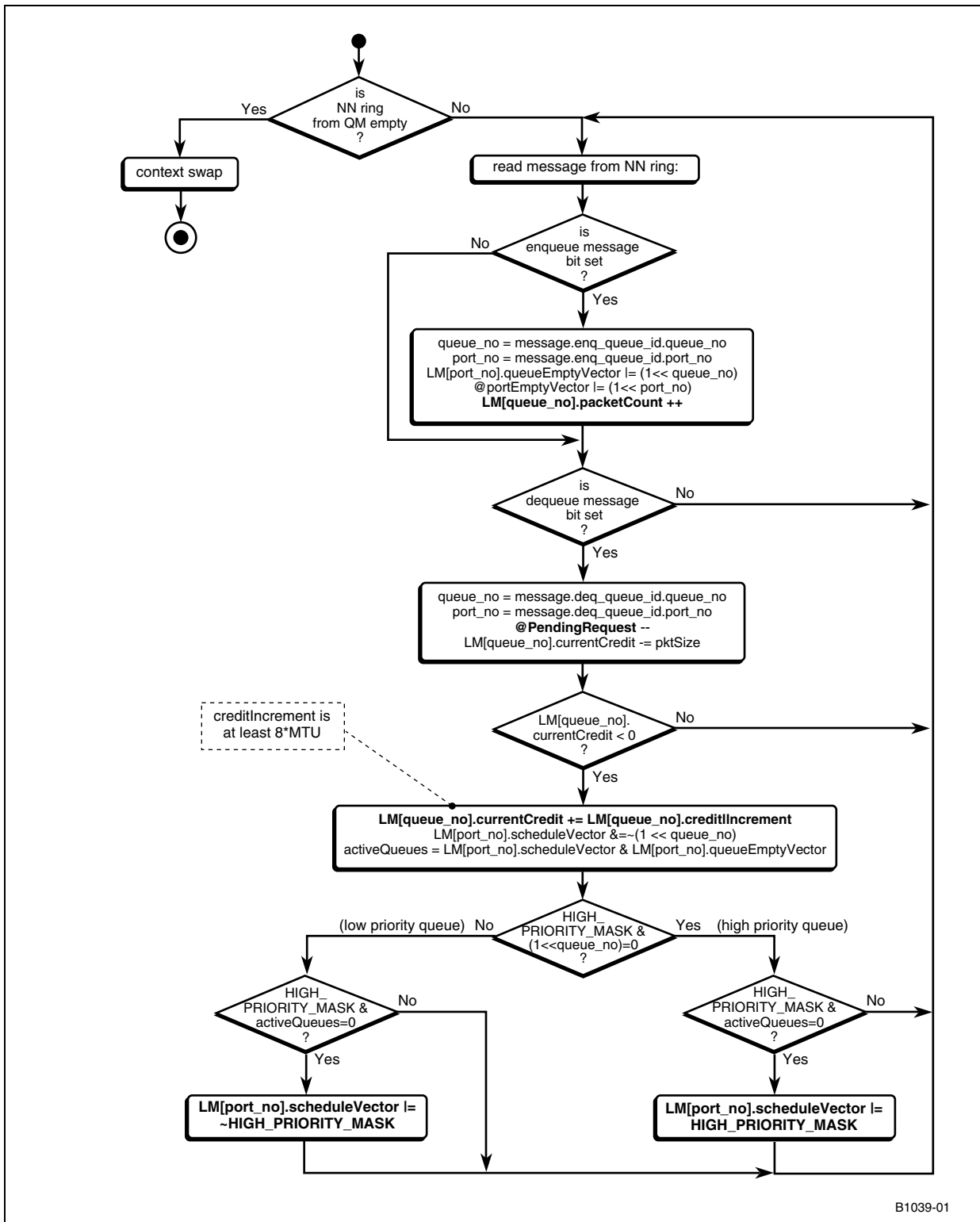
22.6.2 QM Message Handler Thread Algorithm

There are two main differences in comparison to algorithm presented in Chapter 19, “OC-48 WRR/DRR Packet Scheduler.”

- There is no need to handle missed dequeue requests.
- A thread has to track the number of packets in a queue.

Note also that the minimum credit increment is $12 \times \text{MTU}$ size. As a result, at most one-twelfth of packets go through the algorithm path concerning a negative credit condition.

Figure 22-7. QM Message Handler Thread

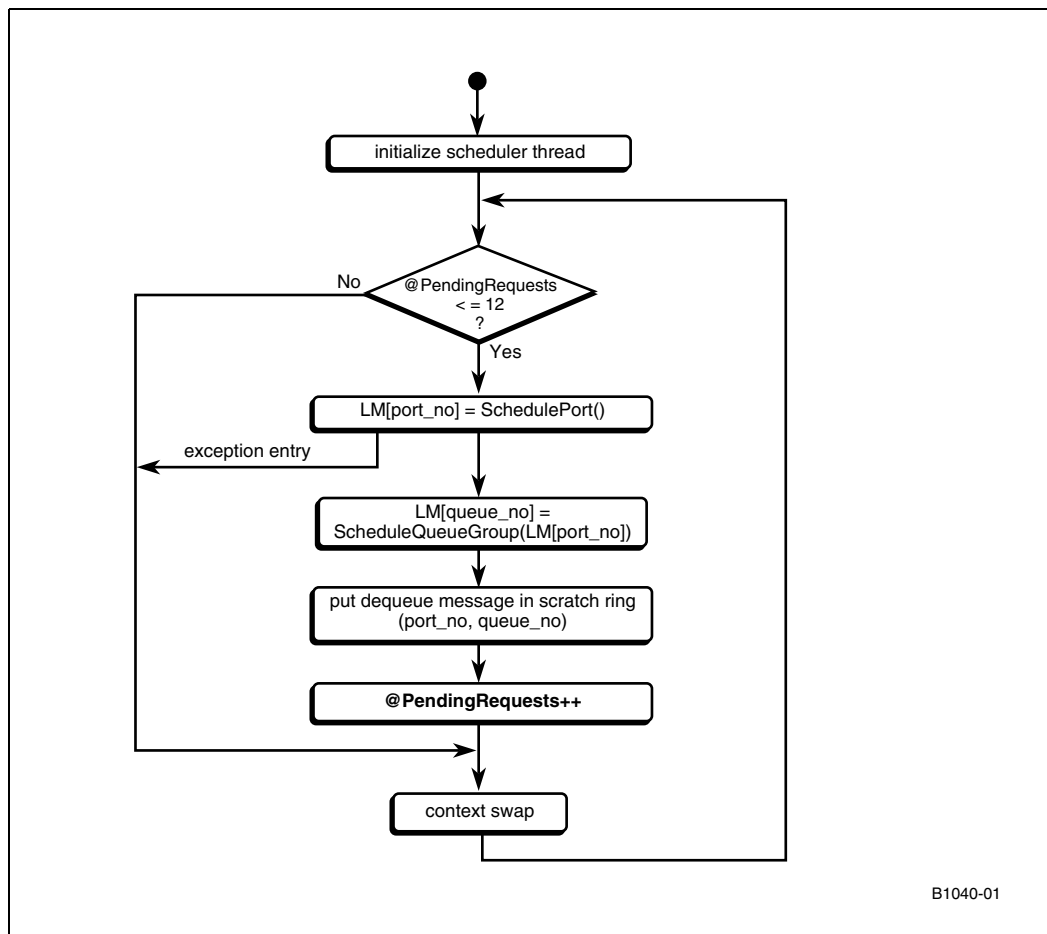


22.6.3 Scheduler Thread Algorithm

the differences in comparison to algorithm presented in the [Chapter 19, “OC-48 WRR/DRR Packet Scheduler”](#) are:

- Scheduler thread keeps track of the number of packets in each queue.
- Scheduler does not generate more than 12 pending requests to the Queue Manager microblock.

Figure 22-8. Scheduler Main Loop



The `SchedulePort()` macro is not changed as compared with the design in [Chapter 19](#).

Figure 22-9. Queue Group Scheduling (Strict Priority)

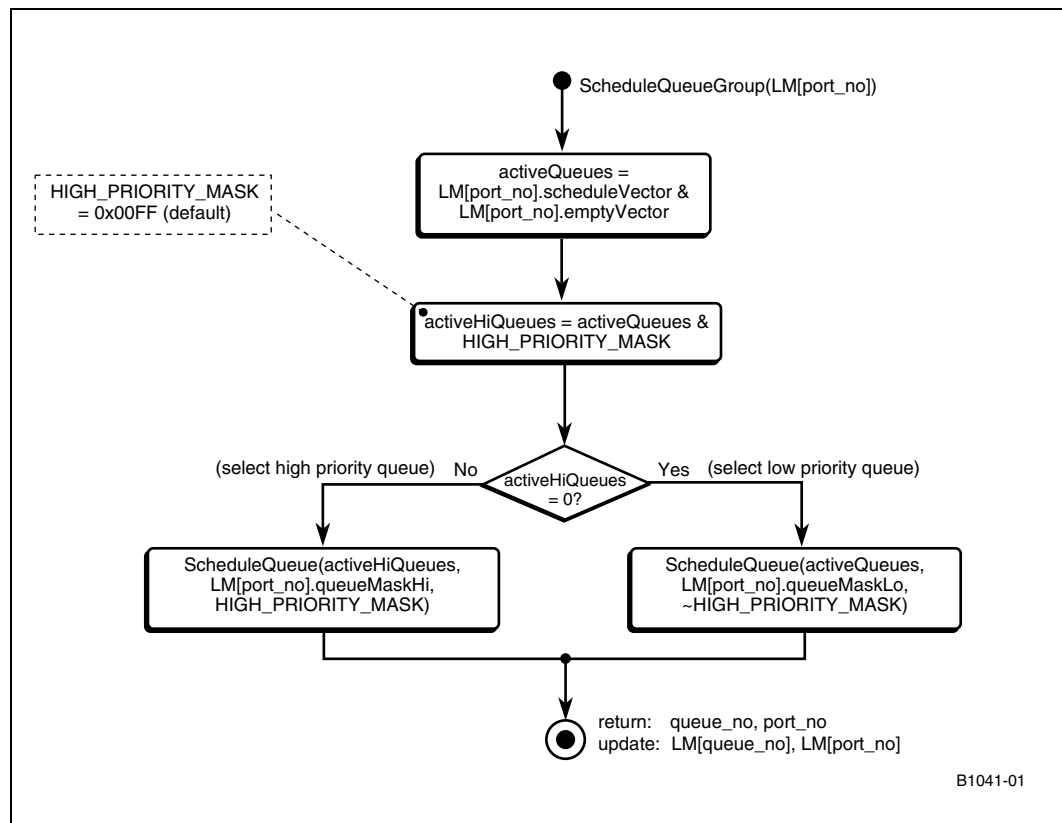
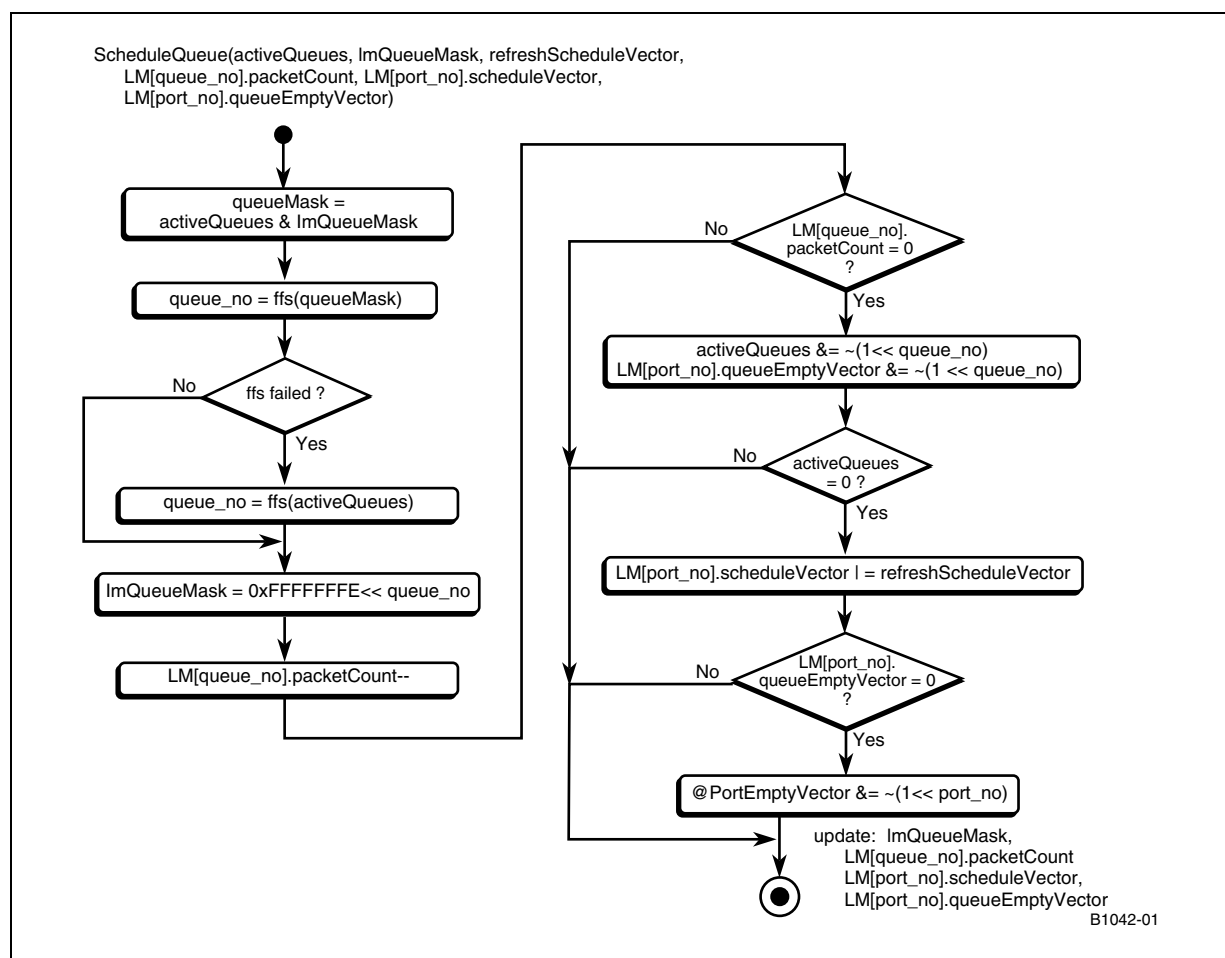


Figure 22-10. Queue Scheduling (Deficit Round Robin)



The algorithm exhibits a kind of self-regulatory characteristics. It runs faster under high load at the expense of increased processing on low load conditions. When queues are backlogged with packets, the decremented packet count remains positive. Conversely, on low load it is more likely that the queue empty condition occurs.

22.7 Micro-code Budget

22.7.1 Performance Analysis

Table 22-7. Cycle Count Table (including unfilled defers)

Phase	Worst case (packet lifetime period)	Worst case (long period) ¹
QM message handler thread	49	30
Scheduler thread	61	59
Total	110	89

1. Long period concerns a scenario when a queue becomes empty at least after two dequeue requests.

Table 22-8. I/O Latency Analysis Table

Operation	I/O type	Accesses	Transfer size (in 32-bit long words)
Put dequeue request in scratch ring	scratch write	1	1

Table 22-9. Memory Accesses Summary Table

I/O type	Accesses
Scratch	1

22.7.2 Memory Footprint Analysis

The Scheduler microblock does not consume messages from any scratch ring. It produces messages to the dequeue request ring.

Table 22-10. Code Store Footprint

Code Store	Size (instruction)
QM message handler thread	405
Scheduler thread	423
Configuration thread	-
Total	828

TM4.1 Shaper and Scheduler Microblock

23

23.1 ATM TM4.1 overview

This section provides an overview of ATM TM 4.1 Traffic Management. Readers familiar with TM 4.1 may skip this section.

23.1.1 What is TM4.1

TM4.1 is a specification by ATM forum for traffic management in ATM networks [TM4.1]. It primarily consists of 8 traffic classes or service categories. The goal of TM4.1 is to ensure efficient and fair utilization of network resources by the individual flows. At the heart of TM4.1 is a simple algorithm called Generic Cell Rate Algorithm (GCRA), described in the following section.

23.1.1.1 GCRA(T , τ)

GCRA is used for policing ATM flows to certain predefined traffic profiles. GCRA has two parameters, T and τ .

- The inverse of T represents the rate allocated to the flow by the network. The rate here could mean either peak rate or average rate depending upon the service class.
- The second parameter τ represents the tolerance around the theoretical arrival times of the cells in a flow.

The goal of GCRA is to ensure that the traffic from the given flow never exceeds the rate allocated and never violates the tolerance limits in the cell arrival times. The actual algorithm follows:

Figure 23-1. GCRA(T, τ) on the Arrival of a New Cell

```
Working variables:
    t = actual cell arrival time
    TAT = theoretical cell arrival time
    t1 = earliest departure time

If (t > TAT)
    t1 = t; TAT = t + T;
Else if (t > TAT -  $\tau$ )
    t1 = t; TAT = TAT + T;
Else /*(t < TAT -  $\tau$ )*/*
    t1 = TAT; TAT = TAT + T;
```

23.1.2 Service Classes in TM4.1

TM4.1 supports the following service classes:

23.1.2.1 CBR

CBR is defined by the following constraints.

- The flow with a given Peak Cell Rate (PCR) and Cell Delay Variation Tolerance (CDVT) should conform to GCRA(LR/PCR, CDVT), where LR is the Link Rate
- The cells of the flow may have a maximum per node delay of Max Cell Transfer Delay (CTD)

23.1.2.2 rtVBR

rtVBR is defined by the following constraints:

- The flow with a given PCR and CDVT should conform to GCRA(LR/PCR, CDVT), where LR is the Link Rate
- The flow with a given SCR and BT should conform to GCRA(LR/SCR, CDVT+BT), where LR is the Link Rate
- The cells of the flow may have a maximum per node delay of Max CTD

23.1.2.3 nrtVBR

nrtVBR is defined by the following constraints:

- The flow with a given PCR and CDVT should conform to GCRA(LR/PCR, CDVT), where LR is the Link Rate
- The flow with a given SCR and BT should conform to GCRA(LR/SCR, CDVT+BT), where LR is the Link Rate

23.1.2.4 UBR (Plain UBR)

UBR is equivalent to the best effort traffic in an IP network and hence is given the lowest priority in the TM space for ATM networks. Hence it does not undergo any GCRA-based shaping for the most part.

23.1.2.5 UBR with PCR (UBR+)

TM4.1 also specifies a UBR traffic class that could optionally have a PCR requirement coupled with CDVT. Such traffic would require shaping and would be defined by the following constraints:

- The flow with a given PCR and CDVT should conform to GCRA(LR/PCR, CDVT), where LR is the Link Rate

23.1.2.6 Differentiated UBR

This is the same as UBR except that one could have several different UBR streams, and the actual UBR stream and the CLP value of the cell would be chosen based on the behavior service class (BSC), which in turn is determined by the Differentiated Services Code Point (DSCP) in the IP header of the frame that is being carried on ATM.

23.1.2.7 GFR

GFR is the newest addition to the TM4.1 suite. It is meant for applications running in the IP world to adapt to ATM networks.

GFR is defined by the following constraints:

- The flow with a given PCR and CDVT should conform to GCRA(LR/PCR, CDVT), where LR is the Link Rate
- The flow with a given MCR and BT should conform to FGCRA(LR/MCR, CDVT+BT), where LR is the Link Rate
- Frame Size < MFS for all the frames of a given flow
- All the cells in the frame should have CLP = 0.

FGCRA(.,.) is very similar to GCRA(.,.) except that variable size frames are considered and the GCRA state is modified proportional to the current frame size.

23.1.2.8 ABR

ABR uses complicated feedback-control mechanisms and is not a consideration in our design due to lack of demand for it.

23.2 Implications of TM4.1

Any cell that passes through the TM4.1 shaping blocks (except for plain UBR) receive a timestamp t_1 —the earliest departure time. This is obtained by the GCRA algorithm.

Note: The architecture does not guarantee that a cell that has a given “earliest departure time” does actually depart at the indicated time.

23.3 Design Overview

The following code values are for various types of traffic:

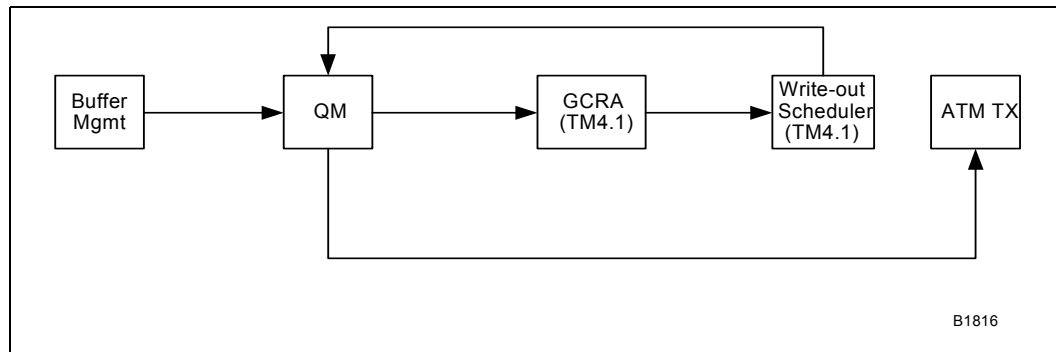
```
CODE = 0x0, for plain UBR,  
CODE = 0x1, for CBR,  
CODE = 0x2, for rt-VBR,  
CODE = 0x3, for nrtVBR VC.  
CODE = 0x4, for UBR+ VC.  
CODE = 0x5, for UBR with MDCR VC.  
CODE = 0x6, for GFR VC.
```

23.3.1 Software Blocks Overview

The design consists of three software blocks:

- TM4.1 Shaper block
- TM4.1 Write-out block
- TM4.1 Scheduler block

Figure 23-2. TM4.1 Architecture overview



The blocks, as shown in [Figure 23-2](#), cooperate with the queue manager (QM) and ATM TX.

The TM4.1 shaper receives input from the queue manager when a cell from a particular Virtual Circuit Queue (VCQ) has been dequeued and there are cells remaining in that VCQ (this is called cell Dequeue without transition) or when cells are enqueued into an empty VCQ (this is called Enqueue with transition). The shaper computes the earliest departure time for the cell using the GCRA traffic descriptors. Then, it computes the time queue ([Section 23.3.2, “Time Queue Data Structure \(TQ\)” on page 357](#)) into which the cell needs to be written based on the earliest departure time and passes the computed time queue#, the VCQ#, and the CODE for the cell on to the next block, the TM4.1 Write-out block.

The Write-out block, based on the CODE value, writes out the cell into

- the real time time-queue if the traffic is CBR or rt-VBR, or
- the non-real time time-queue if the traffic is nrt-VBR, or
- the UBR time time-queue if the traffic is UBR+ or GFR or UBR w/MDCR, or
- one of the UBRwPRI (plain UBR with priority) queues if the traffic is plain UBR.

If no space is available in the requested time queue, the write-out block writes into the next time queue of the same type.

The scheduler block schedules out cells from the time queues and the UBRwPRI queues. It is essentially a priority scheduler with the highest priority for the real-time time-queue, the next priority for non-real time time-queue, then the next priority for UBR time queue, and the lowest priority for UBRwPRI queues.

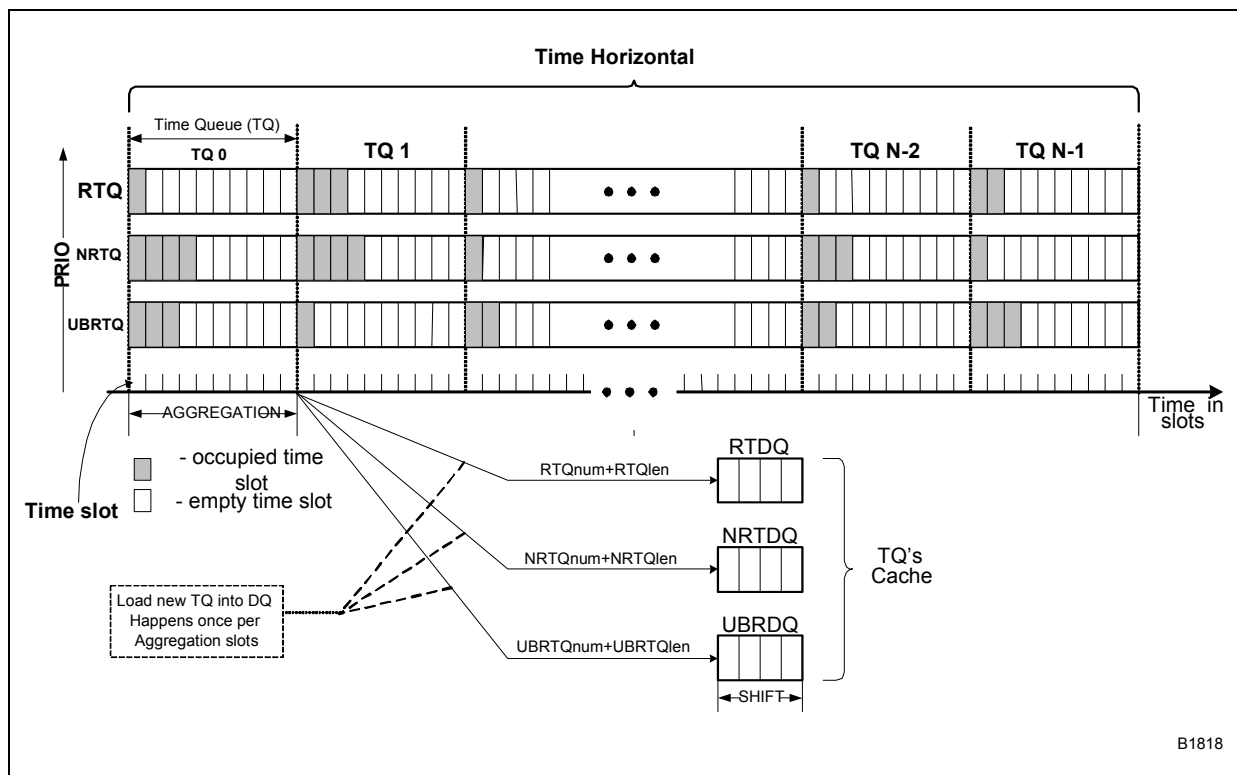
23.3.2 Time Queue Data Structure (TQ)

The design uses time queues to achieve compliance and provide TM4.1 functionality.

The time axis can be divided into small units of cell transmission slots. In each slot, one or no cells depart. A time queue is the aggregation of several cell transmission slots and hence represents an interval of time. The time queue holds the cells that are meant to be transmitted during this time interval.

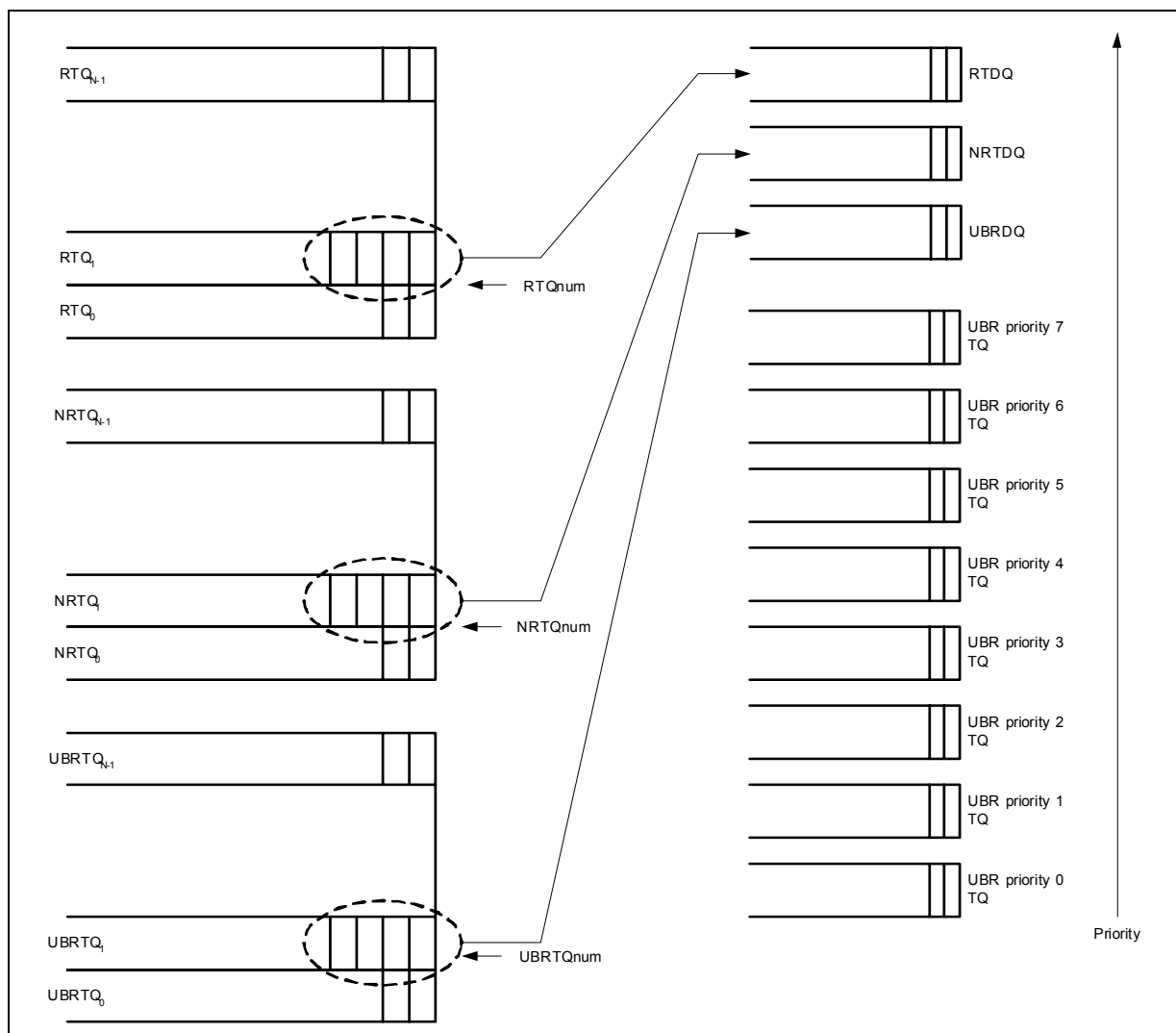
The system has a fixed number of time queues, based on the link rate, the slowest VC bit rate, and the aggregation level of the time queue. The sum of all the time intervals represented by all time queues constitutes the time horizon (TH). The time horizon is nothing but the time after which the time queues wrap around. See [Figure 23-3](#).

Figure 23-3. Time Queues concept



[Figure 23-4](#) illustrates the implementation of the time queues.

Figure 23-4. Time Queue Data structure in SRAM for Low Bit Rate VCs



There are two versions of time queues, one in local memory for high bit rate traffic, and one in SRAM for low bit rate traffic.

23.3.2.1 SRAM Time Queues For Low Bit Rate Traffic

Figure 3 shows SRAM time queues used by the low bit-rate VCs only. There are:

- N real-time time-queues (catering to CBR and rt-VBR) RTQ 0 through N-1,
- N non-real time time-queues (catering to nrt-VBR) NTRQ 0 through N-1,
- N UBR time-queues (catering to UBR+ and GFR) UBRTQ 0 through N-1,
- 8 UBRwPRI queues (catering to plain UBR with priority)

See [Section 23.3.2.1.1, “Number of SRAM Time Queues” on page 359](#) for details about how N is obtained.

Each $RTQ(i) + NRTQ(i) + UBRQ(i)$ has an average size of AGGREGATION cells. In our case, AGGREGATION = 16 (this is programmable). This means that some TQs can have more than AGGREGATION cells and some can have less than AGGREGATION cells. The maximum size of $RTQ(i) + NRTQ(i) + UBRQ(i)$ is $6 * AGGREGATION$.

The actual sizes of $RTQ(i)$, $NRTQ(i)$, and $UBRTQ(i)$ are proportional to the fraction of real-time traffic versus the fraction of non-real-time traffic versus the fraction of UBR+ and GFR traffic.

For example, if 33% of the traffic is real-time, 33% non-real time, and 33% UBR+ and GFR, then the size of $RTQ(i)$ would be $2 * AGGREGATION$ for all “i”, the size of $NRTQ(i)$ would also be $2 * AGGREGATION$ for all “i”, and the size of $UBRTQ(i)$ would also be $2 * AGGREGATION$ for all “i”.

As real time ticks and cells are transmitted, every AGGREGATION cell transmission slots, the cells of the current RTQ move to the real time departure queue RDQ , and the cells of the current $NRTQ$ move to the non-real time departure queue $NRDQ$.

Similarly, the cells of the current $UBRTQ$ move to the UBR departure queue $UBRDQ$. RDQ has a higher priority over $NRDQ$ and $UBRDQ$. Note that cells don't have to be physically moved from the TQ to the DQ - moving the TQ pointer into the DQ is sufficient.

The $RTQnum$ points to the current real-time queue being serviced, while $NRTQnum$ points to the non-real time queue being serviced, and $UBRTQnum$ points to the UBR time queue being serviced currently. These “pointers” may be in sync with the real time or lag the real time. They would lag real time when some of the time queues that have been serviced had more than AGGREGATION cells in them and hence it took more than AGGREGATION time slots to service the time queue.

23.3.2.1.1 Number of SRAM Time Queues

The number of time queues N is a function of the slowest VC rate and the aggregation level of each time queue.

The assumption is that the slowest VC bit rate is 32Kbps = 86 cps (SCR in case of VBR service class and PCR in case of CBR service class) and the aggregation level to be AGGREGATION. These are programmable numbers.

If the slowest VC rate in the system is R_{slow} , then the Time horizon for the system is given by the following:

$$\begin{aligned} TH &= LR / R_{slow} = 5\,636\,096 \text{ cps} / 86 \text{ cps} \\ &= 65536 \text{ cell transmission slots} \\ \text{Here, } LR &= 5\,636\,096 \text{ cps (2.5Gbps), assuming OC-48 rates.} \end{aligned}$$

One transmission slot = 106 cycles.

One timestamp = 16 cycles.

Hence, the value of TH in timestamps is as follows:

$$\begin{aligned} TH &= 65536 * 106 / 16 \\ &= 434176 \text{ MeV2 timestamps, } = 2^{18.7279} = 2^{19} \text{ approx. (1 timestamp = 16 cycles)} \end{aligned}$$

This time horizon is the time after which the time queues wrap around. The value of N is now derived as follows:

$$N = TH/AGGREGATION = 65536 / 16 = 4096 \text{ Time Queues}$$

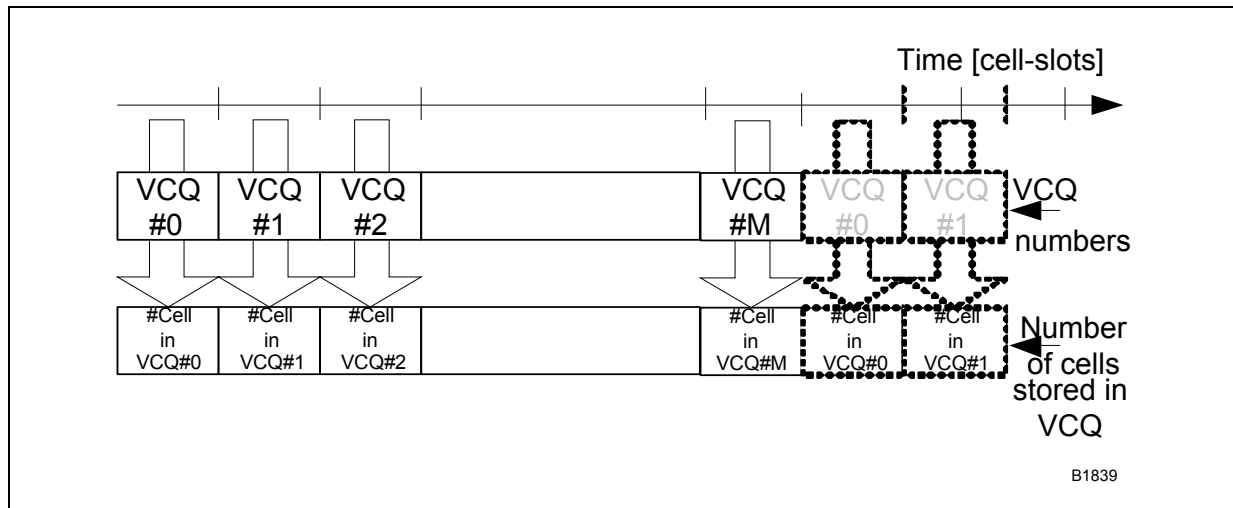
23.3.3 Local Memory Time Queue for High Bit Rate Traffic

Figure 4 shows the time queue in local memory of the Write-out/Scheduler microengine. Only the high bit rate traffic uses this time queue. There are M slots in the time queue. Each of these slots holds two numbers:

- a VCQ to be transmitted
- the number of cells stored in that VCQ

Whenever there are no cells present in the VCQ for the current slot, the SRAM time queue data structure is accessed for cells to be transmitted.

Figure 23-5. Time Queue in Local Memory for Very High Bit Rate VCs



The time queue slots to VCQ mapping is computed in XScale and moved into the local memory whenever a new high bit rate VC connection arrives into or leaves the system.

23.3.3.0.2 Number of slots M

The number of slots M is dependent on the aggregation level of the SRAM time queues. It is also dependent on the latency involved in the feedback to the shaper. The value of M would be the maximum value obtained from both these constraints.

As a feedback, the shaper requires three blocks—the shaper, scheduler, and the queue manager, to be passes. Each block can have a maximum latency of 8 cell slots (due to 8 threads present to hide the latency while effectively meeting the line rate). Hence, the number of slots M is at least 24 from the latency standpoint.

From the aggregation standpoint, a good heuristic would be to assume that M should be (SHIFT+1) times the value of the aggregation level. This is because when a cell is conformant, it is written into a time queue that is SHIFT number of time queues ahead of the current time queue being scheduled (i.e., current time queue + SHIFT) instead of the current time queue - to avoid the complications

that arise out of the same time queue being serviced and written into simultaneously. If the assumption is $AGGREGATION = 16$, $SHIFT = 2$, then the number of slots M is at least 48 from the aggregation standpoint.

Since M has a value of 48. To make it a power of two and have a safety margin, assume $M = 128$. Again, note that M is programmable depending on the way the pipeline is laid out.

23.3.3.1 What exactly are high and low bit rate VCs?

VCs that have inter-departure times of less than $M = 128$ slots (inter-departure time is defined as the inverse of SCR in case of VBR traffic and PCR in case of CBR traffic) need to use the local memory time queue and hence constitute high bit rate traffic. The rest of the VCs are low bit rate VCs

23.3.4 TM4.1 Conformance for Low Bit Rate VCs

TM4.1 conformance can be broken down into two parts for every VC in the system: GCRA conformance and delay conformance. GCRA conformance is more important than the delay conformance since it affects the bandwidth sharing and the fairness of the whole system. Delay conformance typically affects only the given VC in question.

23.3.4.1 GCRA Conformance

GCRA conformance is defined as the cell going out later than the earliest departure time t_1 , obtained in the Shaper block.

A cell that needs to depart at t_1 could depart as early as $(t_1 - AGGREGATION + 1)$ because of aggregating the cell transmission slots into a time queue. This is not counted as a case of conformance violation because there is a finite lower bound to the departure time.

The design provides 100% GCRA conformance under all traffic conditions and profiles, provided time queues do not overflow.

23.3.4.2 Delay conformance

Delay conformance is defined as the cell going out before the latest departure time t_2 . This is relevant only for real-time traffic.

23.3.4.2.1 Heuristic for Delay conformance.

Heuristics are used to provide delay guarantees in case of low bit rate VCs only.

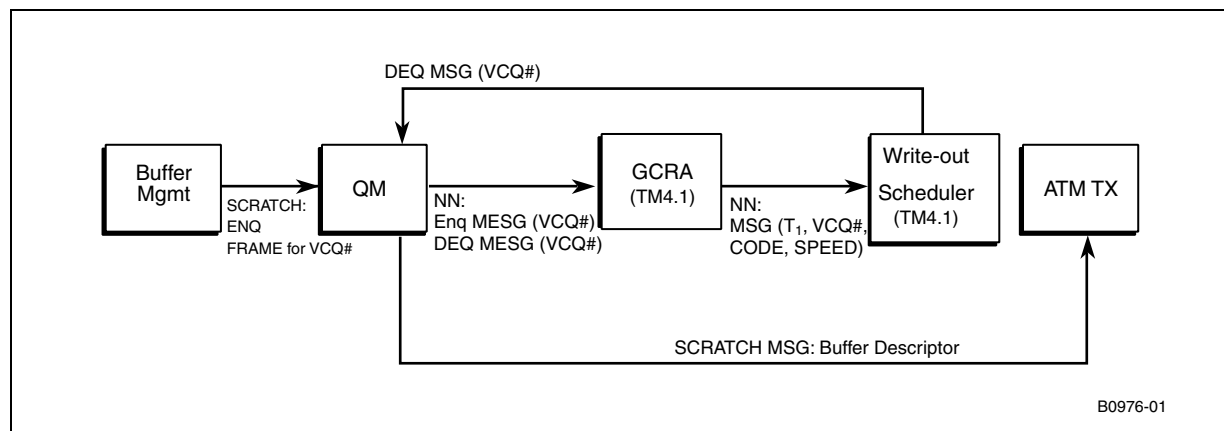
Note: There are solutions that can provide 100% delay conformance, this is not so in this implementation because of a lack of processing cycles on the microengines.

Real-time traffic gets higher priority over non-real-time traffic and the UBR+ and GFR traffic. Hence, there are three sets of time queue data structures—one for real-time traffic, one for the non-real-time traffic, and one for UBR+ and GFR traffic (see [Figure 23-3, “Time Queues concept” on page 357](#)).

23.4 External Interfaces and Communication Data Structures

Figure 23-6 illustrates the external interfaces and communication data structures to TM4.1 blocks.

Figure 23-6. External Interfaces to TM 4.1 Blocks



23.4.1 Interface Between the QM and the GCRA

The GCRA block receives the following message from the Queue Manager through the Next neighbor ring. The message is of 2LWs. The first LW is understood to be the message for which the ENQ happened and the second LW for which the DEQ happened.

- The Valid bit is not used in this design.
- The Transition bit indicates whether the queue to which enqueue or dequeue happened underwent a transition.
- Cell count provides the number of cells in the frame. This is valid only for enqueue message and not valid for the dequeue message.
- The last 19 bits of the LW are the VCQ# to which the enqueue/dequeue happened.

Valid (1)	Transition (1)	reserved (1)	cell_count (11)	SOP(1)	Enq VCQ# (17 bits)
Valid (1)	Transition (1)	reserved (1)	cell_count (11)	SOP(1)	Deq VCQ# (17 bits)

- The Start of Packet (SOP) bit is used for GFR shaping algorithm and it is set by the Queue Manager as follows:
 - “1” for Enqueue message when transition bit is also set, otherwise “0”
 - “1” for Dequeue message when the Queue Manager has transmitted last cell from current packet and there is at another packet in the queue, otherwise “0”

The GCRA block computes the timestamps for the Head of the line cell in the given VCQ# and passes the information onto the write out block.

23.4.2 Interface Between the GCRA and the Write-out/Scheduler

Communication takes place through an NN ring. The GCRA block has the following format to the Write-out block for each cell for the low bit rate VCs (except UBR w/priority VCs).

Table 23-1. GRCA Write-out Block for Each Cell for Low-bit Rate VCs

Reserved (2 bits)	Port (11)	Reserved (2 bits)	VCQ# (17-bits)
Speed (1)	CODE (3)	Resv (9)	t1 (19-bits)

Table 23-2. GRCA Write-out Block for Plain UBR w/priority VCs

Reserved (2 bits)	Port (11)	Reserved (2 bits)	VCQ# (17 bits)
Speed (1)	CODE (3)=0	Resv (14)	PRI (3) cell_count (11)

CODE indicates the traffic service class.

Also, t1 stands for the earliest departure time of the cell and is computed as detailed in the next section and is in the units of Time Queues. Speed = 0.

PRI defines priority of the VC and it can set to a number from the range from 0 to 7, where 7 is the highest priority and 0 is the lowest priority.

- port is the output port number.
- cell_count is the number of cells that were enqueued/dequeued from the VCQ.

Table 23-3 shows GRCA write-out block for the high bit rate VCs, the message is communicated only on enqueues to the writeout.

Table 23-3. GRCA Write-out Block or Each Cell for High-bit Rate VCs

Reserved (2 bits)	Port (11)	Reserved (2 bits)	VCQ# (17 bits)
Speed (1)	Resev (20)		cell_count (11-bits)

Here speed = 1, which means the VC is a high bit rate VC or a UBR VC.

23.4.3 Interface Between the Write-out/Scheduler and the QM

The QM also receives de-queue requests from the TM4.1 Scheduler. The message format is the VCQ#. The dequeued buffer descriptors are sent to ATM TX for transmission onto the MAC layer devices.

Valid (1-bit) "1"	"Reserved" (1)	Port# (11 bits)	Reserved (2-bits)	VCQ# (17-bits)
----------------------	----------------	-----------------	-------------------	----------------

23.5 Design Details

This section describes the implementation of the 3 blocks—the shaper, write-out, and scheduler. The one common feature of all these blocks is that they have strict thread ordering and the threads operate synchronously.

23.5.1 GCRA Shaper

The function of this block is to compute the “earliest transmission time”, t_1 of a cell on a given VC, using GCRA and max delay constraints and pass the information to the write-out block situated in the next ME.

The requests to compute the earliest departure times are provided by the queue manager.

The shaper differentiates between the high and low bit rate VCs by looking at the VCQ#. It is assumed throughout the document that the high bit rate VCQs (which can be only up to a maximum of M) are numbered starting from 1 until a maximum of M.

23.5.1.1 Data Structures

The data structure for a traffic descriptor is as shown below. The data structure fits exactly for rt-VBR VCs. The other service classes use only some fields of the data structure and the unused fields are filled with zeroes. The subscript 1 stands for the first GCRA constraint and 2 stands for the second GCRA constraint.

LW0	TAT1_low (32 bits)			
LW1	TAT1_high (32 bits)			
LW2	TAT2_low (32 bits)			
LW3	TAT2_high (32 bits)			
LW4	Reserved (2-bits)	PRI (3)	Port# (11-bits)	X (16-bits)
LW5	t1 (8 bits)		T1 (24-bits)	
LW6	t2 (24-bits)		CODE (8-bits)	
LW7	Reserved (4-bits)	Line speed (number of bits) (4-bits)		T ₂ (24 bits)
LW8	Correction (21-bits)		TH Mask (11-bits)	

Here T1 and t_1 are parameters for the first GCRA constraint. TAT1 is the theoretical arrival time variable for the first GCRA. T1 is given by LR/PCR and t_1 by CDVT. The corresponding variables for the second GCRA constraint are T2, t_2 and TAT2, where T2 is given by LR/SCR and t_2 by CDVT+BT.

- PRI holds the VC priority - the value is used in the case of plain UBR with priority VCs.
- Port# is used to store output port number the VC is assigned to.
- The X field is used by the FGCRA algorithm as a working variable.

- The Line Speed and TH Mask fields are used to scale down the generic GCRA and FGCRA algorithms designed for OC-48 lines to smaller line rates. The Line Speed field contains information about what number to divide the OC48 line rate by to get the line speed of the link (port) to which the VC belongs. This number is defined as 2^N (where N is a number from 0 to 15), and is stored in this field. The TH Mask field contains number of bits in mask for wrapping Time Horizon for the port to which this particular VC belongs.

All the parameters related to time are in microengine ver2 timestamps (i.e., 16 MeV2 clock cycles).

23.5.1.2 Overview of Operation

The GCRA shaper receives enqueue and dequeue messages from queue manager through NN ring. The message contains the VCQ#, # of cells in frame (invalid for dequeue message), a valid bit, and a transition bit. The shaper first checks if the message is null. If so the message is discarded. The shaper then checks whether the VC is high bit rate or low bit rate by looking at the VCQ#. Based on this information, it decides to shape a cell from the VCQ as described in the sections below.

23.5.1.3 Shaping Decision for Low Bit Rate VCs

The shaper processes only the head of the line cell from the VCQ for the low bit rate VCs. When the scheduler dequeues this cell and when the queue manager lets the shaper know of the dequeue via the dequeue message, the shaper processes another cell from the VCQ (provided that the VCQ has cells present). If no cells are present in the VCQ, the next cell is shaped from this VCQ when a new frame is enqueued into this VCQ and the shaper is notified of it via the QM.

Hence, a decision to shape from the low bit rate VCQ is made only if:

- The message from QM is an enqueue message accompanied by enqueue transition
- The message from QM is a dequeue message accompanied by no transition

However, the GCRA state needs to be updated whenever a cell departs (whenever the message from QM is a dequeue).

23.5.1.4 Shaping Decision for High Bit Rate VCs

For the high bit rate VCQs, the enqueue messages from the QM are passed onto the write-out block. The dequeue messages are discarded. No other processing is done in the shaper.

23.5.1.5 Shaping Macro

Once the decision to shape from the VCQ is made (only for the low bit rate VCQs), the shaper checks for the availability of the VCQ traffic parameters in the local memory of the shaper microengine by looking up the CAM.

If there is a CAM hit, then the traffic parameters are already available in the local memory or being fetched by another thread into the local memory, and a fetch of the parameters from SRAM is not needed. In this case, the thread simply sleeps through until signaled from the previous thread to maintain strict thread ordering and data consistency (in the case that multiple threads are working on the same VCQ).

If there is not a CAM hit, the parameters are fetched from SRAM into the local memory and CAM, in the process overwriting the LRU entry of the CAM.

In the second phase of shaping, the shaper looks at the CODE value as part of the traffic parameter data structure.

- If the VC is CBR or UBR, then single GCRA based on PCR is applied.
- If the VC is VBR, then dual GCRA based on PCR and SCR is applied.
- If the VC is GFR, then FGCRA is applied.
- If the VC is plain UBR, then no shaping is applied. This would yield the “earliest departure time”, t1.

The shaper computes the timequeue# and writes the request into in SRAM as follows:

$$\text{timequeue\#} = t1(\text{slots}) / \text{AGGREGATION}$$

This timequeue#, t1(TQs), along with the VCQ#, CODE, and Speed are communicated to the write-out block using the communication data structure described in [Section 23.4.2, “Interface Between the GCRA and the Write-out/Scheduler” on page 363](#).

GCRA (or FGCRA in the case of GFR) for cell arrival is applied when an enqueue happens with transition and GCRA (or FGCRA) for cell departure followed by cell arrival is applied when dequeue happens without transition. When a dequeue happens with transition, GCRA (or FGCRA) for cell departure is applied. [Section 23.5.1.6](#) details the pseudo code of the operation.

23.5.1.6 Pseudocode

```
//Read the enqueue/dequeue message from QM
Read#:
    Read NN for the enqueue/dequeue messages from QM
    ///////////////////////////////////Enqueue Processing////////////////////////////////////
Enqueue#:
    //Check for Null
    Null_CheckE#:
        If (message == 0) branch Dequeue#
    HBR_LBR_CheckE#:
        If (VCQ# < M), branch HBRE#
        Else
            Branch LBRE#
    HBRE#:
        Write the value of VCQ#, #cells enqueued and SPEED to the next neighbor
        registers of the Write-out/Scheduler block microengine.
        branch Dequeue#
    LBRE#:
        If (transition) Shaper_macro[]

/////////////////////////////////Dequeue Processing////////////////////////////////////
Dequeue#:
    //Check for Null
    Null_CheckD#:
        If (message == 0) Branch Read#
    HBR_LBR_CheckD#:
        If (VCQ# < M), branch HBRD#
        Else
            Branch LBRD#
    HBRD#:
        Branch Read#
    LBRD#:
        Shaper_macro[]
    Branch Read#
```

```

//////////Shaper_Macro[]//////////
Shaper_Macro[]
//Look if the entry is in CAM
CAM#:
    Lookup_CAM(VCQ#)
    If CAM_HIT
        // VC descriptor for the VCQ already in local memory. Just Sleep
        Signal next thread
        Sleep; wake up at VC_Classify#
    If CAM_MISS
        // Read from SRAM and store it in local memory
        Return the LRU value to SRAM
        Issue SRAM read for the VC data structure corresponding to the VCQ
        Signal next thread
        Sleep
        Store the VC descriptor in local memory.
VC_Classify#:
//Check if the VC is CBR or VBR or UBR
    t = instantaneous value of time.
    If (CODE == 0x0) // It's plain UBR VC. Communicate the cell count
        // to write-out
        Set SPEED = 1
        Write the value of VCQ#, cell_count and SPEED to the next
        neighbor registers of the Write-out/Scheduler block microengine.
        Branch Null_loop_UBR#
    If (CODE == 0x1) //It's a CBR VC. Apply single GCRA
        If (dequeue) update GCRA state.
        Apply GCRA1 and obtain t1
        Branch to NNwrite#
    If (CODE == 0x2) //It's a rtVBR VC. Apply Dual GCRA
        If dequeue, update GCRA state
        Apply GCRA1 and obtain t11
        Apply GCRA2 and obtain t12
        t1 = min(t11, t12)
        Branch to NNwrite#
    If (CODE == 0x3) //It's a nrtVBR VC. Apply Dual GCRA
        Apply GCRA1 and obtain t11
        Apply GCRA2 and obtain t12
        t1 = min(t11, t12)
        Branch to NNwrite#
    If (CODE == 0x4) //It's a UBR+ VC. Apply single GCRA
        Apply GCRA1 and obtain t11
        Branch to NNwrite#
    If (CODE == 0x6) //It's a GFR VC. Apply FGCRA
        Apply GCRA and obtain t11
        Apply FGCRA and obtain t12
        t1 = min(t11, t12)
        Branch to NNwrite#
//Write back the computed timestamp to the NN of the Scheduler block
NNwrite#:
    If (!(dequeue with transition))
        Compute the value of t1 in Time Queues (TQs): t1(TQs) =
        t1(tstamp)*16/106/AGGREGGATION.
        Write the value of t1(TQs), VCQ#, CODE and SPEED to the next neighbor
        registers of the Write-out/Scheduler block microengine.
        Signal next thread
        Sleep
Null_loop_UBR#:
    Signal next thread
    Sleep
End_shaper_macro[]

```

23.5.1.7 Divide by 53

A divide by 53 is required in the shaper code to obtain the t1(slot) from the value of t1(timestamp). One timestamp = 16 cycles. One slot = 106 cycles. Hence the calculation is as follows:

$$t1(slot) = t1(timestamp)*16/106 = t1(timestamp)*8/53.$$

The algorithm works in 32-bit space and assumes that $t1$ can have a max value of $2^{19}-1$ as follows:

$$t1(timestamp)/53 = t1(timestamp)*4946/2^{18}$$

23.5.2 Write-out Block

The function of this block is to write the VCQ# of the cell into the Time Queue data structure. The timestamp $t1$ is in Time Queues.

23.5.3 Data Structures

23.5.3.1 Queues for Low Bit Rate VCQ traffic

The key data structure is the Time queue data structure, shown in the figure 3. The low bit rate traffic is written into the real time time-queue, the non-real time time-queue, the UBR time queue, or the UBRwPRI queues belonging to the time queue data structure in SRAM.

The lengths of these time queues are also maintained in SRAM.

With multiple ports, each port has its own time queue data structure.

23.5.3.2 Queues for High Bit Rate VCQ traffic

There is one time queue for the high bit rate traffic. This data structure has the schedules for the high bit rate traffic. This data structure needs to be read from SRAM whenever a high bit rate VC arrives into or leaves from the system.

The number of cells in the VCQ for every high bit rate VC is also maintained in local memory.

23.5.4 Overview of Operation

The write out block obtains 2 LWs using the NN communication from shaper. From this, it checks whether the VC is high bit rate or low bit rate.

For high bit rate traffic, the following information is communicated from shaper:

- Speed
- VCQ
- cell_count
- Port number
- For the low bit rate traffic, the following information is communicated:
 - Speed
 - VCQ
 - CODE
 - Port number
 - t1 in Time Queues

23.5.4.1 Processing for Low bit rate VCs

If the VC is low bit rate, the timequeue# is taken from the message from the shaper block.

If timequeue# is lower than the value of SHIFT added to the current time queue, then $\text{timequeue\#} = \text{SHIFT} + \text{timequeue\#}$.

The implicit assumption for correctness of this algorithm is that the timequeue# used by the write-out would always be at least $\text{current_time_queue} - \text{SHIFT} + 1$. In other words, the communication latency between the shaper and write-out of a conformant cell would only be $(\text{AGGREGATION}) * (\text{SHIFT} - 1)$.

It then reads the length of this timequeue# from SRAM. If this time queue is not full, it computes the SRAM offset and writes the VCQ# of cell into the timequeue#. If not, it increments the timequeue# and repeats the read length operation until it finds a Time Queue where it can add the cell.

23.5.4.2 Processing for high bit rate VCs

The cell count that is received from the shaper for the VCQ# is added to the existing cell count in local memory for that VCQ. No further processing is done.

23.5.5 Write-out block Pseudocode

```

Read Message from Shaper via NN ring
If {Speed = 0} // LBR VC
    Extract Port from the Message
    Load PortInfo(Port) // PortInfo and UBRwPRI - setup LM indexes
    Extract Type of Traffic from the Message
    Extract TQnum from the Message
    If Traffic is UBR
        Increment cell-count for given PRIO
        Set proper bit in UBRwPRI cache (LW7 in PortInfo table)
    Else If Traffic is Real-Time
        Check if RTQnum is not cached in the RTDQ (i.e. currently serviced)
        If Yes RTQnum = CurrTQ+1
            Read length of RTQnum
            //Write request into proper RTQ
            LOOP#:
                Read Len of RTQnum
                If (Len >= MAXRTQLen)
                    TQnum += 1
                    Jump to LOOP#
                Endif
    Else If Traffic is Non-Real Time
        Check if NRTQnum is not cached in the NRTDQ (i.e. currently serviced)
        If Yes NRTQnum = CurrTQ+1
            //Write request into proper NRTQ
            LOOP#:
                Read Len of NRTQnum
                If (Len >= MAXNRTQLen)
                    NRTQnum += 1
                    Jump to LOOP#
                Endif
    Else If traffic is UBRwPCR
        Check if UBRTQnum is not cached in the UBRTDQ (i.e. currently serviced)
        If Yes UBRTQnum = CurrTQ+1
            //Write request into proper UBRTQ
            LOOP#:
                Read Len of UBRTQnum
                If (Len >= MAXUBRTQLen)
                    UBRTQnum += 1
                    Jump to LOOP#
                Endif
    Else //    HBR VC
        Increment cell-count for given VCQ

Endif

```

23.5.6 Scheduler

23.5.6.1 Data Structures

The data structure specific to the scheduler is the Departure queue (apart from, of course, the time queues and UBRwPRI queues per port).

The Departure queue (DQ) is for the time queues in SRAM. The departure queue is used to maintain the SRAM time queues the scheduler needs to service when it lags behind in time (because of the presence of more than AGGREGATION cells in some of the TQs that the scheduler has been servicing).

There are 3 DQs, one for real-time traffic, one for non real-time traffic, and one for UBR+ and GFR traffic. A DQ is implemented in local memory as a software ring containing one long word per entry. Bytes 0 and 1 of this LW are the pointer to the TQ, and bytes 2 and 3 of this LW are the length of the TQ. The DQ is potentially written into every AGGREGATION slots.

Moreover, to operate, the scheduler needs a number of state variables (per port). [Table 23-4](#) shows the definition of variables stored in a structure kept in local memory.

Table 23-4. Definition of Variables Stored in Local Memory

LW	Bits	Length	Name	Notes
LW0	31 to 24	8 bits	RT_for_the_LBR_TQ	The current time slot for the SRAM time queue. This ranges from 0 to AGGREGATION-1 (the aggregation level) and wraps around after AGGREGATION cells are scheduled.
	24 to 8	16 bits	CurTQ	The number of the current TQ that needs to be scheduled from (and is scheduled from if the scheduler is not lagging behind). If the scheduler is lagging in time, a pointer to the curtnum is copied into the software ring, provided the length of the curtnum is non-zero.
	7 to 2	6 bits	Reserved	
	1	1 bit	EntryValid	If set to "1", this PortInfo table entry contains valid information.
	0	1 bit	FC	Indicates whether the port is FC blocked.
LW1	31 to 16	16 bits	RTQnum	Number of the current RTQ being serviced.
	15 to 0	16 bits	NRTQnum	Number of the current NRTQ being serviced.
LW2	31 to 16	16 bits	DQRTlen	Length of the current real time DQ.
	15 to 0	16 bits	DQNRTlen	Length of the current non real time DQ.
LW3	31 to 24	8 bits	RTQlen	Total length of the RTQ being serviced.
	23 to 16	8 bits	NRTQlen	Total length of the NRTQ being serviced.
	15 to 8	8 bits	Working_RTQlen	Number of requests already transmitted from the RTQ being serviced.
	7 to 0	8 bits	Working_NRTQlen	Number of requests already transmitted from the NRTQ being serviced.

Table 23-4. Definition of Variables Stored in Local Memory (Continued)

LW	Bits	Length	Name	Notes
LW4	31 to 24	8 bits	Schcount	Count of the number of cells scheduled out by the scheduler; needed for Flow control Ops.
	23 to 12	12 bits	MaxTQMask	TQ count mask.
	11 to 8	4 bits	Reserved	
	7 to 0	8 bits	Reserved	
LW5	31 to 16	16 bits	UBRTQnum	Number of the current UBRTQ being serviced.
	15 to 0	16 bits	DQUBRTlen	Length of the current UBR+ and GFR DQ.
LW6	31 to 16	16 bits	TQ_offset	Base address for actual TQ (from the beginning of TQ table) - given in elements.
	15 to 8	8bits	UBRTQlen	Total length of the UBRTQ being serviced.
	7 to 0	8 bits	Working_UBRTQlen	
LW7	31 to 28	4 bits	Rtdq_lm_producer	Pointer to an entry in the RTDQ ring where new TQ is added (in LWs).
	27 to 24	4 bits	Rtdq_lm_consumer	Pointer to an entry in the RTDQ ring where is being serviced (in LWs).
	23 to 20	4 bits	Nrtdq_lm_producer	Pointer to an entry in the NRTDQ ring where new TQ is added (in LWs).
	19 to 16	4 bits	Nrtdq_lm_consumer	Pointer to an entry in the NRTDQ ring where is being serviced (in LWs).
	15 to 12	4 bits	Ubrdq_lm_producer	Pointer to an entry in the UBRDQ ring where new TQ is added (in LWs).
	11 to 8	4 bits	Ubrdq_lm_consumer	Pointer to an entry in the UBRDQ ring where is being serviced (in LWs).
	0 to 8	8	UBRwPRICache	A bit set in this byte indicates the presence and location (which queue) of data in the UBRwPRI queue.

Table 23-5. UBR w/priority Table Entry

LW	Bits	Length	Name	Notes
LW0	31 to 15	17 bits	UBRwPRIVcq[0]	VCQ number of a VC with priority 0.
	14 to 0	15 bits	UBRwPRILen[0]	Number of cells stored in this VCQ.
LW1	31 to 15	17 bits	UBRwPRIVcq[1]	VCQ number of a VC with priority 1.
	14 to 0	15 bits	UBRwPRILen[1]	Number of cells stored in this VCQ.
LW2	31 to 15	17 bits	UBRwPRIVcq[2]	VCQ number of a VC with priority 2.
	14 to 0	15 bits	UBRwPRILen[2]	Number of cells stored in this VCQ.
LW3	31 to 15	17 bits	UBRwPRIVcq[3]	VCQ number of a VC with priority 3.
	14 to 0	15 bits	UBRwPRILen[3]	Number of cells stored in this VCQ.
LW4	31 to 15	17 bits	UBRwPRIVcq[4]	VCQ number of a VC with priority 4.
	14 to 0	15 bits	UBRwPRILen[4]	Number of cells stored in this VCQ.

Table 23-5. UBR w/priority Table Entry (Continued)

LW	Bits	Length	Name	Notes
LW5	31 to 15	17 bits	UBRwPRIVcq[5]	VCQ number of a VC with priority 5.
	14 to 0	15 bits	UBRwPRILen[5]	Number of cells stored in this VCQ.
LW6	31 to 15	17 bits	UBRwPRIVcq[6]	VCQ number of a VC with priority 6.
	14 to 0	15 bits	UBRwPRILen[6]	Number of cells stored in this VCQ.
LW7	31 to 15	17 bits	UBRwPRIVcq[7]	VCQ number of a VC with priority 7.
	14 to 0	15 bits	UBRwPRILen[7]	Number of cells stored in this VCQ.

The global variables (across all ports) are:

- **RTLTM** - Pointer to the current time slot for the TQ in local memory. This ranges from 0 to M-1 (the number of entries in the time queue) and wraps around after M slots are scheduled.
- **RTLTM_mask** - Mask to detect wraparounds in the High Bit Rate time queue. This mask has a value of M-1 (M is assumed to be a power of 2).

23.5.7 Overview of Operation

The function of the Scheduler is to schedule cells from TQ-lm, TQ-SRAM, and UBRQ. The scheduling discipline is priority, the priority order being as follows:

- High bit rate VCQ cells from local memory,
- Cells from low bit rate real time TQ in SRAM,
- Cells from low bit rate non real time TQ in SRAM,
- Cells from low bit rate UBR+ TQ in SRAM,
- Cells from UBRQ queues.

23.5.7.1 Operations Not on a per Cell Transmission Slot Basis

23.5.7.1.1 Loading of the low bit rate time queues into respective departure queues in LM

The scheduler is tasked with loading the pointer to and length of the current real time, non real time and UBR+ and GFR time queues into their respective departure queues in local memory. This load happens only if the lengths are non zero, and at a timescale of AGGREGATION timeslots.

23.5.7.1.2 Port Flow control

The scheduler stops scheduling on a port when there is flow control applied on the port. The way the scheduler handles flow control is by reading the count of the number of packets transmitted on the pin from the TX block (the TX block writes this count to scratch and the scheduler reads it from scratch) and comparing this with the count of the number of packets scheduled from the scheduler. If this difference exceeds a particular pre-programmed threshold, then the scheduler applies flow control on the port.

This operation is performed every AGGREGATION cell transmission slots.

23.5.7.1.3 Check for Time gain

The scheduler has to ensure that there is no time gain involved when scheduling from the time queues. Otherwise, the scheduler can schedule cells ahead of its earliest departure time and this would lead to potential conformance problem.

Hence the scheduler needs to periodically compare its time with the real time, and sleep, if it has gained time, until it is in sync with real time.

The scheduler checks for time gain every AGGREGATION number of slots.

23.5.7.1.4 Downloading Time Queue in LM when a HBR VCQ arrives or leaves the system

Since the HBR Time queues are static, they change when a HBR VCQ arrives or leaves the system. This means that the HBR time queue needs to be updated in the scheduler's LM whenever a HBR VC arrives or leaves the system.

The way to do this is via polling. The XScale writes the updated HBR TQs into SRAM whenever a HBR VC arrives into or leaves from the system and sets an indicator bit in SRAM.

The microengine polls every certain number of cell transmission slots for this indicator bit in SRAM. If the bit is set, it downloads the HBR time queue into local memory and resets the indicator bit.

This operation is performed at every programmable multiple of AGGREGATION cell transmission slots. (This is not depicted in the pseudo-code/flow-chart for the sake of simplifying the picture)

23.5.8 Pseudocode

The scheduler assumes a single OC-48 port. Extensions more (up to 2048) ports are described in [Section 23.8, “2048 Ports Hierarchical Port-rate Shaping” on page 384](#) and [Section 23.9, “Up to 8 Ports Hierarchical Port-rate Shaping” on page 389](#).

```

Load first 8 entries of PortShapingTable into PortShapingCache
port_shp_ptr = 0
port_shp_read_ahead = 8
loadnew_tq = 0
rt_hbr = 0

PORT_LOOP#:
  Port = PortShapingCache[port_shp_ptr++]

  If Port is UNUSED
    Jump to PORT_LOOP_CONTINUE#
  EndIf

  // Service Port
  Load PORT PortInfo from SRAM into LM

  If (RT_for_the_LBR_TQ == AGGREGATION)
// Issue read of new RTQ, NRTQ, UBRTQ into DQs
    If ((RTDQ ring is full) || (NRTDQ ring is full) || (UBRDQ ring is full))
      Jump to CONTINUE#
    EndIf

    CurrTQ++
    CurrTQ &= MaxTQMask

    Issue SRAM test_and_clear of the lengths of RTQ[CurrTQ], NRTQ[CurrTQ]
      and UBRTQ[CurrTQ]
    loadnew_tq = 1

  EndIf

```

```

CONTINUE#:
  If HBRVCQlen[rt_hbr] > 0
    Decrement HBRVCQlen[rt_hbr]
    rt_hbr++
    Prepare Dequeue Request to the QM (valid+PORT+VCQ#)
    SCRATCH put the Request
    Jump to LOAD_NEW_TQ#
  EndIf

RT_for_the_LBR_TQ++

If (DQRTlen > 0) // Schedule from Real-Time Queue
  If (RTQlen == Working_RTQlen)
// Current RTQ is done, load new one from RTDQ
    RTQnum = RTDQ[Rtdq_lm_consumer]
    RTQlen = RTDQ[Rtdq_lm_consumer++]
    Working_RTQlen = 0
  EndIf

  TimeSlotAddr = RTQbase[RTQnum] + Working_RTQlen
  Issue VCQ = SRAM[read, TimeSlotAddr]
  DQRTlen --
  Working_RTQlen++
// Optimization: WriteOut part should be done here, since we are waiting for SRAM
  Jump to DEQ#
EndIf

If (DQNRTlen > 0) // Schedule from Non Real-Time Queue
  If (NRTQlen == Working_NRTQlen)
// Current NRTQ is done, load new one from NRTDQ
    NRTQnum = NRTDQ[NRtdq_lm_consumer]
    NRTQlen = NRTDQ[NRtdq_lm_consumer++]
    Working_NRTQlen = 0
  EndIf

  TimeSlotAddr = NRTQbase[NRTQnum] + Working_NRTQlen
  Issue VCQ = SRAM[read, TimeSlotAddr]
  DQNRTlen --
  Working_NRTQlen++

// Optimization: WriteOut part should be done here, since we are waiting for SRAM
  Jump to DEQ#
EndIf

If (DQUBRTlen > 0) // Schedule from UBR Time Queue
  If (UBRTQlen == Working_UBRTQlen)
// Current UBRTQ is done, load new one from UBRTDQ
    UBRTQnum = UBRTDQ[UBRtdq_lm_consumer]
    UBRTQlen = UBRTDQ[UBRtdq_lm_consumer++]
    Working_UBRTQlen = 0
  EndIf

  TimeSlotAddr = UBRTQbase[UBRTQnum] + Working_UBRTQlen
  Issue VCQ = SRAM[read, TimeSlotAddr]
  DQUBRTlen --
  Working_UBRTQlen++
// Optimization: WriteOut part should be done here, since we are waiting for SRAM
  Jump to DEQ#
EndIf
Dequeue from UBR_PRI TQ
// Optimization: WriteOut part should be done here, since we are waiting for SRAM

```

```

DEQ#:
    Prepare Dequeue Request to the QM (valid+PORT+VCQ#)
    SCRATCH put the Request

LOAD_NEW_TQ#:
    If loadnew_tq!= 0
        RTDQ[Rtdq_lm_producer++] = CurrTQ + RTQlen
        NRTDQ[NRtdq_lm_producer++] = CurrTQ + NRTQlen
        UBRTDQ[UBRtdq_lm_producer++] = CurrTQ + UBRTQlen
    EndIf

PORT_LOOP_CONTINUE#:
    If (port_shp_ptr%4!= 0)
        Jump to PORT_LOOP#                // Service next port
    EndIf
    If (port_shp_ptr%8 == 4)
        //Issue read next 8 elements of PortShaping Table:
        SRAM[read, PortShapingTable + port_shp_read_ahead]
        port_shp_read_ahead += 8
        port_shp_read_ahead &= max_port_mask
        Jump to PORT_LOOP#
    Else// Write read entries
        Write new PortShaping Entries from XFER regs into LM
        Jump to CHECK_TIME_GAIN#
    EndIf

CHECK_TIME_GAIN#:
    If Scheduler gained time
        Sleep until it sync up with real time
    EndIf
    Jump to PORT_LOOP#

```

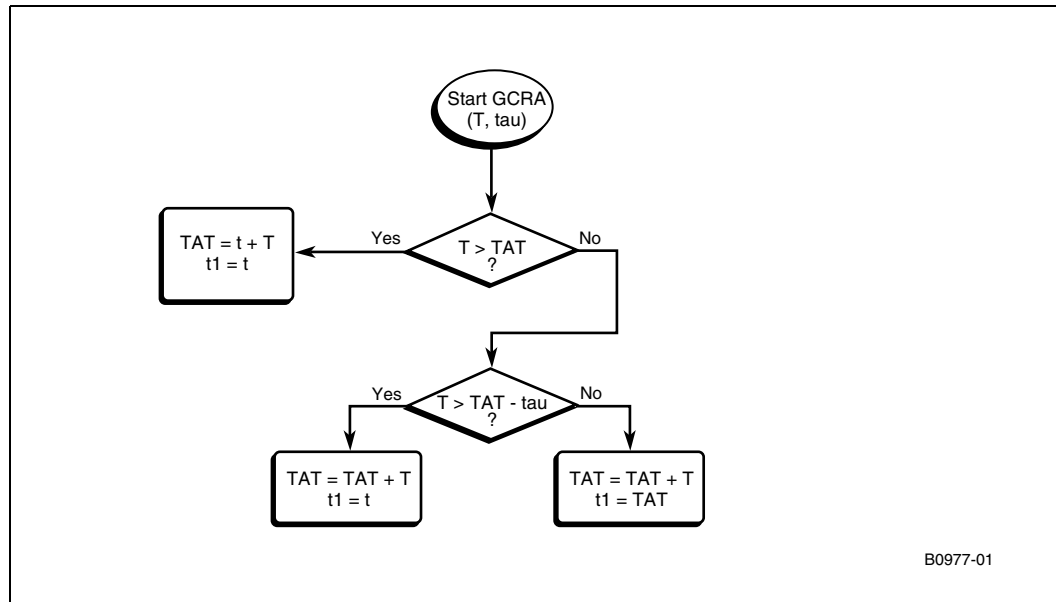
23.6 Flow Charts for Blocks

The following flow charts are included:

- [Figure 23-7, “GCRA Flow Chart” on page 378](#)
- [Figure 23-8, “F-GCRA Shaping Macro Flow Chart” on page 379](#)
- [Figure 23-9, “Shaper \(Macro only\) Flow Chart” on page 380](#)
- [Figure 23-10, “Write-out Block Flow Chart” on page 381](#)
- [Figure 23-11, “Scheduler Flow Chart” on page 382](#)
- [Figure 23-12, “Dequeue Flow Chart” on page 383](#)

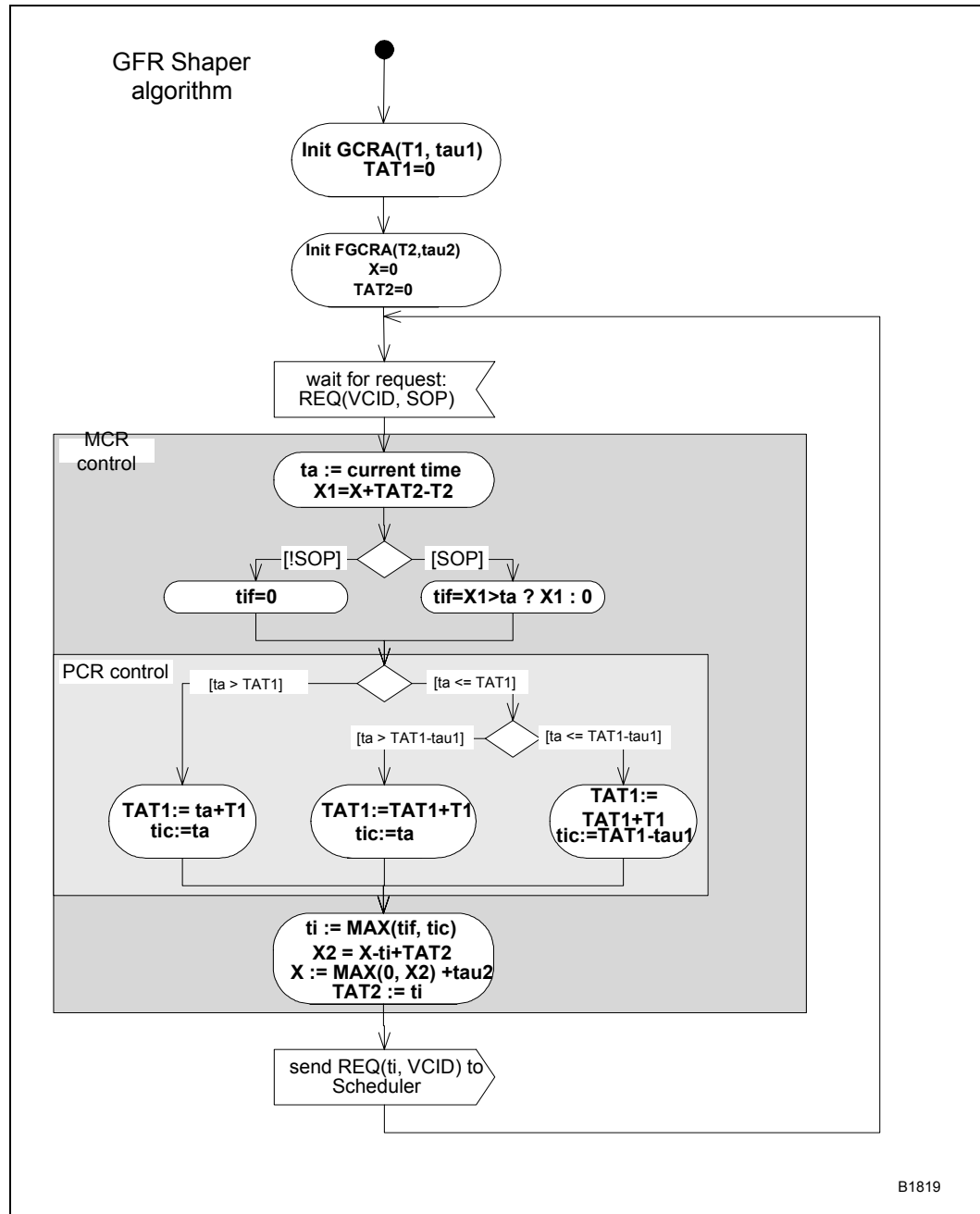
23.6.1 GCRA Flow Chart

Figure 23-7. GCRA Flow Chart



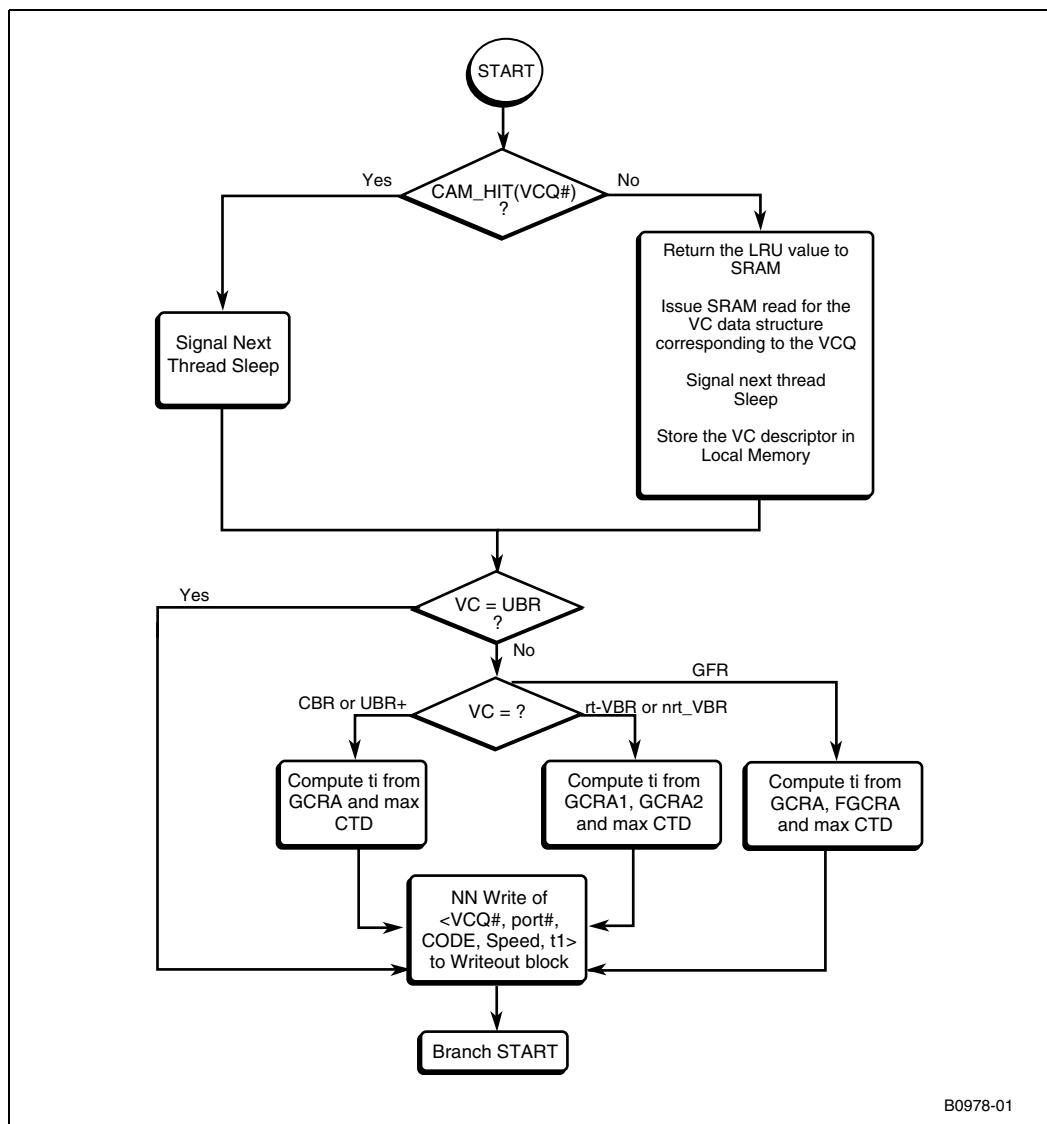
23.6.2 F-GCRA Shaping Macro Flow Chart

Figure 23-8. F-GCRA Shaping Macro Flow Chart



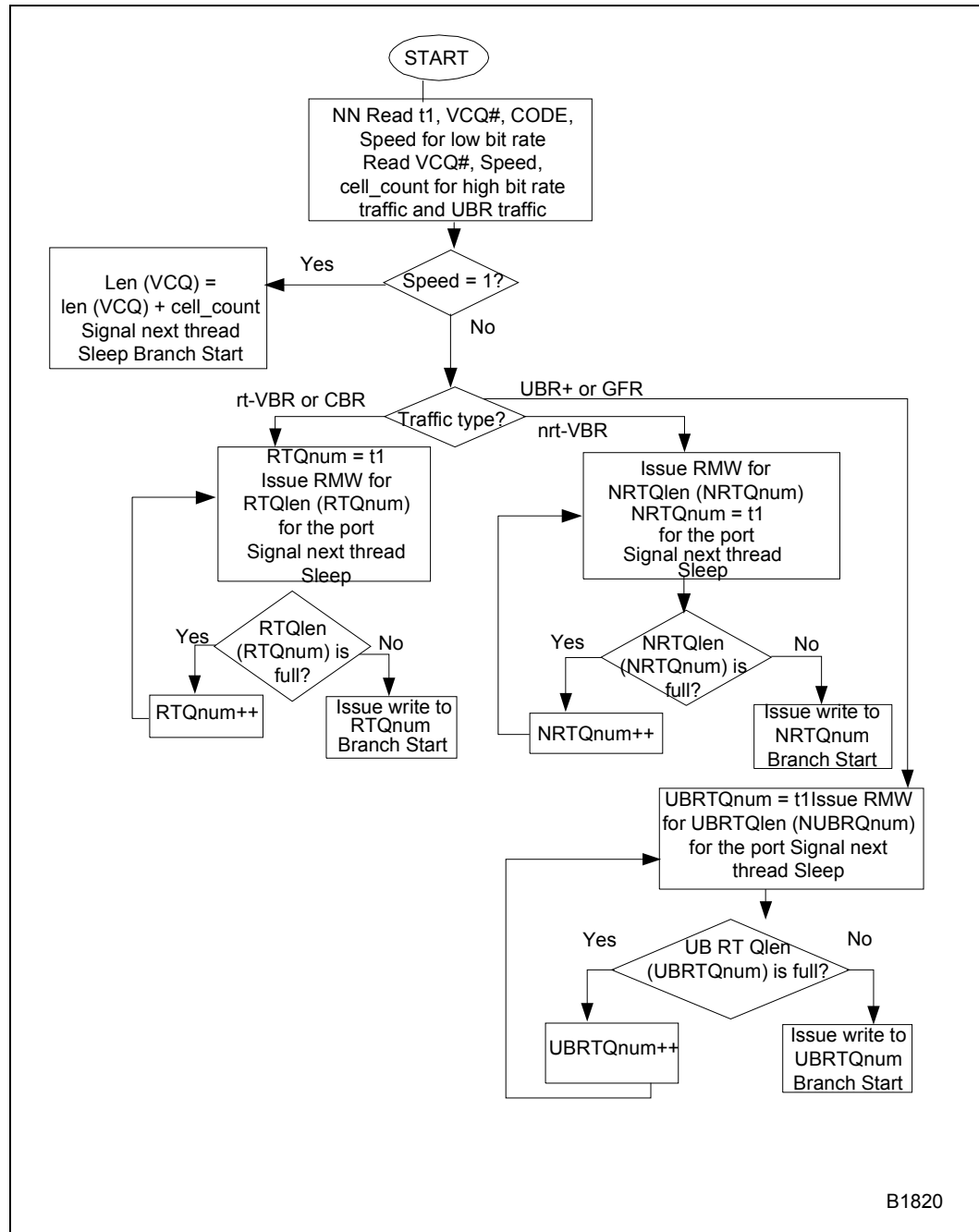
23.6.3 Shaper (Macro only) Flow Chart

Figure 23-9. Shaper (Macro only) Flow Chart



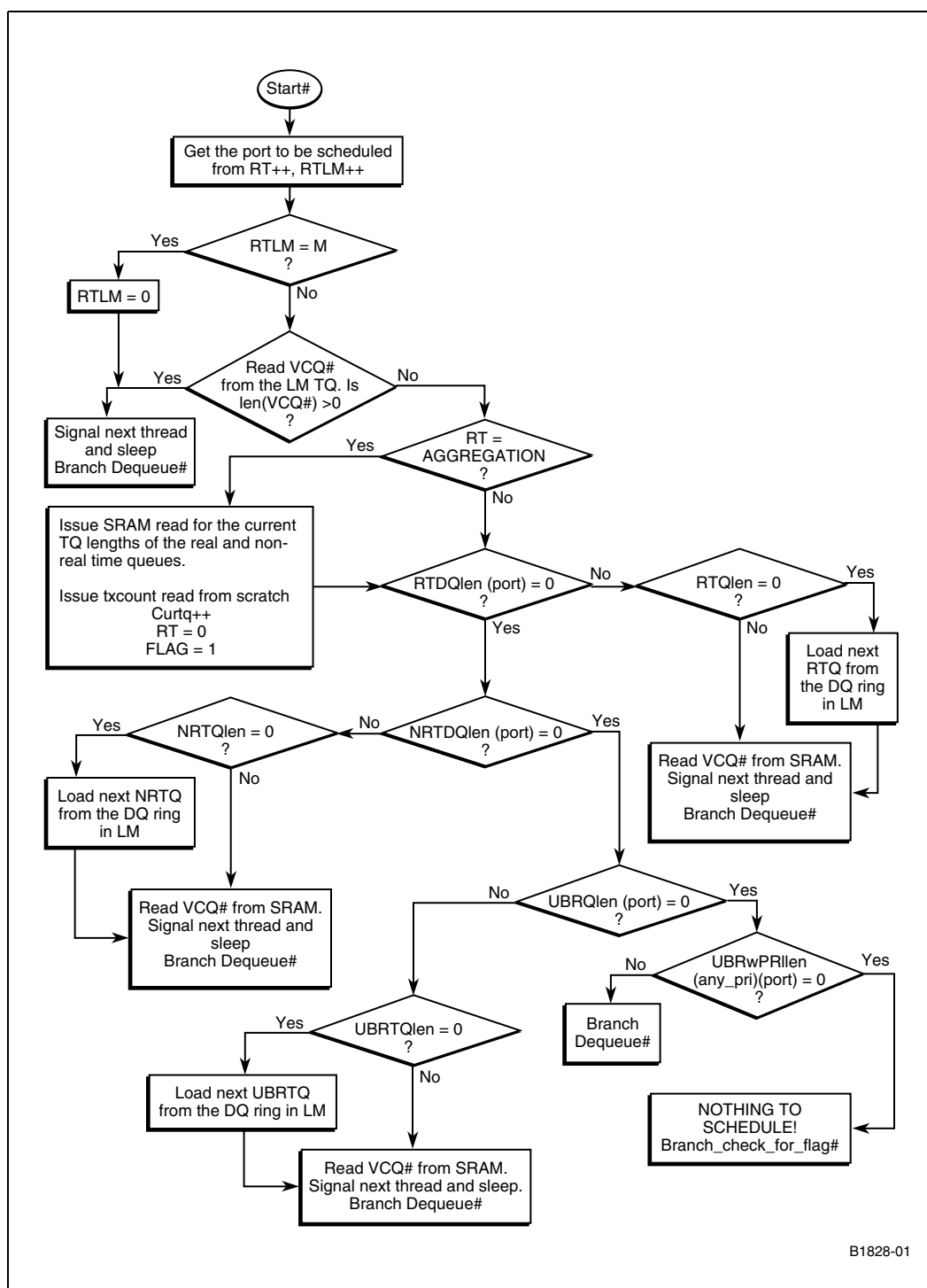
23.6.4 Write-out Block Flow Chart

Figure 23-10. Write-out Block Flow Chart



23.6.5 Scheduler Flow Chart

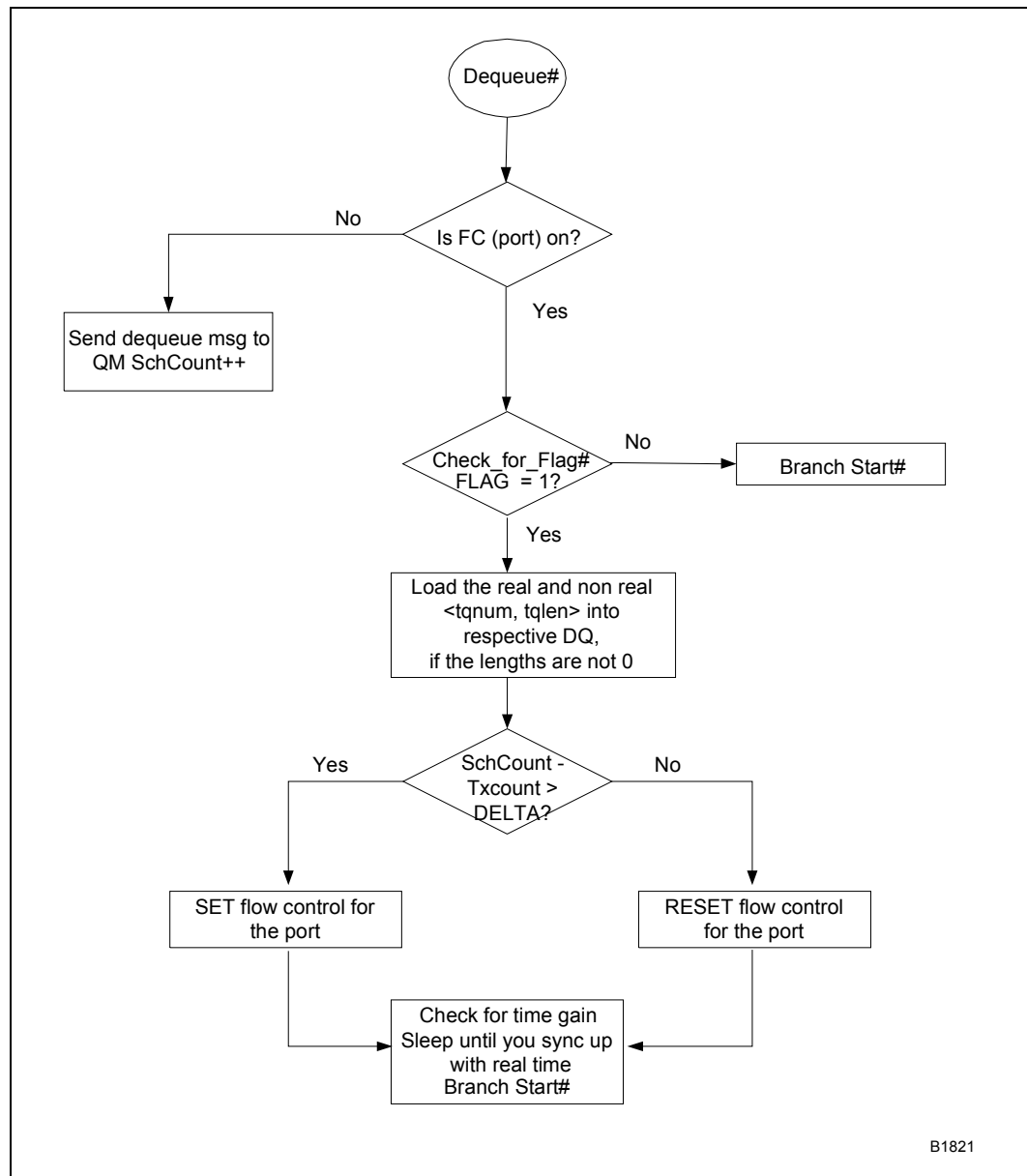
Figure 23-11. Scheduler Flow Chart



B1828-01

23.6.6 Dequeue Flow Chart

Figure 23-12. Dequeue Flow Chart



23.7 Performance Analysis

To process a cell at OC-48 rates on IXP2400, there are 106 cycles. Hence, the following is ensured:

- the TM4.1 blocks take up fewer than 106 cycles each
- the sum of all the read latencies is less than 8×106 cycles.:

Table 23-6. Instruction Counts for the Blocks

	Instruction count	#SRAM reads
Shaper	101	1
Writeout, Scheduler	106	2 ¹

1. Concurrency possible between the reads

Thus the thread issues the time queue read in scheduler and jumps to write out and issues time queue length read for the write-out. Then it sleeps on both the reads. After wakeup, it completes the write-out operations and jumps back to scheduler.

This affects the modularity of the write-out and scheduler as individual blocks, but again, it is not expected that these blocks are modified by a third party to a great extent.

23.8 2048 Ports Hierarchical Port-rate Shaping

This section discusses hierarchical port rate shaping for up to 2048, where each of these 2048 ports needs to be shaped to the individual pre-configured CBR port rates. This is to ensure that the buffers on the ports do not overflow, assuming, in the worst case, there may just a single cell buffer per port.

The idea is to maintain separate sets of time queue data structures and plain `UBRWPRI` queues in SRAM for each port. This does not increase memory size requirements since the memory requirements are proportional to the port rate, which is proportionately smaller in the case of 2048 ports.

23.8.1 Data structures

23.8.1.1 Port State

Each port has a corresponding port state. The port state is stored in data structure described in [Section 23.5.6.1, “Data Structures” on page 371](#). All 2048 port state data structures are stored in SRAM in the PortInfo Table and are read from SRAM when needed. To avoid unnecessary reads and writes a typical caching mechanism using CAM and local memory has been used.

23.8.1.2 Local Memory Map

To improve performance of the Scheduler, some data is kept in local memory. Table shows the local memory map.

Table 23-7. Local Memory Map

LW indexFrom	LW indexTo	Length	Name	Notes
0x000	0x007	8LW	PortInfo for CAM 1	8-long-word-long port context for a port cached in CAM on position 1.
0x008	0x00F	8LW	UBRwPRI for CAM 1	Information about 8 UBR w/priority VCs assigned to a port cached in CAM on position 1. Each VC stores—17-bit-long VCQ# (bits 31 to 15) and 15-bit-long cell-counter (bits 14 to 0).
0x010	0x017	8LW	PortInfo for CAM 2	
0x018	0x01F	8LW	UBRwPRI for CAM 2	
0x020	0x027	8LW	PortInfo for CAM 3	
0x028	0x02F	8LW	UBRwPRI for CAM 3	
0x030	0x037	8LW	PortInfo for CAM 4	
0x038	0x03F	8LW	UBRwPRI for CAM 4	
0x040	0x047	8LW	PortInfo for CAM 5	
0x048	0x04F	8LW	UBRwPRI for CAM 5	
0x050	0x057	8LW	PortInfo for CAM 6	
0x058	0x05F	8LW	UBRwPRI for CAM 6	
0x060	0x067	8LW	PortInfo for CAM 7	
0x068	0x06F	8LW	UBRwPRI for CAM 7	
0x070	0x077	8LW	PortInfo for CAM 8	
0x078	0x07F	8LW	UBRwPRI for CAM 8	
0x080	0x087	8LW	PortInfo for CAM 9	
0x088	0x08F	8LW	UBRwPRI for CAM 9	
0x090	0x097	8LW	PortInfo for CAM 10	
0x098	0x09F	8LW	UBRwPRI for CAM 10	
0x0A0	0x0A7	8LW	PortInfo for CAM 11	
0x0A8	0x0AF	8LW	UBRwPRI for CAM 11	
0x0B0	0x0B7	8LW	PortInfo for CAM 12	
0x0B8	0x0BF	8LW	UBRwPRI for CAM 12	
0x0C0	0x0C7	8LW	PortInfo for CAM 13	
0x0C8	0x0CF	8LW	UBRwPRI for CAM 13	
0x0D0	0x0D7	8LW	PortInfo for CAM 14	
0x0D8	0x0DF	8LW	UBRwPRI for CAM 14	
0x0E0	0x0E7	8LW	PortInfo for CAM 15	
0x0E8	0x0EF	8LW	UBRwPRI for CAM 15	

Table 23-7. Local Memory Map (Continued)

LW indexFrom	LW indexTo	Length	Name	Notes
0x0F0	0x0F7	8LW	PortInfo for CAM 16	
0x0F8	0x0FF	8LW	UBRwPRI for CAM 16	
0x100	0x10F	16LW	PortShaping cache	16 entries from the PortShaping table.
0x110	0x11F	16LW	DQs for CAM 1	RT, NRT, and UBR Departure Queues cached for the port cached in CAM on position 1.
0x120	0x12F	16LW	DQs for CAM 2	
0x130	0x13F	16LW	DQs for CAM 3	
0x140	0x14F	16LW	DQs for CAM 4	
0x150	0x15F	16LW	DQs for CAM 5	
0x160	0x16F	16LW	DQs for CAM 6	
0x170	0x17F	16LW	DQs for CAM 7	
0x180	0x18F	16LW	DQs for CAM 8	
0x190	0x19F	16LW	DQs for CAM 9	
0x1A0	0x1AF	16LW	DQs for CAM 10	
0x1B0	0x1BF	16LW	DQs for CAM 11	
0x1C0	0x1CF	16LW	DQs for CAM 12	
0x1D0	0x1DF	16LW	DQs for CAM 13	
0x1E0	0x1EF	16LW	DQs for CAM 14	
0x1F0	0x1FF	16LW	DQs for CAM 15	
0x200	0x20F	16LW	DQs for CAM 16	
0x210	0x22F	32LW	128 entries of HBR TQ	Each entry is 8 bits long and contains HBR VCQ# (from 0 to 127).
0x230	0x26F	64LW	128 structures for HBR VCs	Each long word contains two 16-bit-long cell counters.
0x270	0x27F	16LW	Other variables needed by the HBR TQ	

23.8.1.3 Static Port Schedule

The write-out/scheduler maintains a static schedule of the various ports depending upon their configured CBR rates. This schedule is maintained in SRAM as a Port Shaping Table, and a window's worth is read into local memory to avoid the SRAM latency every time for scheduling the cell. It is called a static schedule since it is not change often after being set up during initialization, since the number of ports in the system is not expected to change often.

This is a schedule of 2048 entries processed circularly: it wraps around after 2048 entries are completed. The working index, pointing at the Port Shaping Table entry, is also used to check the HBR queue entry status (after a modulo 128 operation). If the corresponding HBR slot is not empty and maps to a VC, it is processed in a 'fast path'. Otherwise, an LBR VC is serviced, so a port index is retrieved from the Port Info table to access all other port-related structures. To speed up the operation, the Port Shaping table is buffered in local memory (a sliding window of 16 entries).

One Port Shaping Table entry corresponds to the smallest bandwidth unit and contains the 16-bit port number to which the unit is allocated. A port may be assigned one or more bandwidth units provided the sum of all units is not greater than total link capacity. In this way, it is possible to divide the total available bandwidth among ports either equally or in desired proportions.

Figure 23-13 shows a sample port shaping table.

Figure 23-13. Sample Port Shaping Table

<i>Port Shaping Table</i>	
port #1	
port #2	
port #3	
port #4	
port #5	
port #6	
port #7	
port #1	
port #8	
port #9	
port #10	
port #11	
port #12	
port #13	
port #14	
port #1	
port #15	
port #16	
port #17	
port #18	
port #19	
port #20	
port #1	
port #8	
port #21	
port #22	
port #23	
port #24	
port #25	
port #26	
port #27	
port #28	

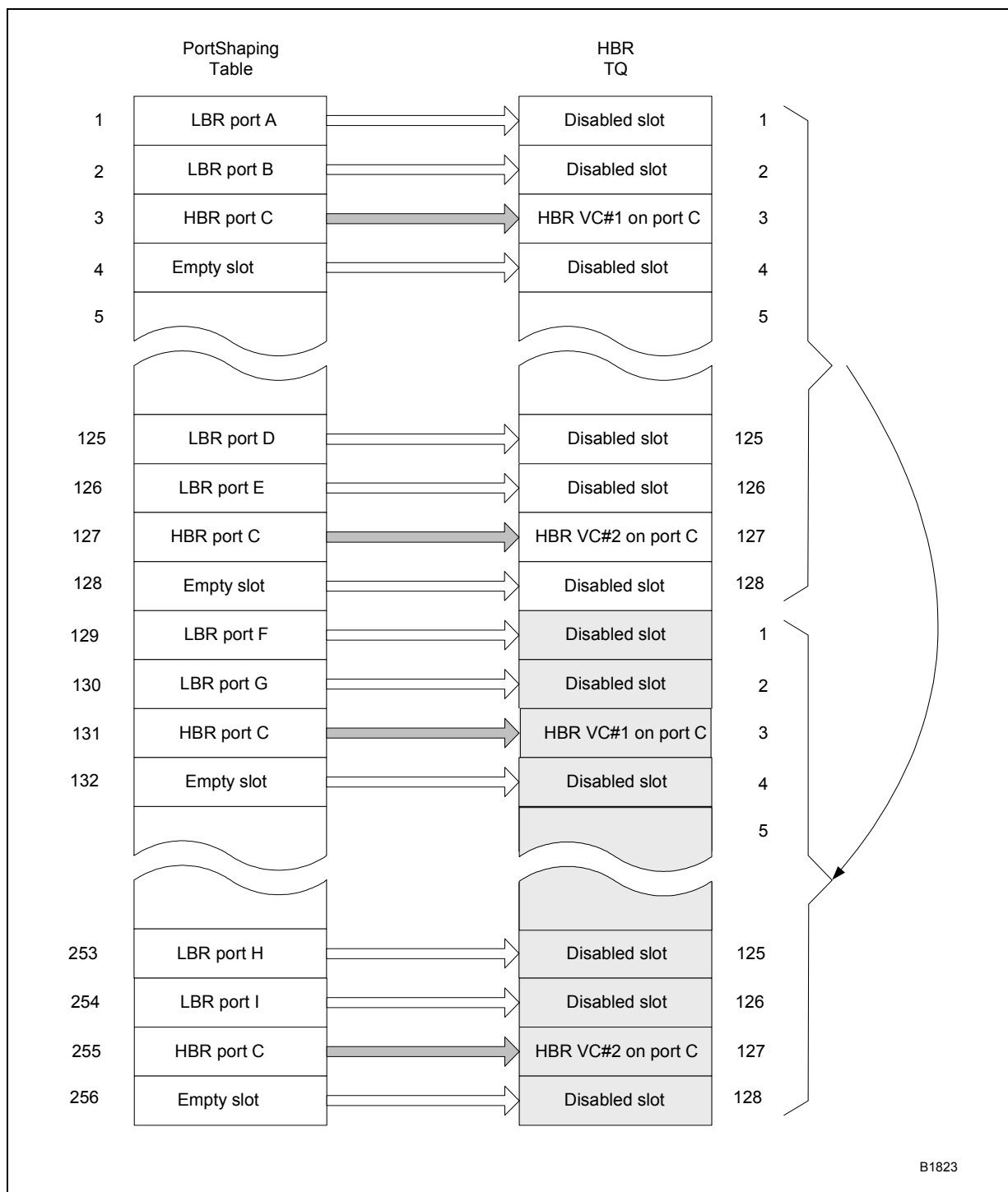
B1822

In the example shown in Figure 23-13, port #1 receives 4/32 of the bandwidth, port #8 receives 2/32 of the bandwidth, and each other ports receives 1/32 of the bandwidth.

23.8.1.4 Local Memory Time queues

Across all the ports, there is just one HBR time queue, the state of which (including the cell counts for the HBR VCQs) is stored in local memory. To ensure that the port rate is not violated by the HBR VCQs belonging to that port, it must be ensured that the HBR VCQ cell transmission slot corresponding to a port matches with the port number for that slot in the static port schedule. See Figure 23-14.

Figure 23-14. Correlation between the PortShaping table and the HBR TQ table



B1823

23.8.2 Write-out Operation

The write-out block extracts the port number from the message the shaper passes. It then indexes into the right time queue for the corresponding port based on the service class CODE and the port number, writes the VCQ entry into this time queue, and then increments by 1 the length of the time queue into which the VCQ was written.

23.8.3 Scheduler Operation

The scheduler uses strict thread ordering and folding techniques.

The scheduler has to do the following to schedule a cell:

1. Read the HBR TQ for the HBR VCQ.
 - If no HBR VCQ or if the HBR VCQ has no cell present, then go to step 2.
 - Else schedule from this HBR VCQ and STOP.
2. Read the static port schedule to find out which port needs to send out a cell.
 - If the end of the current static port schedule window is reached, then read a new window's worth of schedule from SRAM.
3. Do a CAM lookup of the port number to check whether the port state for the current port is already present in the local memory.
 - If yes, simply sleep after signaling the next thread;
 - Else evict LRU and fetch in the state for the current port that needs to be scheduled.
4. Once port the state is available,
 - Do priority queuing among the high bit rate time queue, low bit rate time queues, and the UBR queues for the port, if there is no port flow control.
 - Load new time queues into the departure queue if AGGREGATION of cell transmission slots have passed.
 - Check for port flow control and update the flow control bit for the port accordingly if AGGREGATION number of slots have passed

23.9 Up to 8 Ports Hierarchical Port-rate Shaping

23.9.1 Overview

The case for up to 8 ports differs very little from the case for up to 2048 ports. The only significant difference is that the local memory stores up to 8 unpacked port state structures and there is no need to read and write port state data from and to SRAM. Since 8 ports are available, there is also no need to read the static port schedule from SRAM—all PortShaping table entries fit into local memory. The size of the unpacked port state data structure forced a change of the local memory map—all other aspects are exactly the same.

23.9.2 Scheduler Data Structures

23.9.2.1 Port Information

Table 23-8 shows the performance-oriented port state data structure.:

Table 23-8. Performance-oriented Port State Data Structure

LW	Bits	Length	Name	Notes
LW0	31 to 16	16 bits	Reserved	
	15 to 0	16 bits	Reserved	
LW1	31 to 24	8 bits	Schcount	Count of the number of cells scheduled out by the scheduler; needed for Flow control Ops.
	23 to 16	8 bits	RT_for_the_LBR_TQ	Current time slot for the SRAM time queue. This ranges from 0 to AGGREGATION-1 (the aggregation level) and wraps around after AGGREGATION cells are scheduled.
	1	1 bit	EntryValid	If set to "1" this PortInfo table entry contains valid information.
	0	1 bit	FlowControl	Indicates whether the port is FC blocked.
LW2	31 to 16	16 bits	TQ_offset	Base address for actual TQ (from the beginning of TQ table) - given in elements.
	15 to 8	8 bits	Ubrdq_lm_producer	Pointer to an entry in the UBRDQ ring where new TQ is added (in LWs).
	7 to 0	8 bits	Ubrdq_lm_consumer	Pointer to an entry in the UBRDQ ring where is being serviced (in LWs).
LW3	31 to 24	8 bits	Rtdq_lm_producer	Pointer to an entry in the RTDQ ring where new TQ is added (in LWs).
	23 to 16	8 bits	Rtdq_lm_consumer	Pointer to an entry in the RTDQ ring where is being serviced (in LWs).
	15 to 8	8 bits	Nrt dq_lm_producer	Pointer to an entry in the NRTDQ ring where new TQ is added (in LWs).
	7 to 0	8 bits	Nrt dq_lm_consumer	Pointer to an entry in the NRTDQ ring where is being serviced (in LWs).
LW4	24 to 16	8 bits	UBRwPRlcache	A bit set in this byte indicates the presence and location (which queue) of data in the UBRwPRI queue.
	15 to 0	16 bits	MaxTQmask	TQ count mask.
LW5	31 to 16	16 bits	RTQnum	Number of current RTQ being serviced.
	15 to 0	16 bits	NRTQnum	Number of current NRTQ being serviced.
LW6	31 to 16	16 bits	UBRTQnum	Number of current UBRTQ being serviced.
	15 to 0	16 bits	DQRTlen	Length of current real-time DQ.
LW7	31 to 16	16 bits	DQNRTlen	The length of current non-real-time DQ.
	15 to 0	16 bits	DQUBRTlen	Length of current UBR+ and GFR DQ.
LW8	31 to 0	32 bits	RTQlen	Total length of RTQ being serviced.

Table 23-8. Performance-oriented Port State Data Structure (Continued)

LW	Bits	Length	Name	Notes
LW9	31 to 0	32 bits	NRTQlen	Total length of NRTQ being serviced.
LW10	31 to 0	32 bits	UBRTQlen	Total length of UBRTQ being serviced.
LW11	31 to 0	32 bits	Working_RTQlen	Number of requests already transmitted from RTQ being serviced.
LW12	31 to 0	32 bits	Working_NRTQlen	Number of requests already transmitted from NRTQ being serviced.
LW13	31 to 0	32 bits	Working_UBRTQlen	Number of requests already transmitted from UBRTQ being serviced.
LW14	31 to 0	32 bits	CurTQ	Number of the current TQ that needs to be scheduled from (and is scheduled from if the scheduler is not lagging behind). If the scheduler is lagging behind in time, a pointer to the curtnum is copied into the software ring, provided the length of the curtnum is non-zero.
LW15	31 to 0	32 bits	Reserved	

23.9.2.2 Local Memory Map

Table 23-9 shows the updated local memory map (due to bigger port status data structure).:

Table 23-9. Updated Local Memory Map

LW indexFrom	LW indexTo	Length	Name	Notes
0x000	0x00F	16LW	PortInfo for PORT 1	16-long-word-long port context for a port 1.
0x010	0x01F	16LW	PortInfo for PORT 2	
0x020	0x02F	16LW	PortInfo for PORT 3	
0x030	0x03F	16LW	PortInfo for PORT 4	
0x040	0x04F	16LW	PortInfo for PORT 5	
0x050	0x05F	16LW	PortInfo for PORT 6	
0x060	0x06F	16LW	PortInfo for PORT 7	
0x070	0x07F	16LW	PortInfo for PORT 8	
0x080	0x087	8LW	UBRwPRI for PORT 1	Information about 8 UBR w/priority VCs assigned to a port 1. For each VC stores— 17-bit-long VCQ# (bits 31 to 15) and 15-bit-long cell-counter (bits 14 to 0).
0x088	0x08F	8LW	UBRwPRI for PORT 2	
0x090	0x097	8LW	UBRwPRI for PORT 3	
0x098	0x09F	8LW	UBRwPRI for PORT 4	
0x0A0	0x0A7	8LW	UBRwPRI for PORT 5	
0x0A8	0x0AF	8LW	UBRwPRI for PORT 6	
0x0B0	0x0B7	8LW	UBRwPRI for PORT 7	
0x0B8	0x0BF	8LW	UBRwPRI for PORT 8	

Table 23-9. Updated Local Memory Map (Continued)

LW indexFrom	LW indexTo	Length	Name	Notes
0x0C0	0x0FF	64LW	Reserved	
0x100	0x10F	16LW	PortShaping cache	16 entries from the PortShaping table
0x110	0x11F	16LW	DQs for PORT 1	RT, NRT, and UBR Departure Queues for port 1.
0x120	0x12F	16LW	DQs for PORT 2	
0x130	0x13F	16LW	DQs for PORT 3	
0x140	0x14F	16LW	DQs for PORT 4	
0x150	0x15F	16LW	DQs for PORT 5	
0x160	0x16F	16LW	DQs for PORT 6	
0x170	0x17F	16LW	DQs for PORT 7	
0x180	0x18F	16LW	DQs for PORT 8	
0x190	0x20F	96LW	Reserved	
0x210	0x22F	32LW	128 entries of HBR TQ	Each entry is 8 bits long and contains HBR VCQ# (from 0 to 127).
0x230	0x26F	64LW	128 structures for HBR VCs	Each long word contains two 16-bit-long cell counters.
0x270	0x27F	16LW	Other variables needed by the HBR TQ	

23.9.3 Scheduler Operation

The scheduler has to do the following to schedule a cell:

- Read the HBR TQ for the HBR VCQ.
 - If no HBR VCQ or if the HBR VCQ has no cell present, then go to step 2.
 - Else schedule from this HBR VCQ and STOP.
- Read the static port schedule to determine which port needs to send out a cell.
- Because the state of the port is available,
 - Do priority queuing among the high bit rate time queue, low bit rate time queues, and the UBR queues for the port, if there is no port flow control.
 - Load new time queues into the departure queue if AGGREGATION of cell transmission slots have passed.
 - Check for port flow control and update the flow control bit for the port accordingly if AGGREGATION number of slots have passed.

23.10 Performance Analysis

Table 23-10 shows the worst and average cycle count estimate for the Shaper microblock for all traffic types.

The worst case for the Shaper microblock occurs when there are both:

- Enqueue message with transition
- Dequeue message without transition

The average case for the Shaper microblock occurs when there are both:

- Enqueue message without transition
- Dequeue message without transition

Table 23-10. Worst and Average Cycle Count Estimate for the Shaper Microblock

Phase	CBR		rtVBR		nrtVBR		UBR w/PCR		UBR w/PRIQ	
	AVG	WORST	AVG	WORST	AVG	WORST	AVG	WORST	AVG	WORST
Enqueue	42	81	43	91	43	91	42	81	45	45
Dequeue	75	78	84	91	84	91	75	78	34	34
Total	117	159	127	182	127	182	117	159	79	79

Table 23-11 shows the worst and average cycle count estimate for the Scheduler microblock for all traffic types.

The worst case for the Scheduler microblock occurs once per AGGREGATION when new time queues must be loaded from SRAM to the local memory.

Table 23-11. Worst and Average Cycle Count Estimate for the Scheduler Microblock

Phase	CBR		rtVBR		nrtVBR		UBR w/PCR		UBR w/PRIQ	
	AVG	WORST	AVG	WORST	AVG	WORST	AVG	WORST	AVG	WORST
Scheduler	49	89	34	74	37	77	53	93	42	82
Write-out	54	69	54	89	54	89	54	69	54	55
Total	103	158	88	163	91	166	107	162	96	137

Table 23-12 shows the I/O operations performed in the worst case by the Shaper and Scheduler microblocks for all traffic types.

Table 23-12. I/O Operations Performed in the worst Case by the Shaper and Scheduler

	Operation	Command	CBR, rtVBR, nrtVBR, UBR w/PCR	UBR w/PRIQ
Shaper	Read traffic descriptor	SRAM read	2 * 9LW	2 * 9LW
	Write traffic descriptor	SRAM write	2 * 5LW	2 * 5LW
Scheduler	Load TQ	SRAM test&clear	3	3
	Read slot from TQ	SRAM read	1 * 1LW	
	Increase TQ size	SRAM test&increment	1	
	Write VC# to TQ	SRAM write	1 * 1LW	
	Write to QM scratch ring	SCRATCH put	1 * 1LW	1 * 1LW

Forwarder

The Forwarder microblocks include the following:

- [Chapter 24, “IPv4 Forwarder Microblock”](#)
- [Chapter 25, “IPv6 Forwarder Microblock”](#)
- [Chapter 26, “IPv6 To IPv4 Tunneling Microblock”](#)
- [Chapter 27, “IPv6 To IPv4 Translation Microblock”](#)

IP forwarding microblocks run in a dispatch loop along with other packet processing microblocks (e.g. Diffserv). IP packets are processed based on information contained in IP header. For every packet, the IP header is read from DRAM, the header is validated, a Longest Prefix Match is performed and depending on the nexthop obtained packets are disposed appropriately.

The table below summarizes the different IP forwarding microblocks described in this document:

Microblock	Description	Usage	Cycle Budget
IPv4 Forwarder	RFC 1812 compliant IPv4 forwarder. Uses Longest Prefix Match.	Runs on multiple microengines in parallel. Number of microengines used depends on required data rate (OC-12 to OC-192)	97 for OC48 POS57 for OC192 POS94 10 GBE
IPv6 Forwarder	RFC 2460 & 2373 based IPv6 Forwarder. Uses Longest Prefix Match.	Runs on multiple microengines in parallel. Number of microengines used depends on required data rate (OC-12 to OC-192)	97 for OC48 POS57 for OC192 POS94 10 GBE
IPv6 to IPv4 Tunneling	Provides Configured (RFC 2893), Automatic (RFC 2893) and 6to4 (RFC 3056) tunnels.	Runs on multiple microengines in parallel. Number of microengines used depends on required data rate (OC-12 to OC-192)	97 for OC48 POS57 for OC192 POS94 10 GBE
IPv6 to IPv4 Translation (NAT-PT)	Allows end hosts in v6 realm to communicate with v4 realm and vice versa. Supports Basic NAT-PT, NATPT and Bi-directional NAT-PT.	Runs on multiple microengines in parallel. Number of microengines used depends on required data rate (OC-12 to OC-192)	97 for OC48 POS57 for OC192 POS94 10 GBE

24.1 Overview

The IPv4 Forwarder microblock validates the IPv4 header and performs unicast forwarding on incoming packets. The block is run parallelly on many microengines. The number of microengines depends on the required data rate (OC12 to OC192). Each thread executing this block handles one packet at a time. The threads execute in strict order and the ordering is maintained by the `dl_source` and `dl_sink` blocks in the dispatch loop. This ensures that packet ordering is maintained. The microblock is written such that the same code may be used in the POS, Ethernet and ATM applications.

The microblock performs header checks as specified in [RFC1812] and [RFC2644]. The specific checks performed are described in [Section 24.5, “RFC Compliance” on page 402](#). The functionality of the microblock is described in the high level flow chart in [Section 24.9, “High Level Flow Chart for Microblock” on page 416](#).

There are a few important points about the microblock design.

- The directed broadcast checks [RFC2644] are performed in two stages to reduce the number of trips to SRAM. In the header validation stage, the source address of the packet is subjected to a hash lookup to see if it is not one of the broadcast addresses of locally attached networks. The check on the destination address is performed with the help of route table. For this, a route with the directed broadcast address is added with flag `IPV4_NH_FLAGS_DROP` bit set. In either case, if a match is found, the packet is dropped.
- The checks (3) and (6) from [Section 24.5, “RFC Compliance” on page 402](#), [Table 24-5](#) should be part of the [RFC1812] *SHOULD* checks—see [Table 24-3](#). But these checks are performed along with directed broadcast checks, as they require access to tables in memory. Hence, it is convenient to combine these tests. As discussed earlier, the source address is subjected to a hash lookup, and the destination address is subjected to a route lookup. In either case, if a match is found, the packet is dropped.
- In general any checks on the packet destination address are done with the help of route table since average and worst-case scenarios exercise the LPM lookup anyway. For this, the application has to add the following routes and corresponding flags.
 - For all invalid addresses, set the `IPV4_NH_FLAGS_DROP` flag to drop the packet.
 - For some (or all) multicast address, set the `IPV4_NH_FLAGS_MULTICAST` flag to receive the packets at the core.
 - For broadcast (including the limited broadcast) addresses, set the `IPV4_NH_FLAGS_BROADCAST` flag to receive the packets at the core.
- The microblock sends exception packets to the core through two queues using the dispatch loop framework. The priority of the exceptions is as follows.
 - Packets with the following exception code are sent via the High priority queue.
 - `IPV4_EXCP_LOCAL_DELIVERY`—The rationale is that these packets include Route updates and other signaling data.
 - `IPV4_EXCP_OPTIONS`—The rationale is that the packets might include Router Alert options that the control plane requires.

- All other exceptions are sent via the low priority queue.
- The next hop information may be placed in either SRAM or DRAM (based on a compile time switch in the code). For the applications on the SDK, the next hop information is placed in SRAM. The choice is based on bandwidth availability for SRAM vs DRAM.
- The IPV4 Forwarder microblock can be used together with MPLS, Header compression/IPV6-IPV4 Tunneling and Diffserv (Classifier) microblocks. To facilitate this, the following checks are performed in IPV4 Forwarder microblock when processing a packet.
 - If the `nexthop_id` of the packet is valid (i.e., not -1) then LPM lookup and nexthop processing is skipped. The rationale is that a preceding Classifier block might have already found this information.
 - If the `nexthop_id_type != 0` (i.e., not IPV4), then processing of nexthop information after the LPM lookup is skipped. That is, the nexthop information is processed only for IPV4 nexthop entries. This is to accommodate MPLS or Tunneling blocks which have their own `nexthop_id` to lookup respective Layer2 tables.

Note: It is important to note that if the Tunneling end point shares the same IPV4 address as local interface, then IPV4 Forwarder does not distinguish between tunneled packets and packets to local stack. The packet is simply passed to Tunneling block succeeding IPV4 Forwarder. The Tunneling block has to check whether the packet is actually for local stack and deliver it to the core as exception packet.

- If the `nexthop_id_type != 0`, then IPV4 Forwarder does not decrement TTL and update checksum. This is to facilitate blocks such as MPLS, succeeding the forwarder which might choose to decrement the TTL by more than one.

24.2 Assumptions and Dependencies

The following assumptions are made in the design and implementation of the IPV4 Forwarder microblock:

- The primary goal of this microblock is to achieve maximum performance. Hence the following assumptions apply
- The microblock doesn't perform any error checking on the data structures shared with Xscale. The Intel XScale® core code has to populate the structures with correct values. Otherwise, the results are unpredictable.
- The microblock tries to take advantage of the layout of the above structures and the member fields inside. Hence, the Intel XScale® core code cannot assume to re-use the reserved fields for some other purpose without affecting the microblock functionality.
- A trade-off is evaluated between efficiency of the microblock and the degree of details provided in the exception codes when the packet has to be sent to the Intel XScale® core.
- All reserved fields in the data structures must be cleared to zero, unless otherwise specified.
- This block is an IPV4 Unicast Forwarder. No multicast forwarding is supported and packets with multicast addresses are sent to the Intel XScale® core as exceptions.
- The threads executing this microblock does not maintain strict ordering—because of the different amounts of processing and I/O required per packet. As a consequence, the packets can get out of order by the end of microblock. Hence, it is the responsibility of the dispatch loop using this microblock to ensure packet order around this block.

- The packet ordering is not maintained when there are exception packets, as these packets are sent to Intel XScale® core and the microblock—or the dispatch loop—has no way of ensuring the packet ordering on the Intel XScale® core.
- The current implementation of the microblock heavily uses the xbuf macros (*Intel® Internet Exchange Architecture (IXA) Optimized Data Plane Libraries Reference Manual*) to access the fields in the IP header. The microblock assumes that these xbuf macros can operate on transfer registers, general purpose registers (GPRs) as well as local memory.
- The TOS field is not used in the IPv4 Forwarder and is left unmodified.
- The microblock always maintains 32-bit counters for performance reasons. To support 64-bit counters, it is recommended that the Intel XScale® core code periodically read these counters and update its local copy.
- The forwarding functionality is disabled on a particular output interface by setting the IPV4_NH_FLAGS_DROP flag in all the nexthop entries on this output port.
- The microblock checks size of IP datagram against the MTU field in the nexthop information to determine if fragmentation is needed. The MTU field should be calculated as:

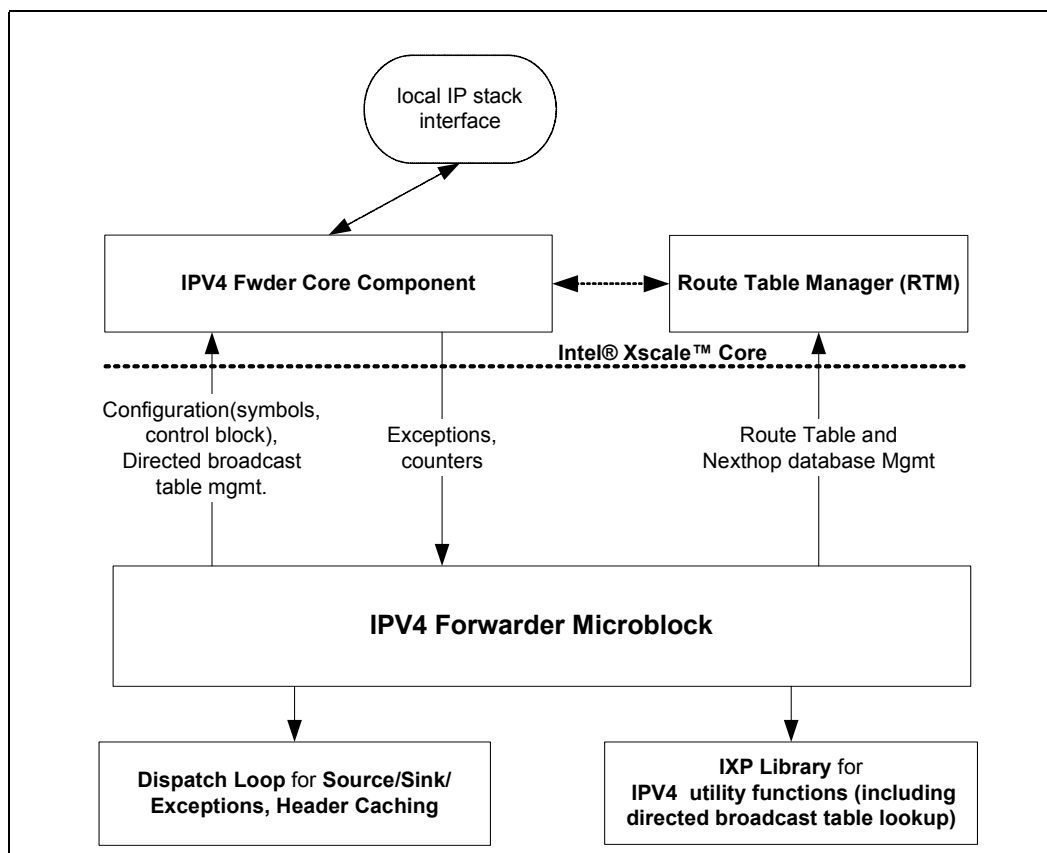
sizeof L2 frame - sizeof L2 header

- The microblock always performs the *MUST* checks specified in RFC1812 (see [Section 24.5, “RFC Compliance” on page 402.](#)) It is assumed that the applications would always require these checks. The RFC *SHOULD* checks may be disabled as a compile time option.
- The Intel XScale® core code has to create and maintain/update the shared data structures. The microblock simply reads and processes the information. Only counters are an exception, as they are updated by the microblock.
- When a packet is dropped by microblock, packet content are not logged as recommended by [RFC1812], as it is not practical to do this at such high data rates.
- The microblock doesn't always fill the packet meta data with nexthop information. See [Section 24.9](#) for more information. Hence, depending on the phase of the microblock from where the exception originated, the Intel XScale® core code has to obtain this data and fill the meta data structure.
- All address and offsets specified in this section are byte addresses.
- The values in the data structures shared by this block are in network byte order—that is, in big-endian order.

24.3 Dependencies

The main dependencies of this microblock are shown in this figure. The inward arrows indicate dependency on the IPV4 Forwarder microblock. The outward arrows show what the microblock expects from other modules.

Figure 24-1. IPv4 Microblock Dependencies



24.4 Configuration Options

24.4.1 Build Switches

Table 24-1 identifies compile time build switches, which can be turned on or off as per the requirements of a specific application.

Table 24-1. Listing of Build Switches

Symbol	Description
MICROENGINE	Enables microengine specific definitions
MICROCODE	Enables definitions specific to microcode. If not enabled, then the definitions apply to microC code.
CHIP_VERSION	Enables IXP library to work on IXP2xxx processors
META_CACHE_SIZE	Defines the number of long words of meta data to be cached
IPV4_START_ME	Indicates which microengine is the first in a series of MEs that run IPv4

Table 24-1. Listing of Build Switches (Continued)

Symbol	Description
DL_NEXT_ME	Next microengine running IPv4 w.r.t the microengine in question. Needed by dispatch loop.
RFC1812_SHOULD	Enables 'SHOULD' checks specified in RFC1812
NEXTHOP_INFO_SRAM	Indicates that the Next hop database is in SRAM. If this switch is not enabled, then the microblock assumes the database is in DRAM.
RFC2644_CHECKS	Enables RFC2644 directed broadcast checks on source address. The check on destination address is through the route table.
DBCAST_TABLE_BLOCK_SIZE	The number of SRAM long words to be read at a time when directed broadcast check is performed. The RFC2644_CHECKS should be enabled for this switch to take effect.
IP_HDR_OFFSET	The number of bytes after which the IP header can be found, relative to the beginning of the cached packet header. Enabling the switch makes the IPV4 Microblocks more efficient as the underlying xbuf macros (<i>Intel® Internet Exchange Architecture (IXA) Optimized Data Plane Libraries Reference Manual</i>) generate optimized code when the offset passed is a compile time constant. This switch should used to obtain optimal performance when the microblock is expected to support only one media type for which the IP header offset is always fixed. This switch is used in the dispatch loop, which uses the microblock.
PROCESS_CONTROL_BLOCK	Defining this switch enables the microblock to read and process the control block information from SRAM. See Section 24.6.1 . If this switch is not enabled, dynamic configuration information cannot be communicated to the microblock, but the performance increase.

24.4.2 Default Configuration

The build switches with the microblock are as follows:

- MICROENGINE¹
- CHIP_VERSION=IXP2xxx¹
- META_CACHE_SIZE=8
- When running in multiple MEs define IPV4_START_ME for the first ME
- Subsequently define DL_NEXT_ME=x for each ME, where x is as shown in an example below where IPv4 is running on 4 MEs : 2, 3, 4 & 5 [Table 24-2](#). The microengine number in 0xAB format implies the microengine B in cluster A.

Table 24-2. Table of Next Microengine Values

ME	Next ME
0x02	0x03
0x03	0x10
0x10	0x11
0x11	0x02

1. These switches should never be turned off. Other switches are configurable, depending on the application.

- RFC1812_SHOULD
- RFC2644_CHECKS
- NEXTHOP_INFO_SRAM
- DBCAST_TABLE_BLOCK_SIZE=8
- PROCESS_CONTROL_BLOCK
- IP_HDR_CACHE_LM
- LM_DBCAST_TABLE
- IPV4_COUNTERS
- IPV4_COUNTERS_SCRATCH
- COMPRESSED_NEXTHOP_INFO

24.5 RFC Compliance

The microblock always performs the [RFC1812] specified *MUST* header checks. These checks are summarized in [Table 24-3](#).

Table 24-3. RFC1812 *MUST* Check Items

Serial No.	RFC1812 'MUST' Check	Action
1	Packet size reported is less than 20 bytes	Drop
2	Packet with version!= 4	Drop
3	Packet with header length < 5	Drop
4	Packet with header length > 5	Exception. See 5.8.2
5	Packet with total length field < 20	Drop.
6	Packet with invalid checksum	Drop
7	Packet with destination address equal to 255.255.255.255	Exception. See Section 24.8.2 .
8	Packet with expired TTL (checked after a forwarding decision is made)	Exception. See Section 24.8.2 .

The other [RFC1812] recommended checks can be turned on, by enabling the RFC1812_SHOULD switch. These checks are summarized in [Table 24-4](#).

Table 24-4. RFC1812 *SHOULD* Check Items

Serial No.	RFC1812 'SHOULD' Check	Action
1	Packet length < total length field	Exception. See Section 24.8.2 .
2	Packet with source address equal to 255.255.255.255	Drop
3	Packet with source address 0	Drop
4	Packet with source address of form {127, <any>}	Drop
5	Packet with source address in Class E domain	Drop
6	Packet with source address in Class D (multicast) domain	Drop

Table 24-4. RFC1812 *SHOULD* Check Items

Serial No.	RFC1812 'SHOULD' Check	Action
7	Packet with destination address 0	Drop
8	Packet with destination address of form {127, <any>}	Drop
9	Packet with destination address in Class E domain	Drop
10	Packet with destination address in Class D (multicast) domain	Exception. See Section 24.8.2 .

In addition, by enabling `INCLUDE_DBCAST_CHECK` switch the [RFC2644] checks are turned on. These checks are summarized in [Table 24-5](#).

Table 24-5. Optional RFC2644 Check Items

Serial No.	RFC2644 Check	Action
1	Packet source address of the form {<network-prefix>, -1}	Drop
2	Packet source address of the form {<network-prefix>, 0}	Drop.
3	Packet with source address of form {0, <host-number>}	Drop.
4	Packet destination address of the form {<network-prefix>, -1}	Drop
5	Packet destination address of the form {<network-prefix>, 0}	Drop.
6	Packet with destination address of form {0, <host-number>}	Drop.

24.6 Data Structures

This section describes the data structures relevant to the IPv4 forwarder microblock. These data structures are shared between various modules as identified.

24.6.1 Control Block

The control block contains dynamic configuration information for the microblock. Currently, this control block contains a table, which identifies whether forwarding is enabled or disabled on a particular input port. The microblock may continue to receive packets destined for local interfaces even though forwarding is disabled on a particular port.

A bit mask is maintained, each bit representing the enable/disable status of one input port. If the bit is set, then forwarding is enabled. Otherwise, if the bit is clear, forwarding is disabled and packets are dropped unless they are destined for a local interface. For output ports, the `IPV4_NH_FLAGS_DOWN` flag in the next hop structure is used to indicate whether forwarding is enabled on a port.

The control block resides in SRAM and is updated by Intel XScale® core code. The layout of the block is shown in Table 24-6.

Table 24-6. Control Block Layout

LW	Bits	Description
0	0	Enable/Disable forwarding on input port 0.
0	1	Enable/Disable forwarding on input port 1.
0
0	31	Enable/Disable forwarding on input port 31.
1 thru 3	ALL	Reserved.

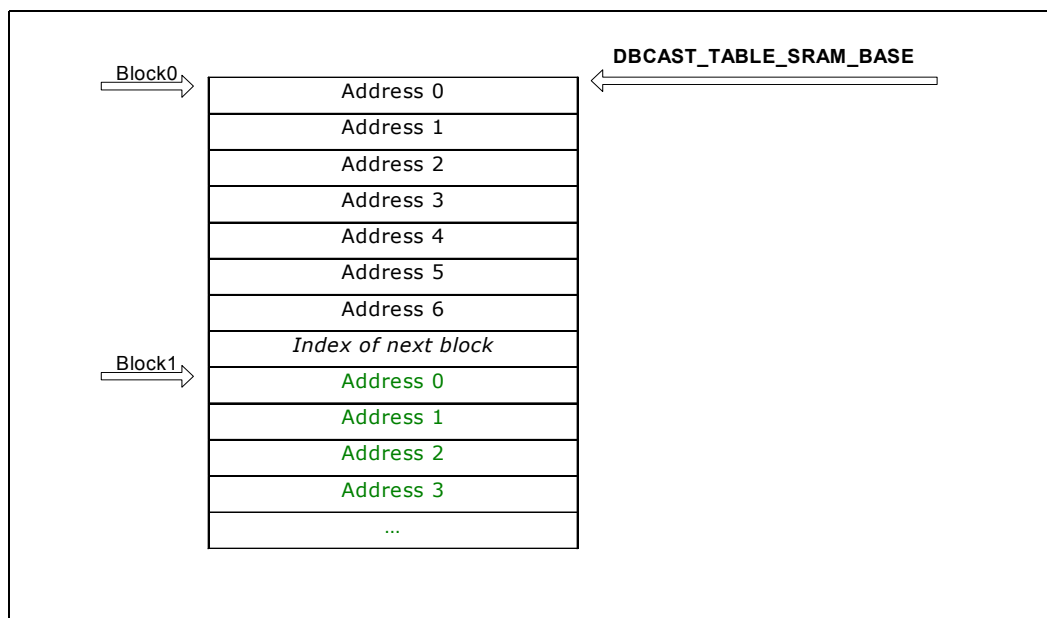
24.6.2 Directed Broadcast Table

The directed broadcast table contains the list of addresses that are invalid. The following addresses belong to this set as per [RFC1812]:

- {<Network-prefix>, -1}
- {<Network-prefix>, 0}

The directed broadcast table is populated by code running on the Intel XScale® core. The IPv4 Microblock uses macros—or C function calls—from the IXP Library (*Intel® Internet Exchange Architecture (IXA) Optimized Data Plane Libraries Reference Manual*) to perform a lookup on this table. As mentioned earlier, only the source address is subjected to a lookup in this table. The destination address check is a by-product of the lookup in the route table. The layout of the table is shown in Figure 24-2.

Figure 24-2. Directed Broadcast Table Layout¹



1. Where the value of DBCAST_TABLE_BLOCK_SIZE is eight.

The table in SRAM starts at address specified as the symbol `DBCAST_TABLE_SRAM_BASE`. The table is divided into blocks of size `DBCAST_TABLE_BLOCK_SIZE`—note that the above figure illustrates the table layout when the block size is eight. This reduces the number of SRAM reads while the address is compared. The following design is explained with respect to [Figure 24-2](#).

The index of the block is obtained by performing a hash operation on the IPv4 address as follows:

The index is an eight-bit value, which implies a maximum of 256 blocks exist.

Note: Given an IPv4 address of form A.B.C.D, the index of the block is $A \wedge B \wedge C \wedge D$; with the least significant eight bits to be considered.

A value of zero for the address field indicates that the entry is free. When the block is full—that is, address 0 thru 6 have non-zero values in the above table—then an additional address cannot be added to this block. In this case, another empty block can be allocated and the address is stored in that block. The index of this block replaces the last long word—*index of next block* field—of the previous block. If there is no list of blocks, the value of this long word is zero.

The microblock compares the address in a block, starting at index zero and finishing at index seven. It aborts the comparison the moment it encounters a zero value for this field. Hence, the Intel XScale® core code should not leave free entries between valid entries.

The Intel XScale® core code may choose not to build a link list of blocks, in which case the microblock compares at most seven entries.

24.6.3 Next Hop Information

Tables [24-7](#) and [24-8](#) show the next hop information associated with each route. The result of route lookup is a nexthop index, which is used to obtain the next hop information. The Next Hop data structure may be 4 long words or 2 long words (compressed format). The compressed format is used in the OC-192 POS application.

[Table 24-7](#) shows the uncompressed format.

Table 24-7. Next Hop Information Associated with Each Route—Uncompressed Format

LW	Bits	Size	Field	Description
0	31:8	24	Reserved	
0	7:0	8	Flags	Flags to indicate quick action as next hop for the packet, if corresponding bit is set.
1	31:24	8	Fabric Port ID	Blade ID with respect to CSIX fabric
1	23:20	4	Next hop ID type	The type of next hop ID to indicate which table to use
1	19:16	4	Reserved.	
1	15:0	16	Next hop ID	ID used on the Egress to lookup the outgoing Link layer information.
2	31:16	16	MTU	Maximum Transmission Unit for out going physical interface.
2	15:0	16	Output port ID	Out going physical interface
3	31:0	32	Reserved	

The table below shows the compressed format:

Table 24-8. Next Hop Information Associated with Each Route—Compressed Format

LW	Bits	Size	Field	Description
0	31:16	16	Next hop ID	ID used on the Egress to lookup the outgoing Link layer information.
0	15:8	8	Fabric Port ID	Blade ID with respect to CSIX fabric
0	7:0	8	Flags	Flags to indicate quick action as next hop for the packet, if corresponding bit is set.
1	31:16	16	MTU	Maximum Transmission Unit for out going physical interface.
1	15:12	4	Reserved	Must be set to 0
1	11:8	4	Next Hop ID type	Zero (default) indicates link layer information will be used. If non-zero, indicates specific lookup table to use.
1	7:0	8	Output port ID	Outgoing physical interface

24.6.3.1 Flags

Table 24-9 lists the flags specifying the action to carry out on a packet. The bit positions are relative to the starting bit position—`IPV4_NH_FLAGS_START_BIT` which is zero. All other bits are reserved.

Table 24-9. Packet Action Flags

Bit Position	Global name	Description
0	<code>IPV4_NH_FLAGS_LOCAL</code>	The packet is for local interface. Packet is sent to XScale.
1	<code>IPV4_NH_FLAGS_DOWN</code>	The interface/nexthop is down. The packet is sent to XScale.
2	<code>IPV4_NH_FLAGS_DROP</code>	The packet is be dropped.
3 - 7	Reserved.	

24.6.3.2 Nexthop ID type

This field indicates the type of table to be used when performing a lookup using nexthop ID. The default value of 0 indicates the IPv4 Forwarder uses link layer information. MPLS or IPv6-IPv4 Tunneling blocks can perform lookup in their corresponding link layer tables using a non-zero value, which has to defined when the application is designed. The following values are defined in the IXA SDK libraries:

- `NHID_TYPE_IPV4` (0x0) indicates IPv4 table should be used.
- `NHID_TYPE_IPV6` (0x1) indicates IPv6 table should be used.
- `NHID_TYPE_MPLS` (0x2) indicates MPLS table should be used.
- `NHID_TYPE_TUNNELING` (0x3)

24.6.3.3 Nexthop ID

The Next Hop ID is passed over the CSIX fabric to an Egress processor where it is mapped to a layer-2 header to be prepended to the buffer.

The Nexthop ID has an invalid value, which is -1—that is, all bits set.

When the Nexthop ID is -1, the microblock sends the packet to the core as exception packet. See [Section 24.8.2](#).

Since the unused entries in the route table are initialized to zero, the microblock picks up zero as nexthop index when there is no matching route. Instead of checking nexthop index for zero, the microblock goes ahead and reads the corresponding nexthop information. This nexthop entry is reserved to store default route information. In such a case, the microblock automatically selects the default route—if one is installed—when a route lookup fails. If there is no default route, then nexthop ID at index zero must be -1.

24.6.3.4 Fabric Port ID

The fabric port ID is used to determine which blade on the fabric the packet is to be sent to.

24.6.3.5 Output Port ID

The output port number is used to identify the port on which the packet must be transmitted on the way out of the system. The application may choose to interpret this field as the physical or logical port number. The microblock uses this field to check for redirection of the packet.

24.6.3.6 MTU

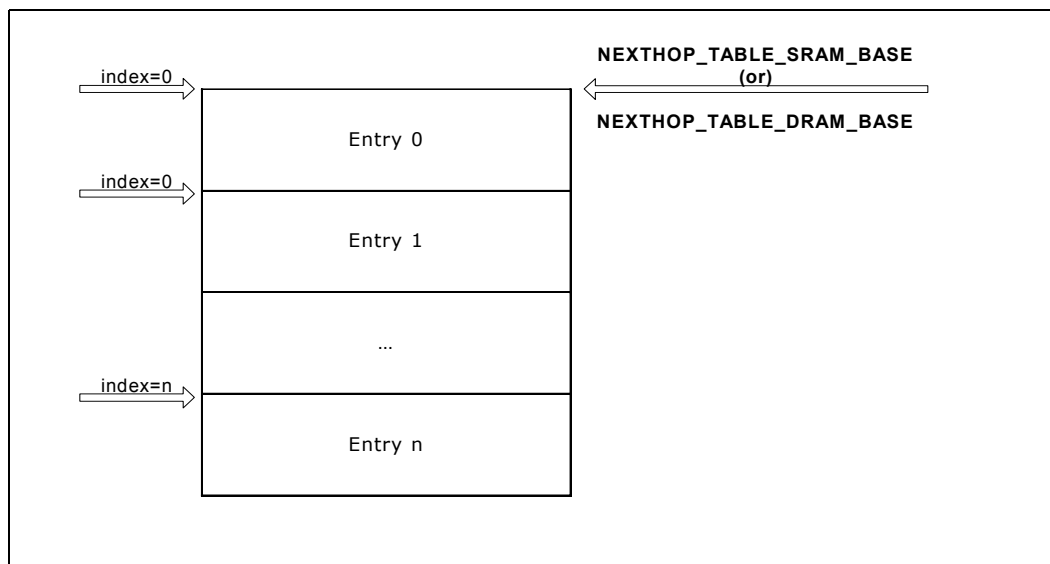
The maximum transmission unit is used by the microblock to check if size of the IP datagram is more than the MTU. If so, the packet is sent to the Intel XScale® core.

24.6.4 Nexthop Database

The nexthop database is used to store nexthop information corresponding to a route. This database is essentially an array of nexthop information entries, with each entry indexed by the nexthop index obtained via the route lookup.

The nexthop database can be in SRAM or DRAM. The layout of the database is shown in Figure 24-3.

Figure 24-3. Nexthop Database Layout



24.6.5 IPv4 Counters

The table below shows the 32-bit counters stored in SRAM and their respective offsets from base. The base address is patched as the symbol `STATS_TABLE_SRAM_BASE` by the Intel XScale® core.

Table 24-10. IPv4 Counters and SRAM Offsets

32-bit Counter	Offset from Base
Packets Received	0x0
Packets Forwarded	0x4
Packets dropped	0x8
Packets sent as exception	0xc
Packets with bad header	0x10
Packets with bad IP total length	0x14
Packets with bad TTL	0x18
Packets with no route	0x1c
Packets with length too small	0x20

The microblock always maintains 32-bit counters. Code running on the Intel XScale® core can keep 64-bit counters if required (see [Section 2.5, “Statistics and Handling of 64-bit Counters”](#) on [page 63](#)).

24.6.6 Route Table

The route table is described in detail in the next section. The microblock depends on the Route Table Manager on the Intel XScale[®] core to create and manage the route table.

24.7 Longest Prefix Match Lookup

24.7.1 Introduction

The longest prefix match (LPM) algorithm is a key component of the microblock and deserves a detailed description. The design of the routing table algorithm and software is based on the reference design that shipped with IXA SDK 1.0 [IXP1200 SRM]. This algorithm is optimized for speed, performing route lookups in the microengines using a maximum of five SRAM references. The algorithm, however, uses large amounts of memory to obtain this speed advantage. A single instance of the routing table using the algorithm in the reference design occupies a minimum of 256 KB. The primary motivation of this section is to describe the route table structure and lookup algorithm, which are of interest to the microblock.

24.7.2 Data Structures

The algorithm we use involves the use of trie blocks—or simply tries—to implement the route table. A trie is an array of entries indexed by a portion of the destination IP address. The trie entries can contain index to nexthop information, an index to another trie, or both. The use of index instead of pointer increases the amount of memory that can be referenced and hence the number of nexthop entries and tries. The current layout assumes 16 bits for the next trie, which implies an address space of 64k tries. If a larger number of tries is needed, more bits can be used at expense of bits for the next hop index.

In our design, each trie contains sixteen entries—except the starting block, which is the root trie. This implies that the lookup algorithm considers four bits at a time from the destination address. The number of tries traversed in any particular lookup is based on how long the prefixes are in the routes installed in the route table. The trie entry layout is shown in [Table 24-11](#).

Table 24-11. Trie Entry Layout

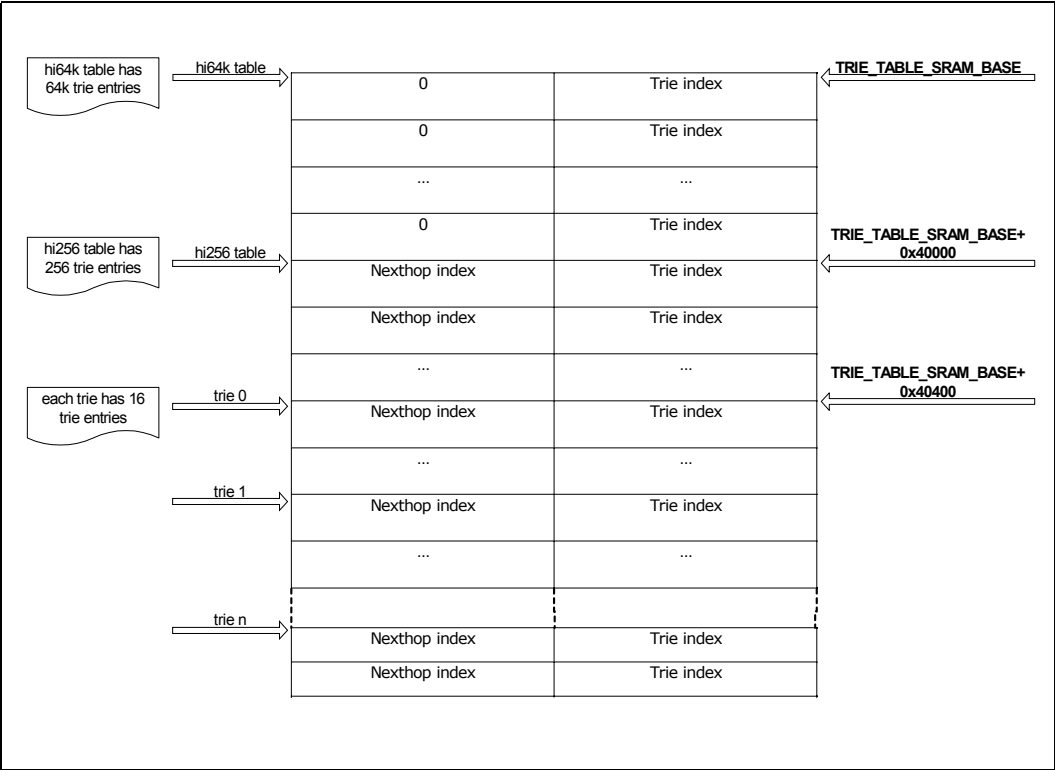
Index to Nexthop information(16 bits)	Index of next trie (16bits)
---	-----------------------------

The algorithm is called a dual lookup algorithm as two tables are used to perform a route lookup. The starting block of the first table is a hi64k table. This has 64k trie entries, indexed by the most significant 16 bits of the IP address. Routes with prefix length greater than 16 are added to this table. Hence the lookup uses 16-4-4-4-4 bits from the IP address.

The starting block of the second table is a hi256 table that has 256 Trie entries indexed by the most significant eight bits of the IP address. Routes with prefix length less than or equal to 16 are added to this table. Hence the lookup uses 8-4-4 bits from the IP address.

The route table layout in SRAM is shown in [Figure 24-4](#).

Figure 24-4. Route Table Lookup

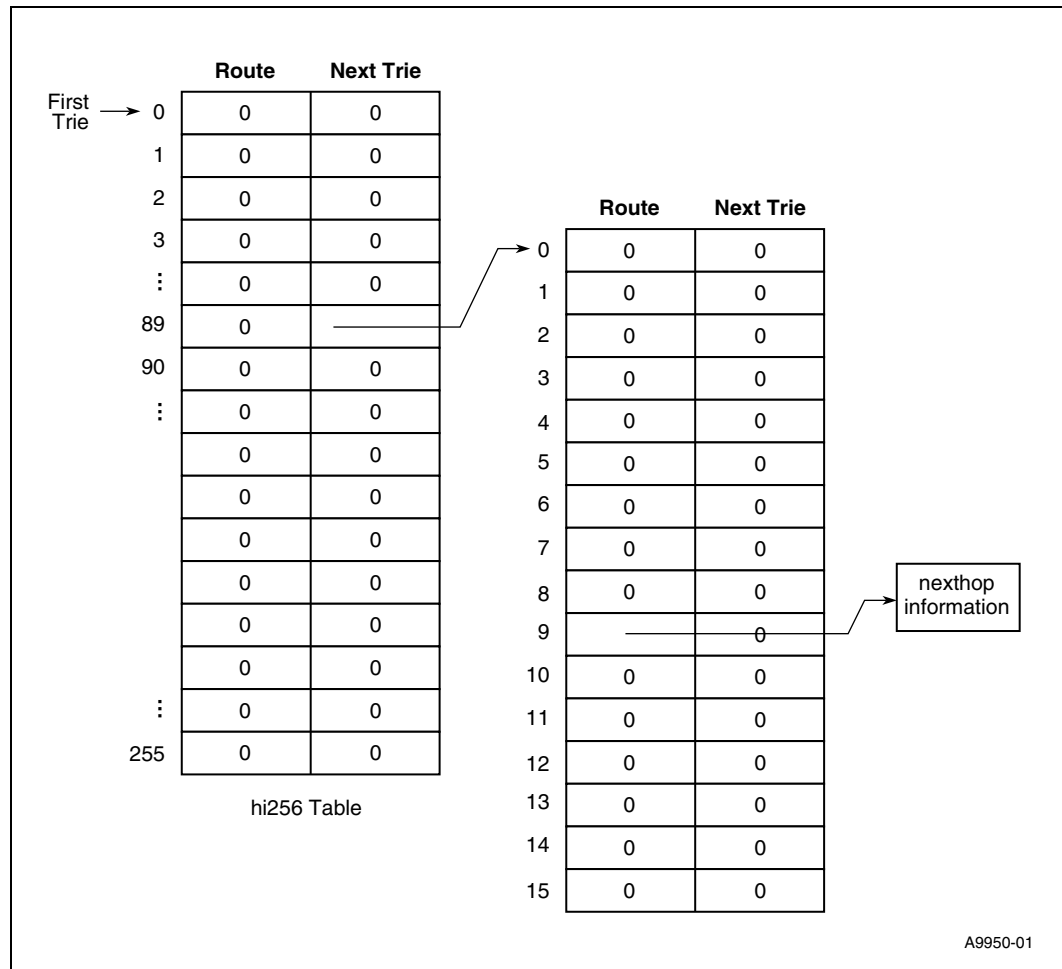


The purpose of having two tables is to have at most five dependent SRAM reads for a route lookup as well as to reduce the number of trie entries—one minimum, eight maximum—to update when a route is added/deleted.

Note: The number of trie entries to be updated when the prefix length is less than or equal to four varies from 16 thru 128. But such small prefix lengths are not used.

[Figure 24-5](#) shows a sample trie structure of a table with a single 12-bit prefix route installed.

Figure 24-5. Trie Table with One Route¹



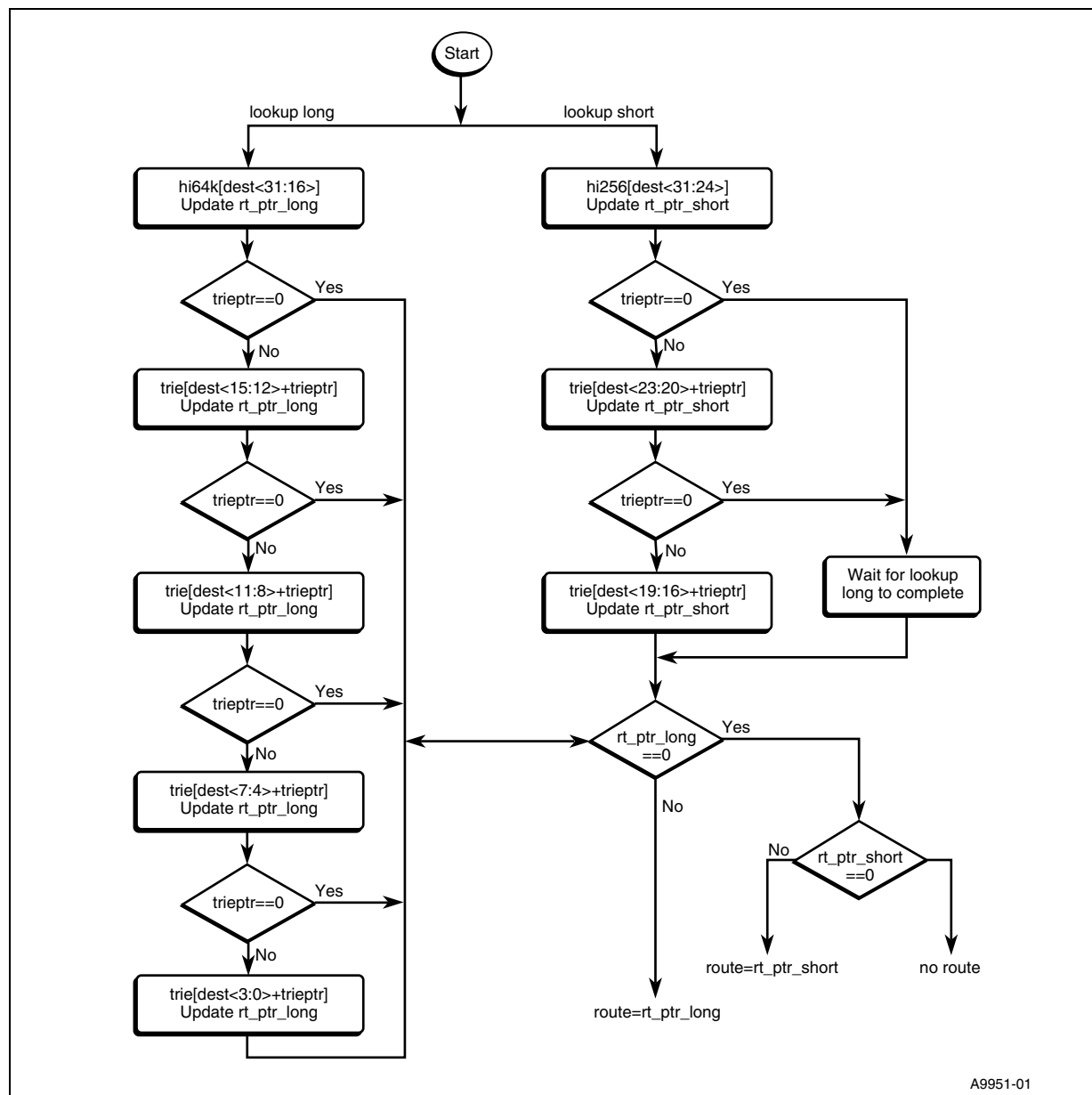
1. This figure shows a routing table with a route installed. The route has a destination address of 0x59900000 and a mask of 0xffff0000. If the route has less than 12 but more than 8 bits, the same Trie structure is used, but with duplicate route pointers to the same next-hop entry. For example, a netmask of 0xffe00000 in the above example would add a next-hop route entry pointer in slot 8 of the second Trie. The pointer would be the same as the one currently in slot 9.

The most significant 8 bits of the IP address index the hi256 table in [Figure 24-5](#).

24.7.3 Lookup

Given the trie structure above, the flow chart shown in Figure 24-6 illustrates the process of performing a route lookup.

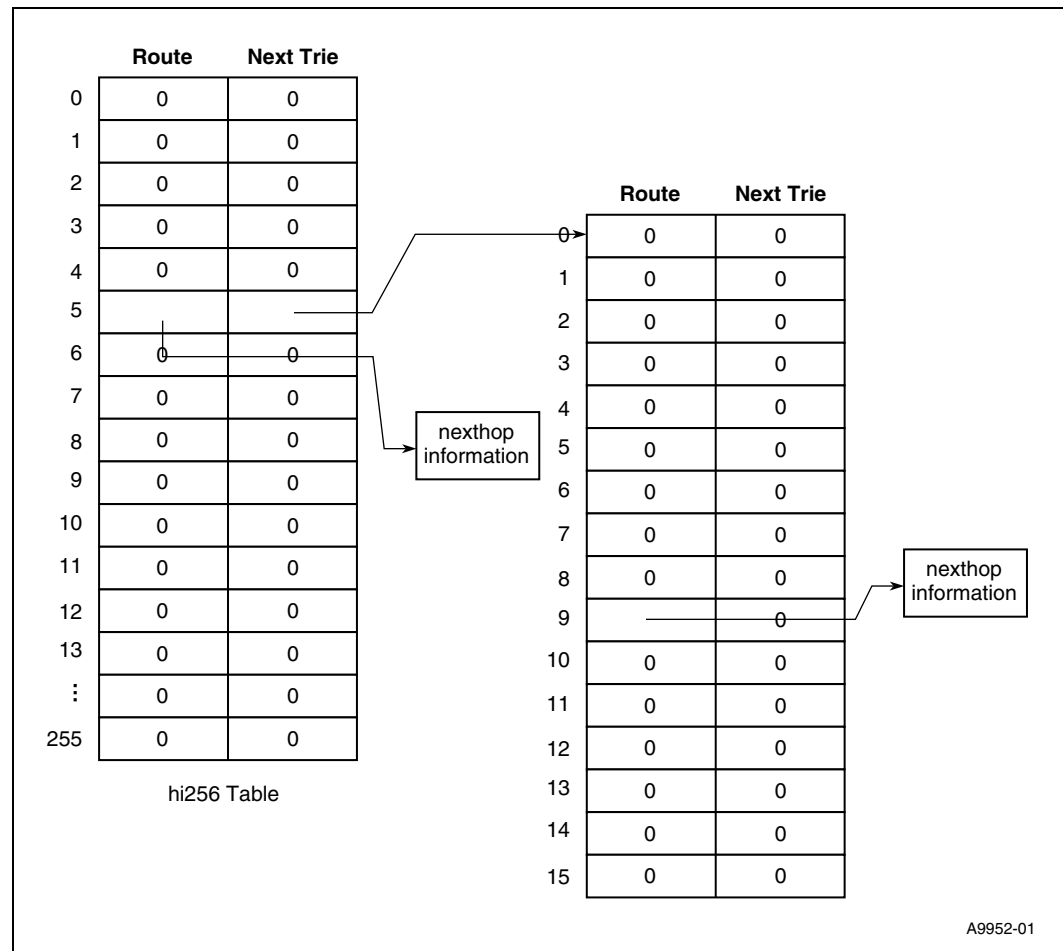
Figure 24-6. Longest Prefix Match Dual Lookup



There are a couple of important things to note here. The first is that the next-hop route index does not necessarily have to come from the last trie considered in the lookup. If the last trie contains a zero route index in the entry pointed to by the IP address, the last non-zero route index encountered is used. This enables the algorithm to correctly look up route addresses in scenarios where the

destination IP address matches the first part of an installed route but not the end, and yet the destination still matches a shorter prefix route considered earlier in the trie structure. The following diagram illustrates this special case.

Figure 24-7. Trie Table with Two Routes¹



1. This figure shows a routing table with two routes installed. The first route has a destination address of 0x05000000 and a mask of 0xff000000. The second route has a destination address of 0x05900000 and a mask of 0xff000000. A route lookup for 82.1.1.1 (0x05010101) follows the next Trie index to the second trie, but when it finds no route or next trie, it should use the cached nextthop entry 1 index found when considering the first trie, as described above.

24.8 Intel XScale® Core Interface

24.8.1 Symbols

The symbols listed in [Table 24-12](#) have to be patched from the Intel XScale® core component of the IPv4 Forwarder microblock.

Table 24-12. List of IPv4 Microblock Symbols

Symbol	Description
STATS_TABLE_SRAM_BASE	Counters are maintained starting at this address in SRAM.
TRIE_TABLE_SRAM_BASE	The Trie tables for LPM start at this address in SRAM.
NEXTHOP_TABLE_BASE	The next hop database starts at this address in SRAM or DRAM. Using a build switch identifies the memory type. See 5.4.1
DBCAST_TABLE_SRAM_BASE	The directed broadcast hash tables start at this address in SRAM.
_IPV4FWDER_ID	The IPV4 Forwarder block ID
_IPV4FWDER_OUTPUT	The next block ID to which IPV4 Forwarder microblock has to send packets.
CONTROL_BLOCK_SRAM_BASE	The base address of the control block in SRAM.

24.8.2 Exception Codes

[Table 24-13](#) identifies the exception codes generated by IPv4 Forwarder block that are passed along with the packet to the Intel XScale® core counterpart.

Table 24-13. IPv4 Microblock Exceptions

Exception code	Value	Description
IPV4_EXCP_OPTIONS	0x11	The IP header received with options.
IPV4_EXCP_LENGTH_MISMATCH	0x12	The packet length reported by link layer is less than the total length field.
IPV4_EXCP_BAD_TTL	0x13	The packet can't be forwarded as the TTL has expired.
IPV4_EXCP_MULTICAST	0x14	The packet received is a multicast packet.
IPV4_EXCP_FRAG_REQUIRED	0x15	The MTU for outgoing interface is less than the packet size.
IPV4_EXCP_REDIRECT	0x16	The outgoing port is same as the one on which the packet is received.
IPV4_EXCP_DOWN	0x17	The nexthop information has 'IPV4_NH_FLAGS_DOWN' flag set.

Table 24-13. IPv4 Microblock Exceptions

Exception code	Value	Description
IPV4_EXCP_NO_ROUTE	0x18	There is no route in the route table corresponding to the packet destination address. This is indicated by nexthop ID value of -1 in nexthop information at index zero.
IPV4_EXCP_LOCAL_DELIVERY	0x19	The packet is for a local interface. The nexthop information has 'IPV4_NH_FLAGS_LOCAL' flag set.
IPV4_EXCP_LIMITED_BROADCAST	0x1a	The packet received as limited broadcast.

24.9 High Level Flow Chart for Microblock

The flow charts shown in Figure 24-8 and Figure 24-9 describe the path of IP packet inside the IPV4 Forwarder macro. The flow charts provided do not describe the path within the IXP IPV4 macros. Within these figures the blocks marked in **bold red text** are in critical path for processing the packet. The microblock is optimized to execute this path more efficiently.

Figure 24-8. High Level Flow Chart for IPv4 Microblock

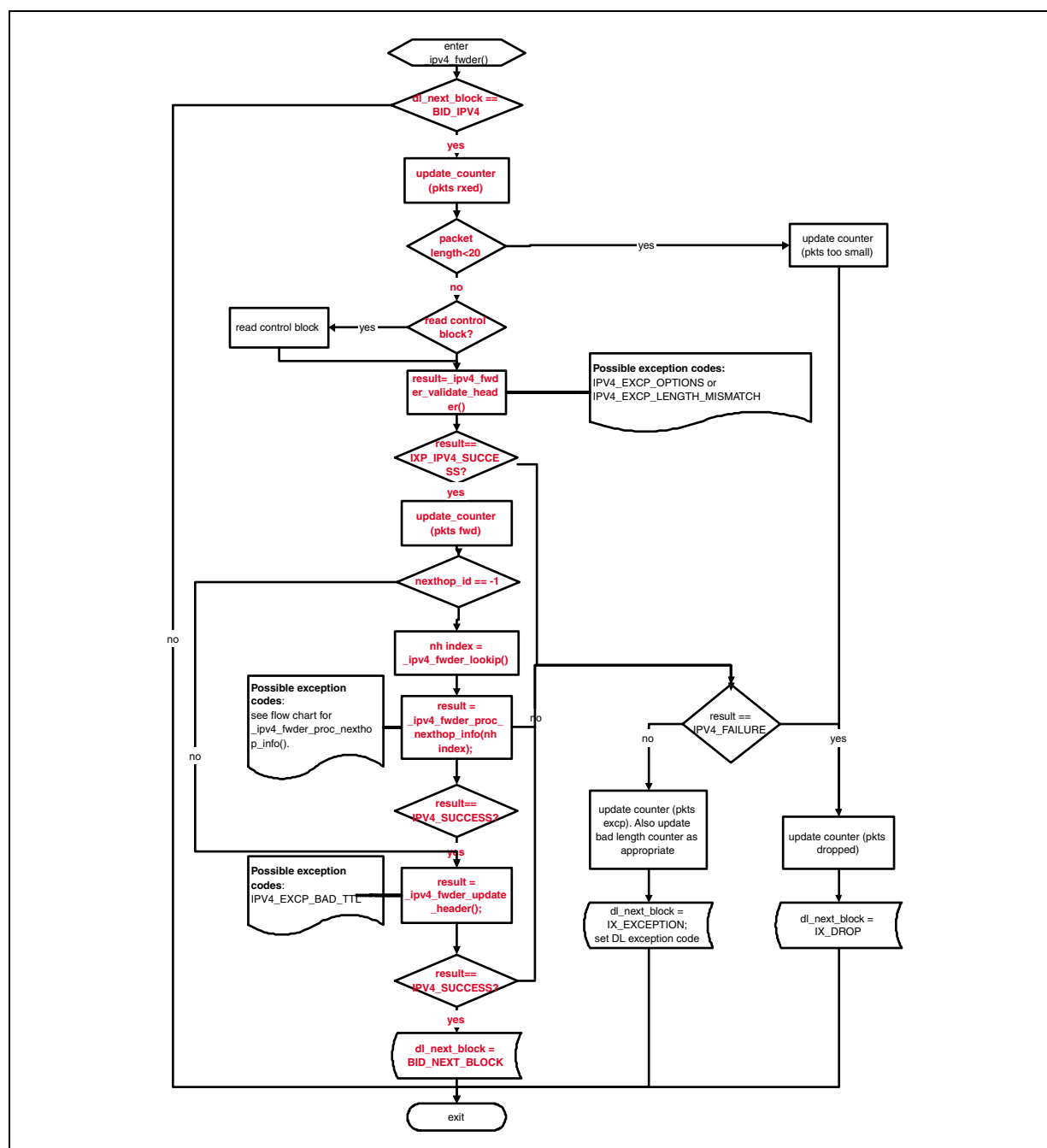
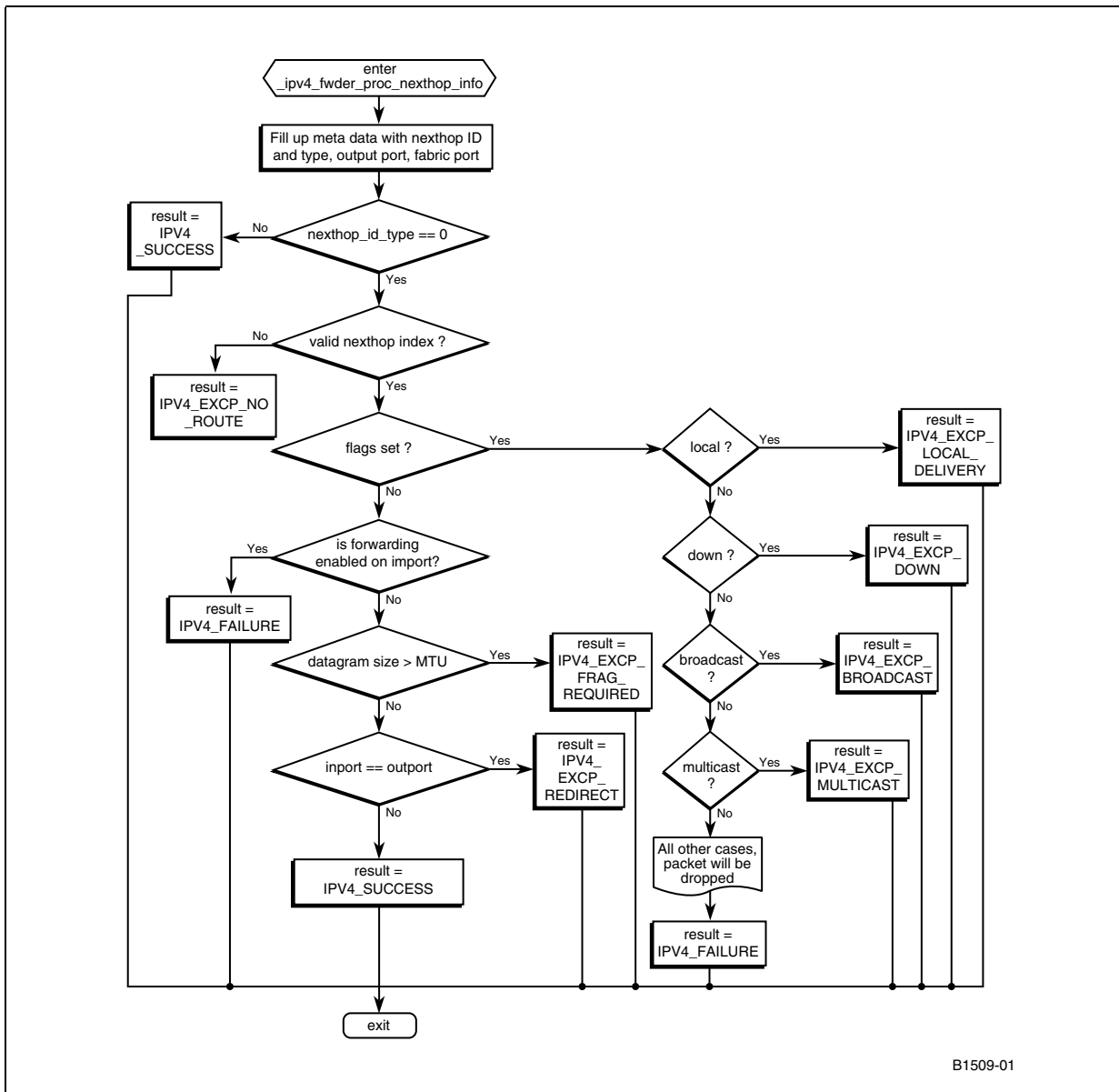


Figure 24-9. IPv4 Microblock Processing Nexthop Information



24.10 Performance Analysis

The IPv4 Forwarder runs in parallel on multiple microengines for different data rates (OC12 to OC192). In this section we provide an analysis for running IPv4 microblock at OC48 data rates using 4 MEs of IXP2400 @ 600 MHz. The I/O latency available for this packet processing stage to handle a POS minimum packet is $97 * 8 * 4$ cycles.

The worst case is for a packet with proper IP header with a host specific destination address—longest prefix match is 32 bits.

Table 24-14 and Table 24-15 summarize the performance analysis for the IPv4 Forwarder microblock for OC48 data rates running on 4 MEs in IXP2400 @ 600 MHz

Table 24-14. IPv4 Forwarder Cycle Counts

Cycle Count Analysis	Values
Available cycle count for 49 byte min POS packet	(97 * 4) cycles (assuming 600 MHz IXP2400)
Estimated cycle count for min packet based on flow chart	~250 cycles (assuming 110 cycles for the LPM)

Table 24-15. IPv4 Forwarder I/O Latency Analysis for Minimum Packet

I/O latency analysis for min packet	Values
Available I/O latency for min packet	4*97*8 cycles
Read packet header from DRAM	32 bytes
Trie5 lookup (long)	3-5 SRAM accesses each 4 bytes
Read next hop info from DRAM (16bytes)	16 bytes
Write packet header to DRAM (32* bytes)	32 bytes
Scratch read	16 bytes
Scratch write	12 bytes

24.10.1 Characterization Data

Table 24-16. IPv4 Forwarder Microblock Characterization Data

Data	Value
General:	
Microblock Name	ipv4_fwder
Microblock Version Number	1
Implementation Language	microcode
Configuration Options use to gather this set of data	1. MICROENGINE
	2. MICROCODE
	3. CHIP_VERSION=IXP2XXX
	4. RFC1812_SHOULD
	5. META_CACHE_SIZE=5
	6. IP_HDR_OFFSET=2
	7. RFC2644_CHECKS
	8. NEXTHOP_INFO_SRAM
	9. PROCESS_CONTROL_BLOCK
	10.IPV4_COUNTERS

Table 24-16. IPv4 Forwarder Microblock Characterization Data (Continued)

Data	Value
Measurement Environment (tool settings)	-
Performance: (cycle counts, latencies, bandwidths)	
Instruction cycle count per packet (common case packet / worst case cycle count)	271
Common-case packet/path assumptions to be documented here	Assumptions
	<ul style="list-style-type: none"> • The header is cached in transfer registers. • Packet is forwarded in common case. • Worst-case scenario for lookup is 32bit lookup with both hi64k and hi256 tables have to be traversed. • Only 1 SRAM read for Dbcast checks. Since this is a linked list, worst case is non-deterministic. The count is for comparing 1 entry. • RFC1812 MUST, SHOULD cases, RFC2644 checks enabled. • PPP Header is 2 bytes, IP HDR offset is fixed at 2 bytes. • All numbers reported are for worst-case normal path.
Scratch Memory	
# of longwords read (for bandwidth calculations)	0
# of longwords written (for bandwidth calculations)	0
# and type of each atomic operation performed (for bandwidth calculations)	2
SRAM	
# of longwords read	DBCast table (configurable): 8
	Trie Table: 8
# of longwords written	Control Block: 1
	NH information: 4
# of longwords written	0
# and type of each atomic operation performed (for bandwidth calculations)	0
DRAM	
# of quadwords read	0
# of quadwords written	0
Other I/O instructions (i.e. MSF, PCI, CAP, and Hash)	0
List of dependent I/O accesses in the longest latency path	DBCast table ->
	Trie Table ->
	Control Block ->
	NH information
Per-Microengine Resources:	

Table 24-16. IPv4 Forwarder Microblock Characterization Data (Continued)

Data	Value
Control-Store Usage (# of instructions used)	383
Local Memory Footprint (# of long words used)	128 LW if LM_DBCAST_TABLE switch is enabled
Local Memory Configuration (shared, or per-context pointer)	Shared. Each context points to this area.
Local Memory - # of LM pointers used	1
GPR Usage – minimum, static usage (absolutes, static, globals)	-
Transfer Reg. Usage – minimum, static usage	-
Next Neighbor Reg. Usage – minimum, static usage	0
Signal Usage – minimum, static usage	-
CAM used? (yes or no)	No

Global Resources:

Scratch footprint (# of longwords used) – constant or formula ...	16 LW * Number of Ports - used to maintain statistics.
SRAM footprint (# of longwords used) – constant or formula ...	1.4Mbytes for Mae-West route table 8Kbytes for Directed Broadcast Table
DRAM footprint (# of quadwords used) – constant or formula ...	0
Q-Array usage - # of queues used and if they need to be cached	0
CRC Unit used?	no
Hash Unit used? (yes or no)	no

Other Information:

Critical Section Length (compute cycles + memory accesses)	0
# of phases	1
Packet Metadata - fields read	packetSize inputPort nhIdType nextHopId nexthopId
Packet Metadata - fields written	Output Port Fabric Port nexthop ID type
Header - fields read	Typically depends on application dispatch loop. Minimum, IPV4 Header must be read.
Header - fields written	Typically depends on application dispatch loop. Updates TTL and checksum fields.

Table 24-16. IPv4 Forwarder Microblock Characterization Data (Continued)

Data	Value
Entry and Exit setup (assumptions on ME context)	Packet header must be available (ie., must be cached). If LM_DBCAST_TABLE option is used, then succeeding blocks should reset LM index appropriately.
# of MEs required to run a single instance of Microblock	1
Data Rates	Upto OC192
Performance considerations- any issues that have a direct impact on performance	1) Allocate multiple MEs to multiple different clusters to balance I/O 2) Dbcast table may be moved to LM to save SRAM BW
Documentation:	
Thread Ordering Requirements	Within this block, threads may go out of order. It is the responsibility of dispatch loop to ensure order.
OS dependencies	VxWorks, Linux
Chip/hardware dependencies (i.e. Crypto Unit in 2850)	IXP2400, IXP2800, IXP2850
Tested on which SDK Release(s)	-
Tested on hardware? Which hardware configuration?	OC48 POS, 4GB Ethernet, OC192 POS, 10x1Gb Ethernet, 1x10Gb Ethernet
Tested in which applications (not an all inclusive list)	several SDK applications (IPV4, MPLS, ATM)
Possible Configuration Options	1. MICROENGINE 2. MICROCODE 3. CHIP_VERSION=IXP2XXX 4. RFC1812_SHOULD 5. META_CACHE_SIZE=5 6. IP_HDR_OFFSET=2 7. RFC2644_CHECKS 8. NEXTHOP_INFO_SRAM 9. PROCESS_CONTROL_BLOCK 10.LM_DBCAST_TABLE 11.IP_HDR_CACHE_LM 12.FLUSH_EXCP_PKT_HDR 13.COMPRESSED_NEXTHOP_INFO 14.IPV4_COUNTERS
Major limitations (i.e. Rx/Tx microblocks written to use entire ME)	High latency due to numerous dependent read operations
Packet Sequencing Issues (esp. in POT applications)	None
Core Component or Interface requirements or dependencies	IPV4 CC, RTMV4

XX

25.1 Overview

The IPv6 Forwarder microblock on the Ingress IXP2400 validates the IPv6 header and performs unicast forwarding on incoming packets. The block is run in a functional pipeline on four microengines (preferably with two microengines per cluster to balance the usage of the command bus and the S-Push/Pull buses). Each thread executing this block handles one packet at a time. The threads execute in strict order and the ordering is maintained by the `dl_source` and `dl_sink` blocks in the dispatch loop. This ensures that packet ordering is maintained.

The microblock performs header checks as specified in [RFC2460] and [RFC 2373]. The specific checks performed are described in [Section 25.5, “RFC Compliance” on page 427](#). The functionality of the microblock is described in the high level flow chart in [Section 25.9, “High Level Microblock Flow Charts” on page 443](#).

25.2 Assumptions

The following assumptions are made in the design and implementation of the IPv6 Forwarder microblock:

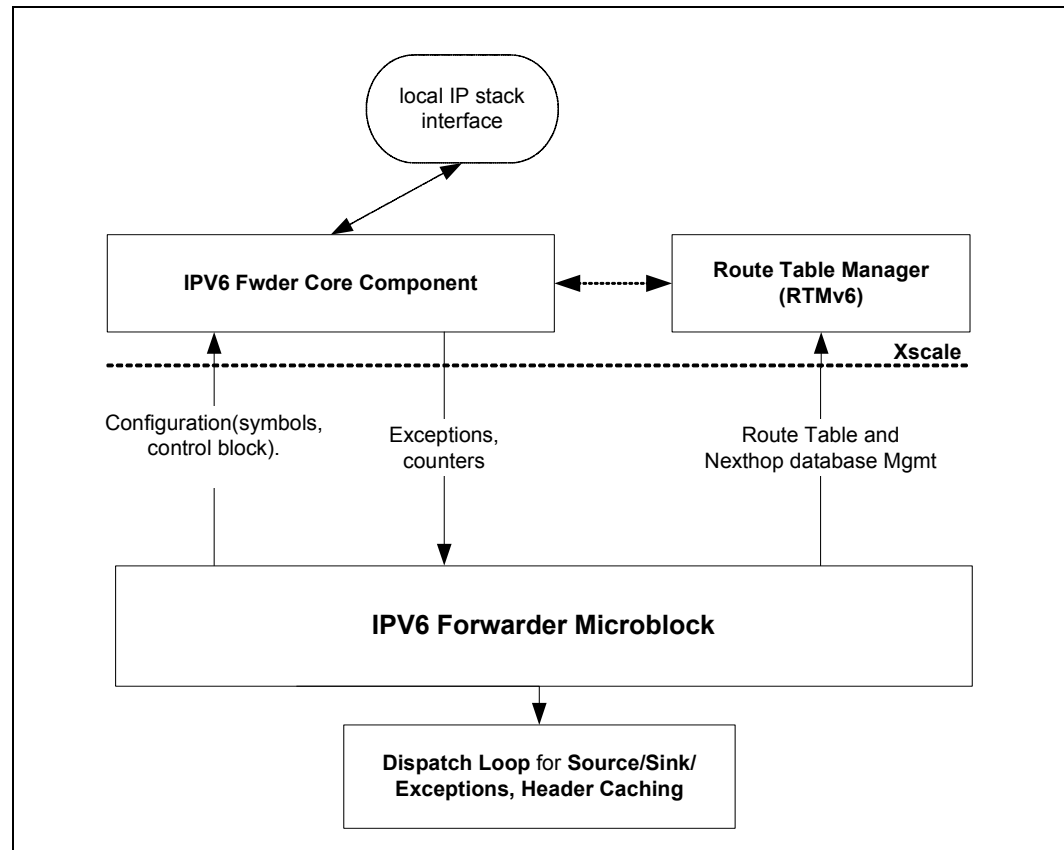
- The primary goal of this microblock is to achieve maximum performance. Hence the following assumptions apply
 - The microblock does not perform any error checking on the data structures shared with the Intel XScale® core code. The Intel XScale® core code has to populate the structures with correct values. Otherwise, the results are unpredictable.
 - The microblock tries to take advantage of the layout of the above structures (and the member fields inside). Hence, the Intel XScale® core code cannot assume to re-use the reserved fields for some other purpose without affecting the microblock functionality.
 - A trade-off is evaluated between efficiency of the microblock and the degree of details provided in the exception codes when the packet has to be sent to the Intel XScale® core.
- All reserved fields in the data structures must be cleared to zero, unless otherwise specified.
- This block is an IPv6 Unicast Forwarder.
- If the source address of a packet is a multicast address, the packet is dropped.
- If the destination address of a packet is a multicast address, the packet is sent to the Intel XScale® core for exception processing.
- The threads executing this microblock does not maintain strict ordering (because of the different amounts of processing and I/O required per packet). As a consequence, the packets can get out of order by the end of microblock. Hence, it is the responsibility of the dispatch loop using this microblock to ensure packet order around this block.
- The packet ordering is not maintained when there are exception packets, as these packet are sent to the Intel XScale® core and the microblock (or the dispatch loop) has no way of ensuring the packet ordering on the Intel XScale® core.

- The current implementation of the microblock heavily uses the xbuf macros detailed in the *IXA Portability Framework Reference Manual* to access the fields in the IP header. The microblock assumes that these xbuf macros can operate on transfer registers, general purpose registers (GPRs) as well as local memory.
- The Hop-limit should be decremented at each hop. If the hop-limit reaches zero, the packet is sent to the Intel XScale® core for exception processing. The Intel XScale® core sends an ICMPv6 *time exceeded* message to the sender of the packet.
- If the destination is a link-local address, the packet is sent to the Intel XScale® core for processing.
- The microblock always maintains 32-bit counters for performance reasons. To support 64-bit counters, it is recommended that the Intel XScale® core code periodically read these counters and update its local copy.
- The forwarding functionality is disabled on a particular output interface by setting the `IPV6_NH_FLAGS_DROP` flag in all the next-hop entries on this output port.
- The microblock checks size of IP datagram against the MTU field in the next-hop information to determine if fragmentation is needed. If the MTU fields is smaller than the size of the packet, the packet is sent to the Intel XScale® core.
- The microblock always performs the MUST checks specified in RFC2460 and RFC2373 (see [Section 25.5, “RFC Compliance” on page 427](#)). It is assumed that the applications always require these checks. The RFC SHOULD checks may be disabled as a compile-time option.
- The Intel XScale® core code has to create and maintain/update the shared data structures. The microblock simply reads and processes the information. Only counters are an exception, as they are updated by the microblock.
- The microblock doesn't always fill the packet Intel XScale® core data with next-hop information. See [Section 25.9, “High Level Microblock Flow Charts” on page 443](#) for more information. Hence, depending on the phase of the microblock from where the exception originated, the Intel XScale® core code has to obtain this data and fill the metadata structure.
- All address and offsets specified in this section are byte addresses.
- The values in the data structures shared by this block are in network byte order (big-endian).

25.3 Dependencies

The main dependencies of this microblock are shown in [Figure 25-1](#). The inward arrows indicate dependency on the IPV6 Forwarder microblock. The outward arrows show what the microblock expects from other modules.

Figure 25-1. IPV6 Microblock Dependencies



25.4 Configuration Options

25.4.1 Build Switches

[Table 25-1](#) identifies compile time build switches, which can be turned on or off as per the requirements of a specific application.

Table 25-1. Compile Time Build Switches (Sheet 1 of 2)

Symbol	Description
MICROENGINE	Code specific to a microengine
ETHER_RECEIVE	Input Port Type - Ethernet
PPP_RECEIVE	Input Port Type - PPP

Table 25-1. Compile Time Build Switches (Sheet 2 of 2)

Symbol	Description
IPV6_STATS	To enable counters to gather various IPv6 statistics.
RFC2373_CHECKS	Checks for compliance with RFC2373
IPV6_LOADBALANCE	To enable support for up to four-equal cost next-hops.
CHIP_VERSION	Enables IXP library to work on IXP2xxx processors
META_CACHE_SIZE	Defines the number of long words of metadata to be cached
ETHERNET_HDR_SIZE	Specify the size of the Ethernet header as a compile time option. If a size is not specified, 0xE is used as default.
ME_NUMBER	ME in the functional pipeline on which the microblock runs.
DL_NEXT_ME	Next ME in the functional pipeline w.r.t the ME in question. Needed by dispatch loop.
DONT_HASH_LOWER_64_BITS_OF_DIP	If it is defined perform a hash on the lower 64 bits of the destination address during LPM lookup.
SCHEDULER_ME	The ME on which the scheduler runs. The microblock that transmits mpackets uses this value to give the ME a count of the number of packets that were transmitted.
Q_FLUSH_LOGIC	The flush logic to be used by the Q-Manager.
QOS_ME_0, QOS_ME_1	Used for SRAM Q-Array allocation.

25.4.2 Default Configuration

The build switches with which the microblock is released are as follows:

- MICROENGINE¹
- CHIP_VERSION=IXP2xxx¹
- META_CACHE_SIZE=8
- ME_NUMBER =x, where x is the microengine on which a particular microblock runs.
- DL_NEXT_ME=x, where x is given in Table 25-2. (The ME number in 0xAB format implies the ME 'B' in cluster "A").

Table 25-2. Table of Next ME Numbers

ME	Next ME
0x01	0x02
0x02	0x10
0x10	0x11
0x11	0x01

- RFC2373_CHECKS=checks performed per RFC 2373
- SCHEDULER_ME=0x12 (ME number 2, in cluster 1)
- Q_FLUSH_LOGIC
- QOS_ME_0, QOS_ME_1 defined for an ME on which the queue manager code runs.

1. These switches should never be turned off. Other switches are configurable, depending on the application.

25.5 RFC Compliance

The microblock always performs the [RFC2460] specified 'MUST' header checks. [Table 25-3](#) summarizes these checks.

Table 25-3. RFC2460 “Must” Checks Performed in the Microblock

Serial No.	RFC2460 MUST Check	Action
1	Packet with version!= 6	Drop
2	Packet size reported is less than 40 bytes	Drop
3	Packet with Hop Limit zero (checked after a forwarding decision is made)	Raise an exception. See Section 25.8.2, “Exception Codes” on page 443

The other [RFC2373] recommended checks can be turned on by enabling the `RFC2373_CHECKS` switch. These checks are summarized in [Table 25-4](#).

Table 25-4. RFC2373 *Should* Checks Performed in the Microblock

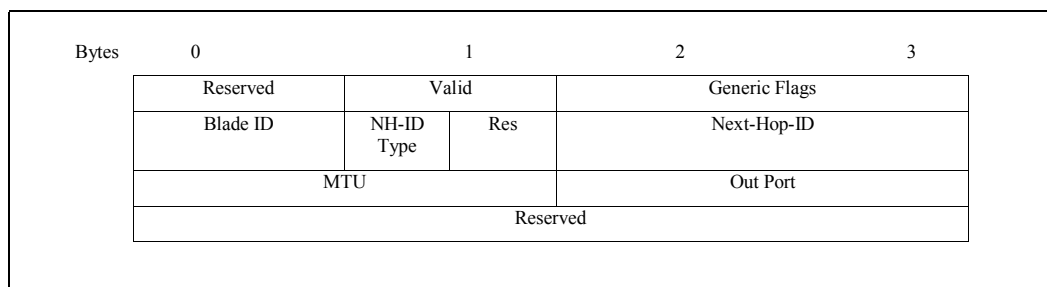
Serial No.	RFC2373 'SHOULD' Check	Action
1	Packet with link-local source address	Drop
2	Packet with multicast source address (the MSB of the source address is 0xFF)	Drop
3	Packet with destination set to 0 or ::1	Drop
4	Packet with multicast destination address (the MSB of the destination address is 0xFF)	Raise an exception. See Section 25.8.2, “Exception Codes” on page 443
5	Packet with source address set to loopback (::1)	Drop
6	Packet with a link local destination address	Raise an exception. See Section 25.8.2, “Exception Codes” on page 443
7	Packet with next header set to Hop-by-Hop	Raise an exception. See Section 25.8.2, “Exception Codes” on page 443

25.6 Data Structures

This section describes the data structures relevant to the IPv6 forwarder microblock. These data structures are shared between various modules as identified. The next-hop tables are split between the ingress and egress side. The L3 next hop information resides in the ingress IXP2400 while the L2 next-hop table resides in the egress IXP2400.

25.6.1 L3 Next Hop Information

[Figure 25-2](#) shows the next hop information associated with each route. The result of route lookup is a next-hop index, which is used to obtain the next hop information. There are two types of next-hop structures—real and intermediate. Intermediate next-hop structures exist to support up to four-equal cost next-hops. [Figure 25-2](#) shows the structure of a *real* next-hop entry.

Figure 25-2. Structure of a *Real* Next-Hop EntryTable 25-5. Fields of a *Real* Next-Hop Entry

LW	Bits	Size	Field	Description
0	7:0	16	Reserved	Reserved
0	15:8	8	Valid	If 1, indicates that this entry is in use.
0	31:16	16	Generic Flags	Flags to indicate action as follows:IPV6_NH_FLAGS_LOCAL: Local DeliveryIPV6_NH_FLAGS_DOWN: Interface is down.IPV6_NH_FLAGS_DROP: Simply drop the packet.IPV6_NH_FLAGS_MULTIPLE: This entry is an intermediate entry.
1	7:0	8	Blade ID	CSIX information to send the packet to the right egress microengine for encap processing
1	15:12	4	NH-ID Type	Next Hop ID Type. The following values are defined:NHID_TYPE_FWRD: Forwarder TypeNHID_TYPE_TUNNEL: Tunneling TypeNHID_TYPE_COMPRESS - Compression Type
1	11:8	4	Reserved	Reserved
1	31:16	16	Next-Hop ID	Next-Hop ID, can be the L2 table next-hop index, Tunneling table index or some other information.
2	15:0	16	MTU	The MTU supported by this outgoing interface.
2	31:16	16	Out Port	Output port to use to transmit the packet.
3	31:0	32	Reserved	Reserved

A brief description of some of the fields is below.

25.6.1.1 Valid

Indicates that the entry at this location is Valid—that is, in use. If it is set, the entry is in use, otherwise it is cleared.

25.6.1.2 Generic Flags

The defined generic flags are listed in [Table 25-6](#). The bit positions are relative to the starting bit position—which is bit zero. All other bits are reserved.

Table 25-6. Generic Flags

Bit Position	Global name	Description
0	IPV6_NH_FLAGS_LOCAL	The packet is for local interface. Packet is sent to XScale core.
1	IPV6_NH_FLAGS_DOWN	The interface/next-hop is down. The packet is sent to XScale core.
2	IPV6_NH_FLAGS_DROP	The packet is dropped.
3	IPV6_NH_FLAGS_MULTIPLE	The entry is an intermediate entry. See Section 25.6.2.2 .
4-15		Reserved

25.6.1.3 Blade ID

The blade ID is used to determine to which blade on the fabric the packet is to be sent to.

25.6.1.4 Next-Hop ID Type

The next-hop ID type specifies the type of the entry. The IPv6 forwarder performs different actions based on the ND-ID Type field. The bit positions for this field are defined in [Table 25-7](#).

Table 25-7. Next-Hop ID Types

Bit Position	Global name	Description
0	NHID_TYPE_FWRD	Forwarder entry.
1	NHID_TYPE_TUNNEL	Tunneling entry.
2	NHID_TYPE_COMPRESS	Compression entry.
3		Reserved

25.6.1.5 Next-Hop ID

The Next Hop ID is passed over the CSIX fabric to an Egress IXP2400 where it is mapped to a layer-2 header to be prepended to the buffer. The Next Hop ID can carry the L2 table index, tunneling table index or some other information.

The Next Hop ID has an invalid value, which is -1 (all bits set).

When the Next Hop ID is -1, the microblock sends the packet to the core as exception packet (No Route exists). See [Section 25.8.2, “Exception Codes” on page 443](#).

Since the unused entries in the route table are initialized to zero, the microblock picks up zero as next-hop index when there is no matching route. Instead of checking next-hop index for zero, the microblock goes ahead and reads the corresponding next-hop information. This next-hop entry is reserved to store default route information. In such a case, the microblock automatically selects the default route (if one is installed) when a route lookup fails. If there is no default route, then next-hop ID at index zero must be -1.

25.6.1.6 Output Port

The output port number is carried over to the egress IXP2400 where it identifies the port on which the packet must be transmitted. The application may choose to interpret this field as the physical or logical port number. In this application, the microblock uses this field to check for redirection of the packet.

25.6.1.7 MTU

The maximum transmission unit is used by the microblock to check if size of the IP datagram is more than the MTU. If so, the packet is sent to the XScale core.

25.6.2 Intermediate Next Hop Information

The IPv6 forwarder can support up to four-equal cost next-hops. To support this requirement, the next-hop table defines an “intermediate” next-hop entry. The intermediate next-hop entry is an entry that stores the location of up to four-equal cost “real” next-hop entries. The structure of a real next-hop entry is described in [Section 25.6.1, “L3 Next Hop Information” on page 427](#).

[Figure 25-3](#) shows the structure of an *intermediate* next-hop entry.

Figure 25-3. Structure of an Intermediate Next-Hop Entry

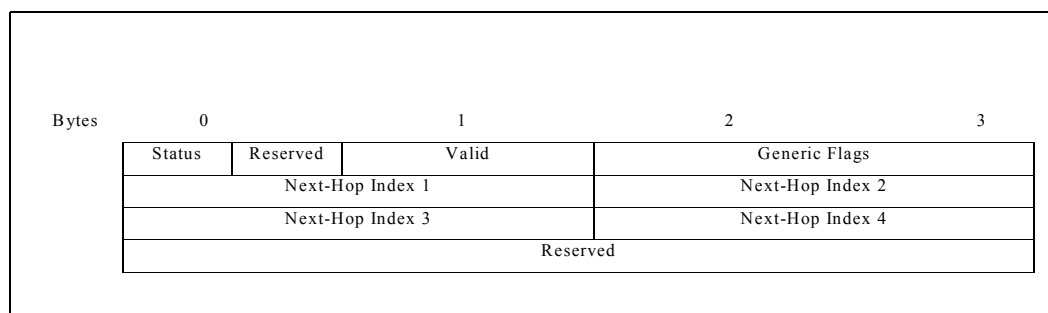


Table 25-8. Fields of an Intermediate Next-Hop Entry

LW	Bits	Size	Field	Description
0	3:0	4	Status	A four bit field indicating the link status (UP/DOWN) of the 4 equal cost next hop interfaces. 1 indicates that the interface is UP, 0 indicates that the interface is DOWN.
0	7:4	4	Reserved	Reserved
0	15:8	8	Valid	If 1, indicates that this entry is in use.
0	31:16	16	Generic Flags	This type of an entry has only the following flag set: IPV6_NH_FLAGS_MULTIPLE: Indicates that this entry is an “intermediate” entry with up to 4-equal cost next hops embedded within it.
1	15:0	16	Next-Hop Index 1	L3 Next Hop Index of the first equal cost entry
1	31:16	16	Next-Hop Index 2	L3 Next Hop Index of the second equal cost entry

Table 25-8. Fields of an Intermediate Next-Hop Entry

LW	Bits	Size	Field	Description
2	15:0	16	Next-Hop Index 3	L3 Next Hop Index of the third equal cost entry
2	31:16	16	Next-Hop Index 4	L3 Next Hop Index of the fourth equal cost entry
3	31:0	32	Reserved	Reserved

A brief description of some of the fields follows.

25.6.2.1 Status

A four bit field indicating the status (UP/DOWN) of each one of the four-equal cost next-hops stored in this entry. This field has been added for performance reason. The lookup algorithm (explained later) shows how this field is used.

25.6.2.2 Generic Flags

This is similar to the generic flags in a “real” entry, but the only value that can be assigned to this entry is `IPV6_NH_FLAGS_MULTIPLE`.

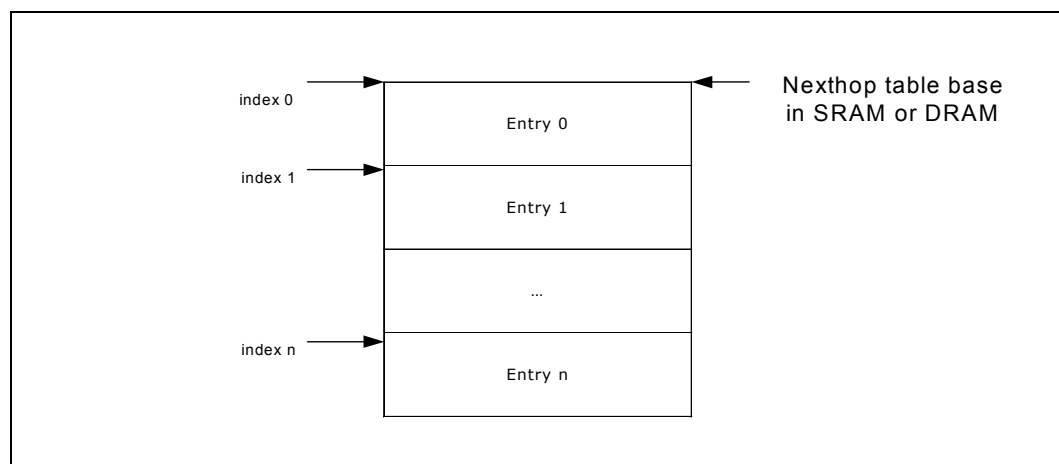
25.6.2.3 Next-Hop Index1-4

The next-hop index array consists of indexes of up to four-equal cost “real” entries. See the lookup algorithm on how this array is used.

25.6.3 Next-Hop Database

The next-hop database is used to store next-hop information corresponding to a route. This database is essentially an array of next-hop information entries, with each entry indexed by the next-hop index obtained via the route lookup (LPM lookup).

The next-hop database can be in SRAM or DRAM. Each entry (entry0, entry1 etc.) is a L3 next-hop structure. The layout of the database is shown in [Figure 25-4](#).

Figure 25-4. Next-Hop Database Layout


25.6.4 IPv6 Counters

The IPv6 Forwarder maintains a number of per-interface statistics counters. These counters are stored in SRAM. Two kinds of counts are maintained, counts for incoming packets and counts for packets that are sent out by the Forwarder. The incoming counts are maintained for the packets received on the interfaces on the blade on which the Forwarder is running. As a result of the forwarding, packets might be sent on to other blades. The forwarder therefore keeps outgoing counts for all blades.

Storage for the counters is maintained in a hierarchical fashion. The counters for a particular interface are stored contiguously, thereby forming a statistics block for that interface. The statistics blocks for the different interfaces are stored one after the other in increasing order of the interface number. The statistics for the set of interfaces on a blade together form a statistics group for that blade. Finally, the blade statistics are stored one after the other in increasing order of blade identification number. Note that incoming counters only need to be stored for the blade on which the forwarder runs. Therefore we store these counters at the start of the memory area set aside for IPv6 counters. This is followed by the outgoing statistics for all blades. (For example, to get an outgoing counter of the third interface on the second blade, we index into the second group from the base of the outgoing counters, from there into the third block of the group and finally to the specific offset for the counter in the block.) The offset of a particular counter in a block is fixed. Thus, knowing the blade id, interface number and the specific statistics counter, one can index to the appropriate memory location. [Table 25-9](#) and [Table 25-10](#) list the offsets of the counters in a specific block.

Also note that since outgoing counters are maintained in the SRAM on all the blades, the Forwarder core component collects the counter values from all the blades before presenting the statistics of a particular blade up to higher level applications.

While some counters need to be maintained by the microblock, many are maintained by the XScale core component. For example, each time there is an error, the core is informed and so it can take care of maintaining the relevant error counters. This is done to minimize the time spent in updating counters in the fast path.

The microblock always maintains 32-bit counters. Code running on the XScale core can keep 64-bit counters if required (see [Section 2.5, “Statistics and Handling of 64-bit Counters”](#) on page 63).

Table 25-9. Counter Offsets in Incoming Statistics Block

Counter	Meaning	Maintained By M=Microblock C=Core	Offset from Base
InReceives	Total packets received (including error)	M	0
InOctets	Total bytes received in IP packets (includes those received in error)	M	4
InForwDatagrams	Count of non local packets for which a route was searched	M	8
InAddrErrors	Count of packets discarded because the destination address was in error [e.g.::0, unsupported (unallocated) address]	M	12
InTruncated	Count of IP packets discarded because the packet didn't carry enough data	M	16

Table 25-9. Counter Offsets in Incoming Statistics Block (Continued)

Counter	Meaning	Maintained By M=Microblock C=Core	Offset from Base
InHdrErrors	Number of packets in error(version number mismatch, other format errors, hop count exceeded, errors discovered in processing their IP options, etc)	M+C	20
CoreInReceives	64-bit counter maintained by the core corresponding to the counter InReceives	C	24
CoreInOctets	64-bit counter maintained by the core corresponding to the counter InOctets	C	32

Table 25-10. Counter Offsets in Outgoing Statistics Block

Counter	Meaning	Maintained By M=Microblock C=Core	Offset from Base
OutForwDatagrams	Count of IP packets for which a path to the destination was found(Increment count on outgoing interface)	M	0
OutTransmits	Count of forwarded packets supplied to lower layer	M	4
OutOctets	Number of bytes in IP packets delivered to lower layers for transmission	M	8
CoreOutTransmits	64-bit count of packets (local + forwarded) supplied to lower layer.The forwarded packet count is got from OutTransmits	C	12
CoreOutOctets	64-bit count of bytes in IP packets delivered to lower layer. The forwarded octet count is got from OutOctets	C	20

25.6.5 Route Table

The route table is described in detail in the [Section 25.7, “Longest Prefix Match Lookup” on page 433](#). The microblock depends on the Route Table Manager on the XScale core to create and manage the route table.

25.7 Longest Prefix Match Lookup

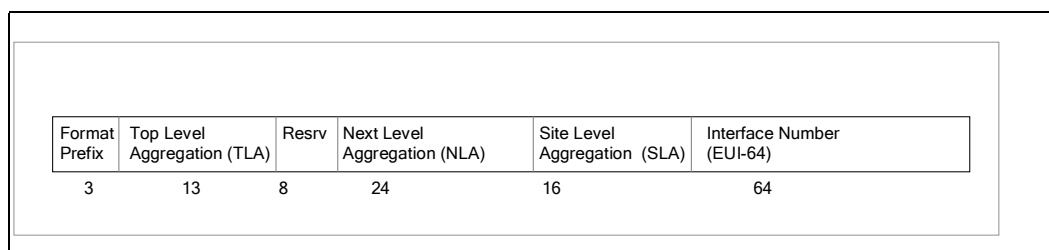
25.7.1 Introduction

The longest prefix match (LPM) algorithm is a key component of the microblock and deserves a detailed description. Even though the IPv6 addresses are structured to facilitate a hierarchy or aggregation of address space, such a hierarchy may or may not be used in practice. The IPv6 lookup scheme therefore has two paths—Normal Lookup and Optimized Lookup.

The Normal Lookup provides an efficient implementation of LPM without making any assumptions about the hierarchy. In this implementation, the route-entries are stored in a tree structure in SRAM, and the LPM is performed by traversing the tree.

The Optimized Lookup takes advantage of the hierarchical addressing to speed up the performance of the Normal Lookup. Optimized Lookup works even when the hierarchy is not present, but has no advantage in those situations. The optimizations are implemented using hardware features like Local Memory and the Hash unit. Figure 25-5 shows the structure of an IPv6 address.

Figure 25-5. Structure of an IPv6 Address



25.7.2 Data Structures

The Normal lookup is implemented using Trie-blocks. A route-entry is divided into portions, and each portion is represented by a trie-block. A trie-block is an array of trie-entries. Each trie-entry is 32-bits in size, and it can contain a pointer to next-hop information, a pointer to another trie-block, or both. Figure 25-6 shows the structure of trie-blocks and trie-entries. The trie-blocks are linked together to create a multi-way tree structure. This tree is traversed during the lookup to find the longest matching route-entry.

This implementation uses a 16-bit trie-block (65,536 entries) for the root node of the tree, and 8-bit trie-blocks (256 entries) for nodes in the next levels of the tree. The tradeoffs associated with the size of trie-blocks and reasons choosing these particular trie-block sizes are covered in detail in the IPv6 forwarder design document. Figure 25-7 shows how the trie-blocks are linked to form a tree structure.

Figure 25-6. Route Table Data Structures—Trie-Blocks and Trie-Entries

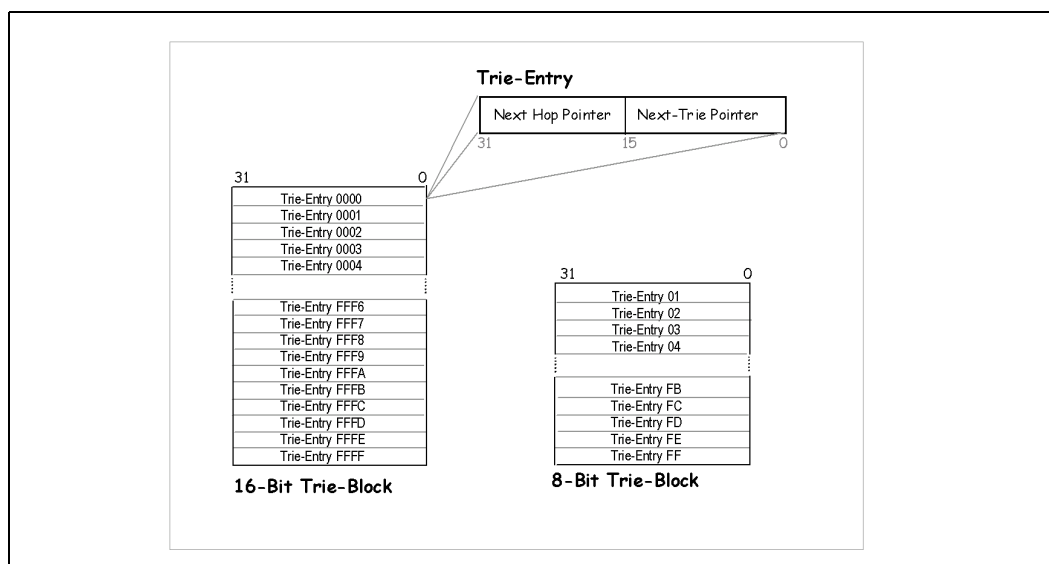
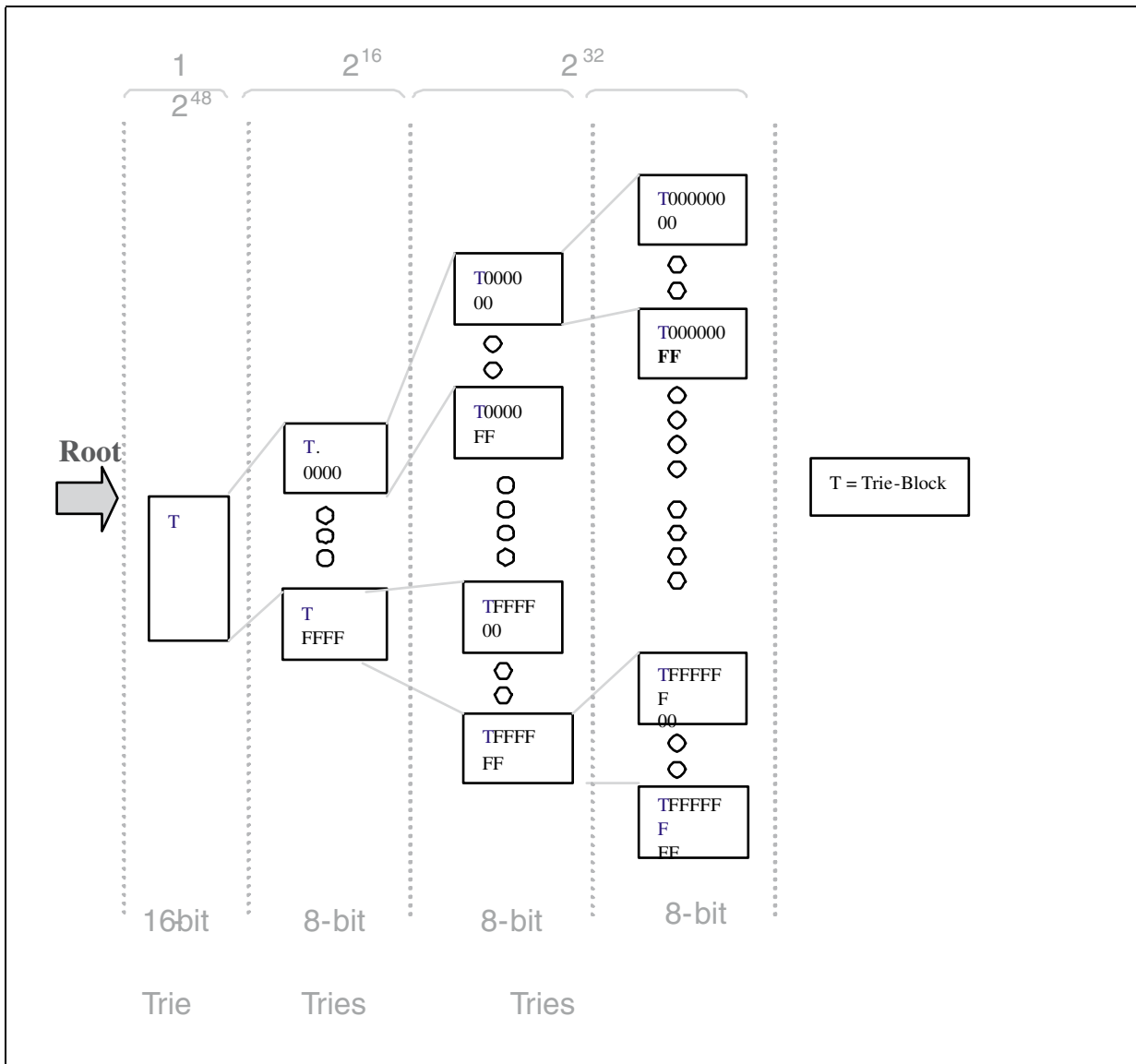


Figure 25-7. Route Table Data Structures—Multi-Way Tree which Trie-Blocks



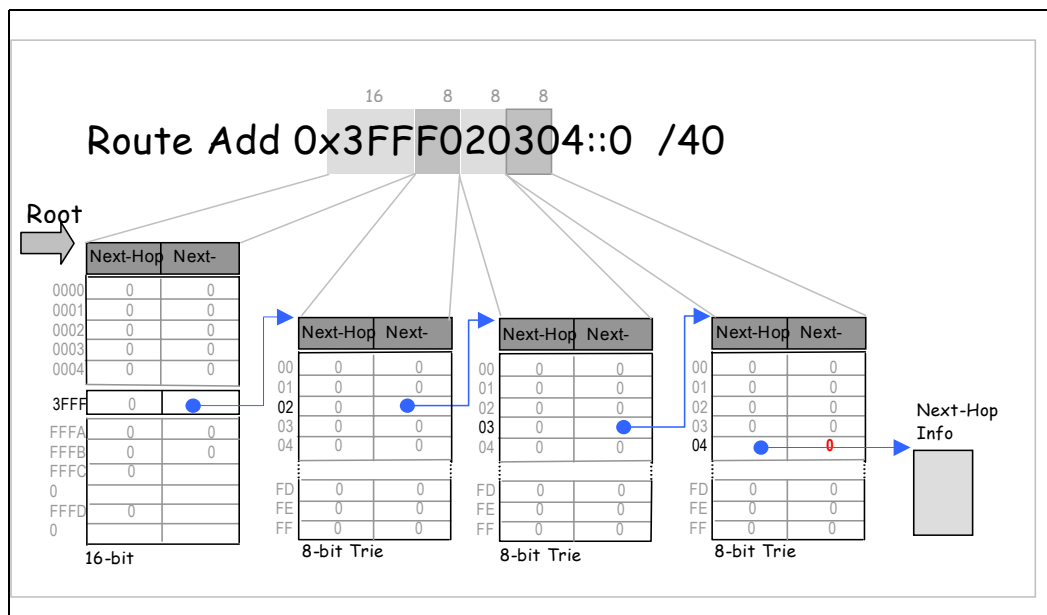
25.7.2.1 Creating the Route Table

Trie-entries are stored in the route table by the RouteAdd routine. The RouteAdd routine is implemented on the XScale core in the following way. For each route entry, the first 16 bits (MSBs) are extracted and used as a direct index to set a trie-entry in the root 16-bit trie-block. The remaining bits of the route-entry, if any, are mapped to 8-bit trie-blocks - 8 bits at a time. The trie-blocks are linked by setting the next-trie pointers in appropriate trie-entries. The next-hop pointer in the last trie-entry in the chain contains the next-hop-id (L3 table index) associated with that route-entry.

25.7.2.2 Route Add Example

Figure 25-8 shows how a 40 bit route-entry 0x3FFE020304::0 /40 and its associated next-hop-id is stored in the trie structures.

Figure 25-8. Representation of a 40-Bit Route Entry 0x3fff020304 Using Trie-Blocks



25.7.3 Route Lookup using Longest Prefix Match (LPM)

The 128-bit destination IPv6 address is extracted from the packet header and the first 16 bits (MSBs) are used to index into the Root 16-bit trie-block. Subsequent trie-blocks, if any, are traversed using 8 bit portions of the address until a null next-trie pointer is encountered. The last valid next-hop pointer encountered during the traversal is returned as the result of the search. The pseudo code for the route lookup algorithm is shown in Figure 25-9.

25.7.4 Lookup Pseudo Code

Figure 25-9 shows the pseudo-code of the basic lookup algorithm. In the actual implementation, the while loop is completely unrolled to improve the efficiency of the lookup. The resulting lookup scheme can match a typical 64-bit long prefix using a maximum of seven SRAM accesses. This lookup scheme scales very well and its performance does not deteriorate with an increase in the number of route-entries. This scheme also allows simultaneous execution of route updates from Intel XScale® core core without disrupting the lookups.

Figure 25-9. Pseudocode for the Basic Lookup Algorithm.

```

Route Lookup (IN DestinationIPv6Address, OUT next-hop-info)

    LongestMatchSoFar = DefaultRouteNextHopIndex
    Index = First 16 bits of the DestinationIPv6Address
    Current-trie-block = Root-trie-block
    While (Current-trie-block not NULL)
    Current-trie-entry = Current-trie-block [Index]
        If (Current-trie-entry.next-hop not NULL)
            LongestMatchSoFar = Current-trie-entry.next-hop
        endif
        Index = next 8 bits of the DestinationIPAddress
        Current-trie-block = Current-trie-block.next-trie
    End while
    next-hop-info = LongestMatchSoFar
End

```

25.7.4.1 Optimized Route Lookup Algorithm

The Optimized Lookup takes advantage of the hierarchical addressing to accelerate the Normal Lookup. These optimizations are based on the following observations:

- When the hierarchical addressing is used, the longest-prefix route-entries share a common prefix. In the case of aggregable IPv6 addresses, all packets destined to outside of the network is aggregated. Therefore, the traffic requiring more specific routing information are packets destined to inside of the network. The packet destined to inside of the network should share a common IPv6 prefix (prefix of the network).
- Each router's interfaces share a common prefix, which is the prefix of the network where the router lies. All packets destined to inside of the network should have this prefix same as the router's interfaces addresses.
- The last 64-bits of IPv6 addresses are normally not used for routing. In case, they are used for routing, the routing information is contained within a subnet. In other words, only for traffic going inside the network can the last 64-bits can be used for routing.

At any time in this optimization phase, if the destination address fails to match the router's prefix bits, the algorithm falls back to the normal trie lookup.

In the implementation of route lookup for IPv6 addresses, the optimized lookup has a trie for first the 64 bits of the destination addresses. We should expect that the trie should terminate (get the null next trie) for all paths before we match the first 64 bits. This is true if hierarchical addressing is followed and hence, appropriate aggregation is followed. However, in the case when addressing architecture is random, we can expect to have non-null next trie even after the match of the first 64-bits. In such a case, a hash lookup is performed on the last 64-bits after the match of the first 64-bits. An exception to this is for the path in the trie, that corresponds to the router's IPv6 prefix. If a destination address matches the first 64 bits of the router's IPv6 prefix, then instead of performing a hash lookup, we perform further trie lookups until the end of 128 bits. This is because if a packet is destined to a subnet and the router lies in the same subnet, we may have routing infrastructure within the subnet that used the last 64 bits. Therefore, a simple hash table for last 64 bits may prove impossible to maintain. We have a 64-bit tries lookup, followed by 64-bit hash lookup in our data structure, except for 128 bits tries lookup for the case when the first 64 bits matches the router's IPv6 prefix

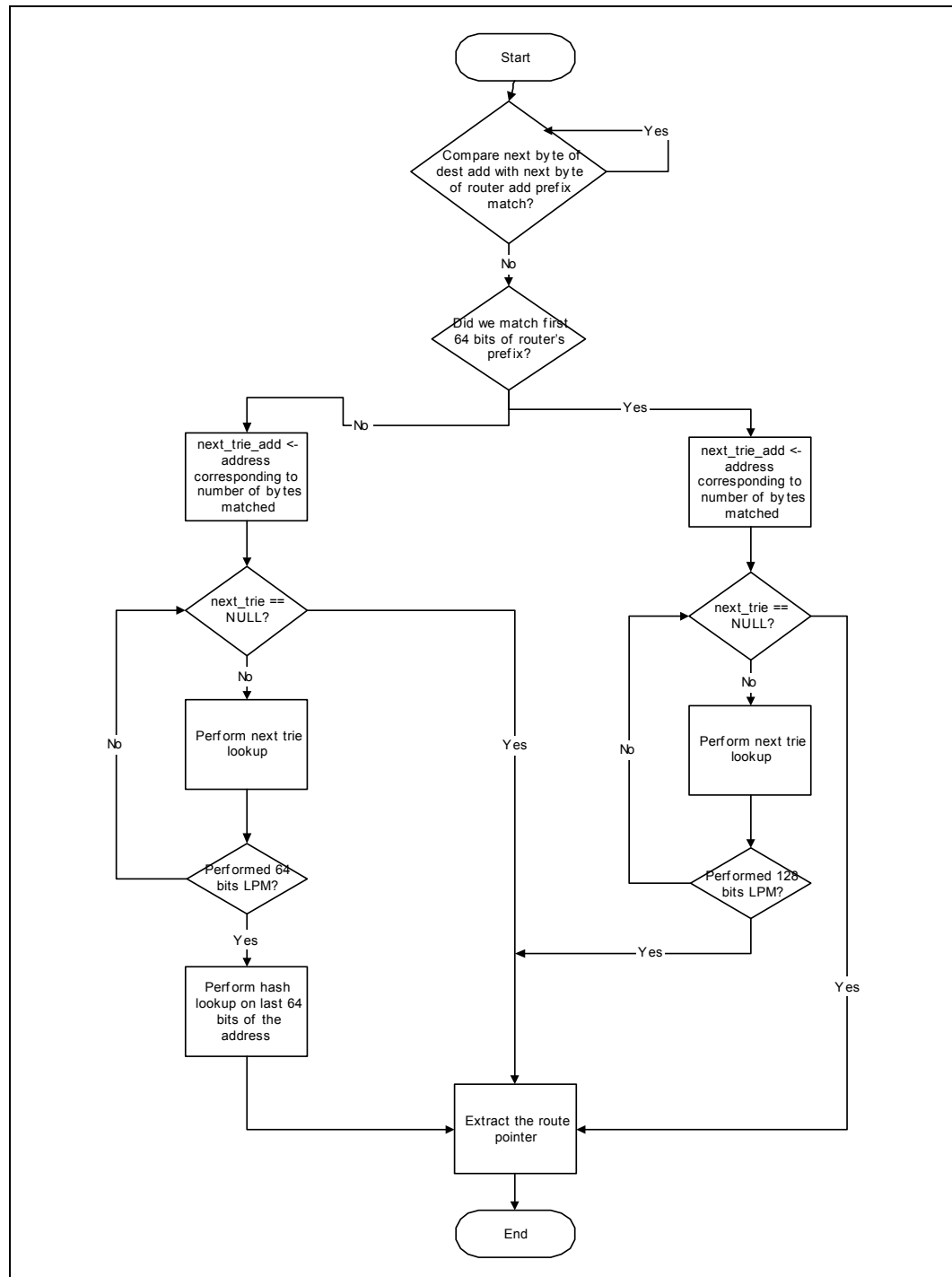
Figure 25-10 shows the flow chart for the lookup algorithm. The algorithm roughly works as follows. For all IPv6 packets, we first match the initial bits of the destination IPv6 address with the locally stored IPv6 prefix of the router. We perform this matching byte by byte¹. For each byte of the destination address that matches the portion of the local IPv6 prefix, we retrieve a data structure from the local memory, which gives us the pointer to the next trie and the pointer to the forwarding entry corresponding to the match until now.

Basically, consider a single trie (called base trie) created for matching up to 64 bits of the destination address. If the destination address matches the first byte of the router prefix, that points into the portion of the base trie that contains the trie for the rest of the bits to be matched. So at each stage, if we have matched x bytes so far ($x < 16$), that match takes us directly to the portion of the base trie that can be used for finding the match for the rest of the bytes. When the match of the destination address fails against the router's IPv6 prefix, we get a pointer to the corresponding data structure and continue with the normal trie lookup.

When we start normal trie lookup, we might be in the 64 bits trie for normal lookup or in 128 bits for specific case lookup. If we are in 64 bits trie, we continue to perform lookup until we get a null next trie or complete full 64 bits lookup. In case, we did not reach a null trie, we perform a hash lookup on last 64-bits. The pointer to the hash table is retrieved from the last trie traversed on the first 64-bits trie lookup. In case, we are into normal lookup in the 128-bits specific case, we continue to perform trie lookup until we reach a null next trie or complete full 128-bits trie lookup.

1. We can change the granularity for matching the destination address with the prefix address.

Figure 25-10. Flow Chart for the Lookup Algorithm



25.7.4.2 Trie-Cache Example

Figure 25-11 shows a snap shot of the trie-blocks and Local Memory data structures after the following steps:

- A 64-bit common prefix 0x3FFA010203040506 is cached.
- A 64-bit route 0x3FFA0102030405FF /64 is added
- A 128-bit specific route 0x3FFA010203040506::ABCD is added.
- A 64-bit route 0x3FFF0102030405FF /64 is added

Figure 25-11. Route Tables with Prefix Optimization

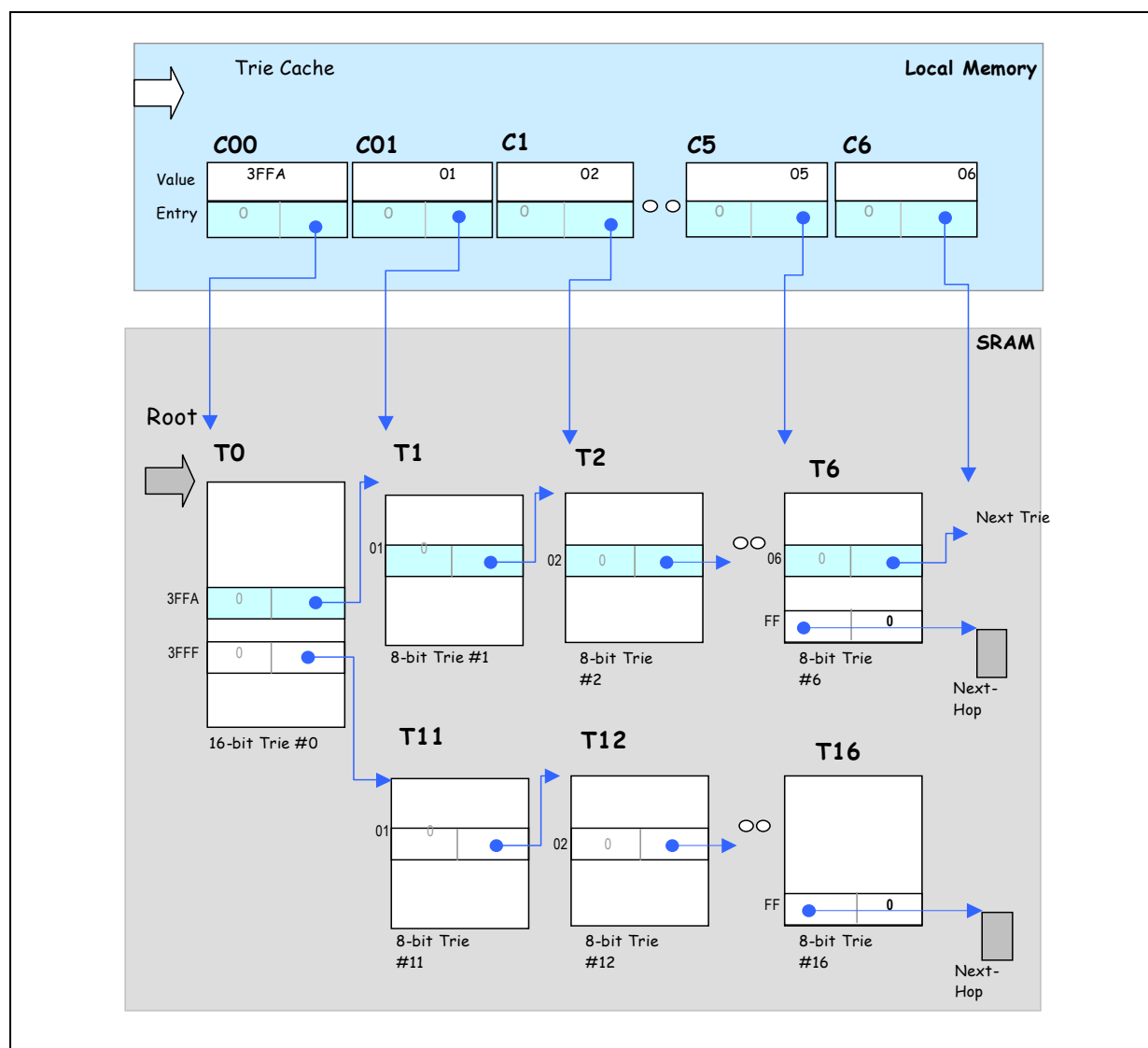


Table 25-11 captures how the LPM is performed for different destination addresses.

Table 25-11. Longest Prefix Match Algorithm Performance

Destination IP	Traversal Path	Length of Matched Prefix	Number of Local Memory Accesses	Number of Trie-Block SRAM Accesses
0x3F::00	C00, T0	NA	1	1
0x3FFA:00	C00,C01,T1	NA	2	1
0x3FFA01:00	C00,C01,C1,C2	NA	3	1
0x3FFA0102030405FF::00	C00,C01,C1,C2,C3,C4,C5,T6	64	7	1
0x3FFA010203040506::abcd	C00-C6, ...	128	8	8
0x3FFF0102030405FF::00 (worst case)	C00, T11-T16	64	1	7

25.7.4.3 Controlled Prefix Expansion

For each forwarding prefix, its prefix length decides the mapping of that address on to the trie-entries in 16-bit or 8-bit trie structures. The first 16 bits of the prefix address are taken and used as direct index to root-trie entries. The remaining bits are mapped to 8-bit trie blocks. If the number of the bits in the prefix does not match the trie tables' boundaries, controlled prefix expansion is used and multiple trie-entries having the same entry key are set in the last trie block. Forwarding entries that have prefix length of less than 16 bits are added to the Base 16-bit trie block directly using controlled prefix expansion.

In this scheme, where the bits to be mapped are less than the bits required for the current stride (for instance when the prefix length is not a multiple of 8), the indices of the trie entries are generated by concatenating the available bits with combinations of remaining bits to make the full stride length.

Figure 25-12 shows the mapping of a route entry FFFE:201::/32 to the trie data structures. Figure 25-13 illustrates how a route entry FFFE:200::/30 is represented by three blocks using controlled prefix expansion.

Figure 25-12. Trie Table Representation for Routing Prefix FFFE:201::/32

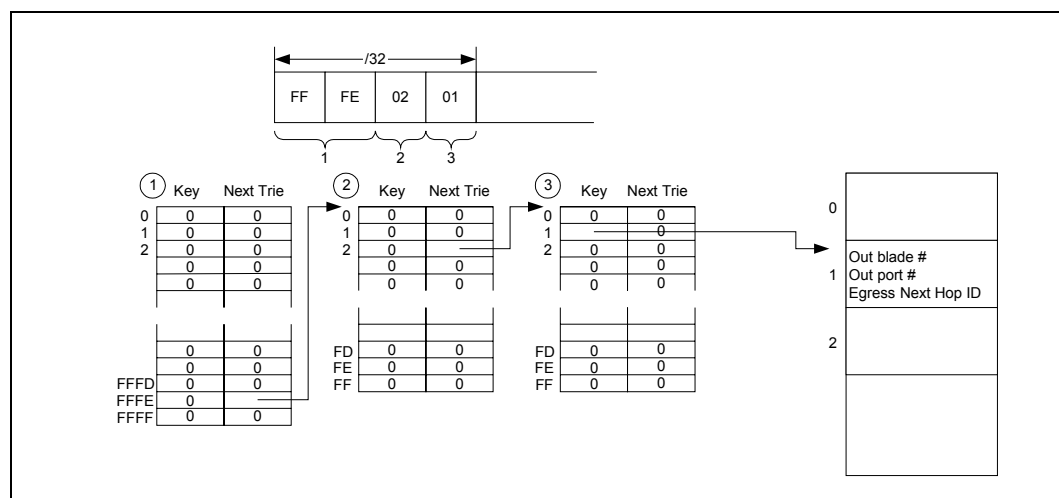
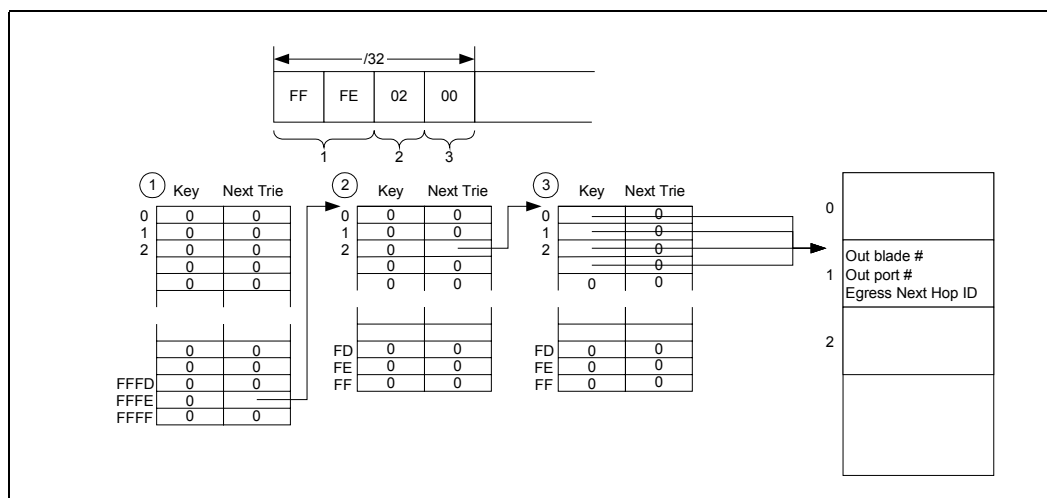


Figure 25-13. Trie Table Representation for Routing Prefix FFFE:200::/32



25.8 Intel XScale® core Interface

25.8.1 Symbols

The following data structures are shared between the data plane and the Intel XScale® core component.

- Statistics Counters
- Route Table (Trie Blocks)
- Next Hop Informations (both the L3 and L2 tables)
- Scratch ring for exception processing.

The information about these data structures are passed to the data plane by dynamically patching the following symbols at load time. In the current implementation, these are defined as compile time values.

Table 25-12. IPV6 Symbols

Symbol	Description
IPV6_STATS_TABLE_BASE	Counters are maintained starting at this address in SRAM.
IPV6_TRIE_TABLE_SRAM_BASE	The Trie tables for LPM start at this address in SRAM.
NEXTHOP_TABLE_SRAM_BASE	The L3 next-hop table starts at this address in SRAM.(This table is shared with the IPv4 microblock)
THIS_BLADE_ID	ID of the blade on which the IPv6 core component is running.

25.8.2 Exception Codes

Table 25-13 identifies the exception codes generated by IPv6 Forwarder block that are passed along with the packet to the Intel XScale® core counterpart.

Table 25-13. IPv6 Exception Codes

Exception code	Value	Description
IPV6_EXCP_BAD_HOPLIMIT	0x42	The IP header with hop count equal to zero.
IPV6_EXCP_MULTICAST	0x43	Packet with multicast address for destination
IPV6_EXCP_LINKLOCAL	0x44	Packet with link-local destination address.
IPV6_EXCP_OPTIONS	0x45	Packet with next-header set to hop-by-hop header (0).
IPV6_EXCP_LKUPFAILED	0x46	Packet destination address lookup failed. This could be because the route is DOWN, LPM lookup on the destination address failed or that the next-hop information indicates that no route exists for a next-hop index.
IPV6_EXCP_LOCAL	0x47	This packet is destined to the local node.
IPV6_EXCP_MTU	0x48	The packet size is greater than the MTU for the outgoing interface. The packet needs fragmentation.
IPV6_EXCP_REDIRECT	0x49	An ICMP redirect message needs to be sent by the core.

25.9 High Level Microblock Flow Charts

The following flow charts describe the path of IP packet inside the IPV6 Forwarder macro. The flowchart in Figure 25-14 shows the overall microblock functionality while the flowchart in Figure 25-15 shows the header validations performed by the IPv6 forwarder as part of the forwarder functionality. Figure 25-16 shows the flow chart for the macro that processes next-hop information.

Figure 25-14. IPv6 Forwarder Microblock Flowchart

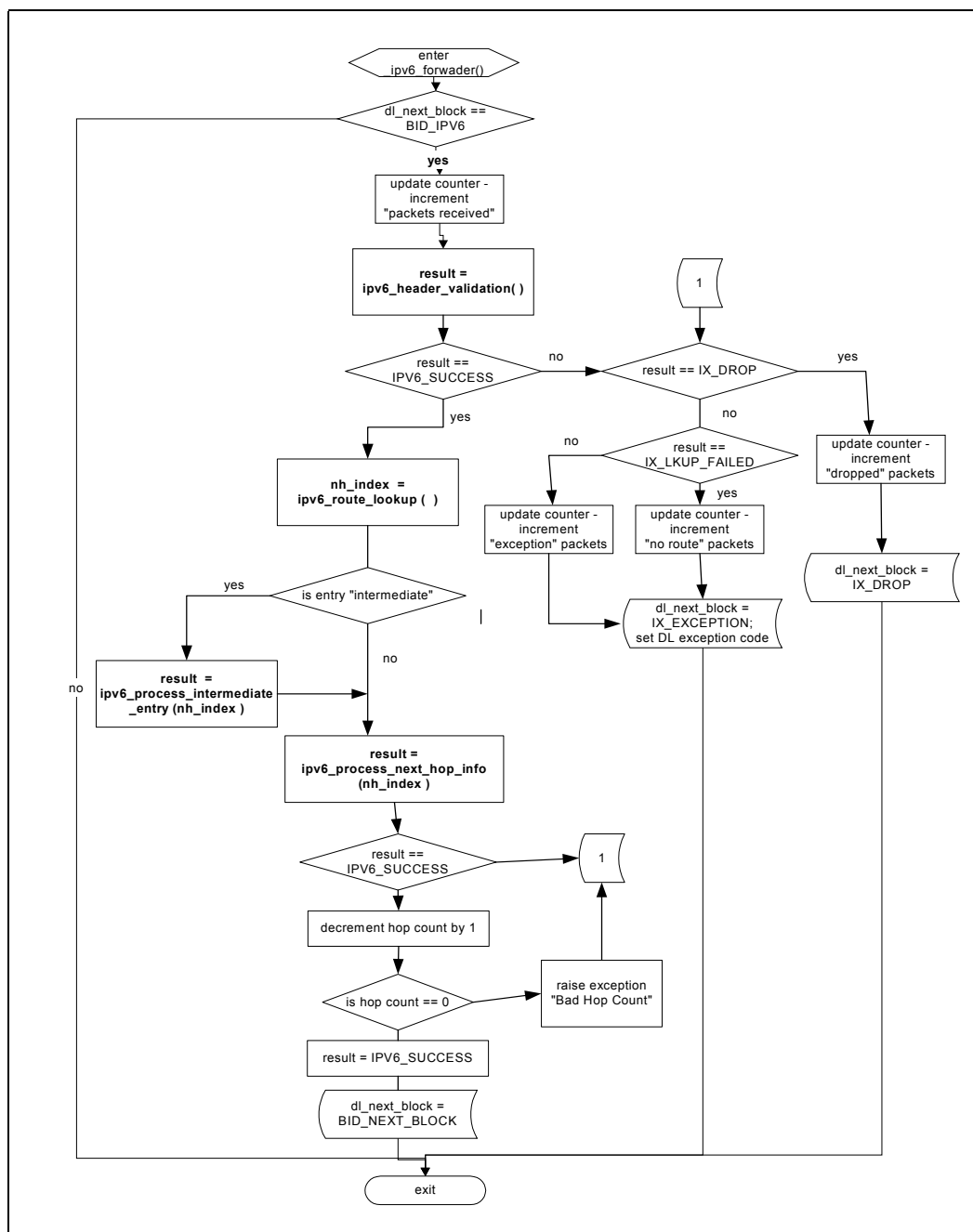


Figure 25-15. IPv6 Header Validation Flowchart

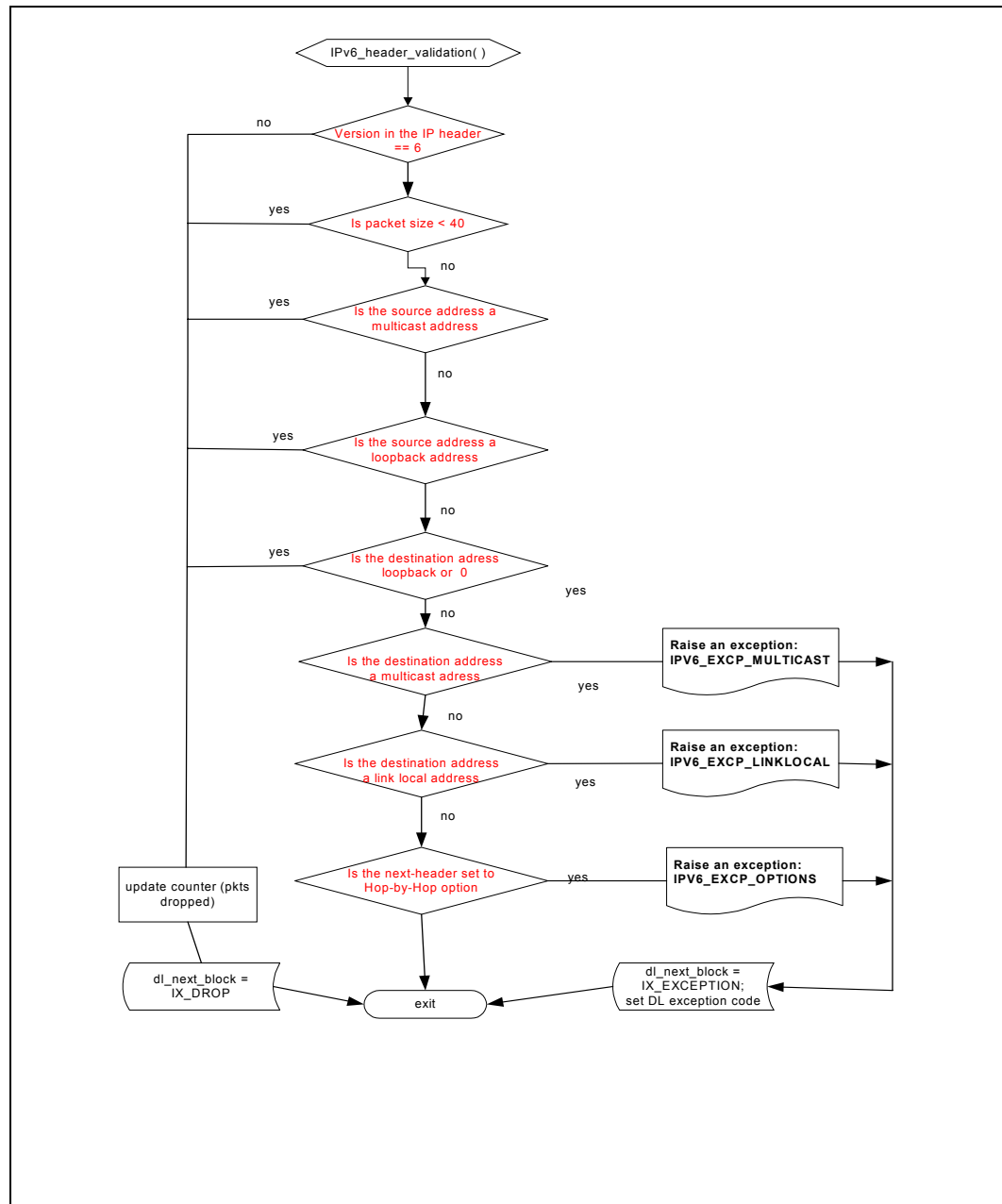
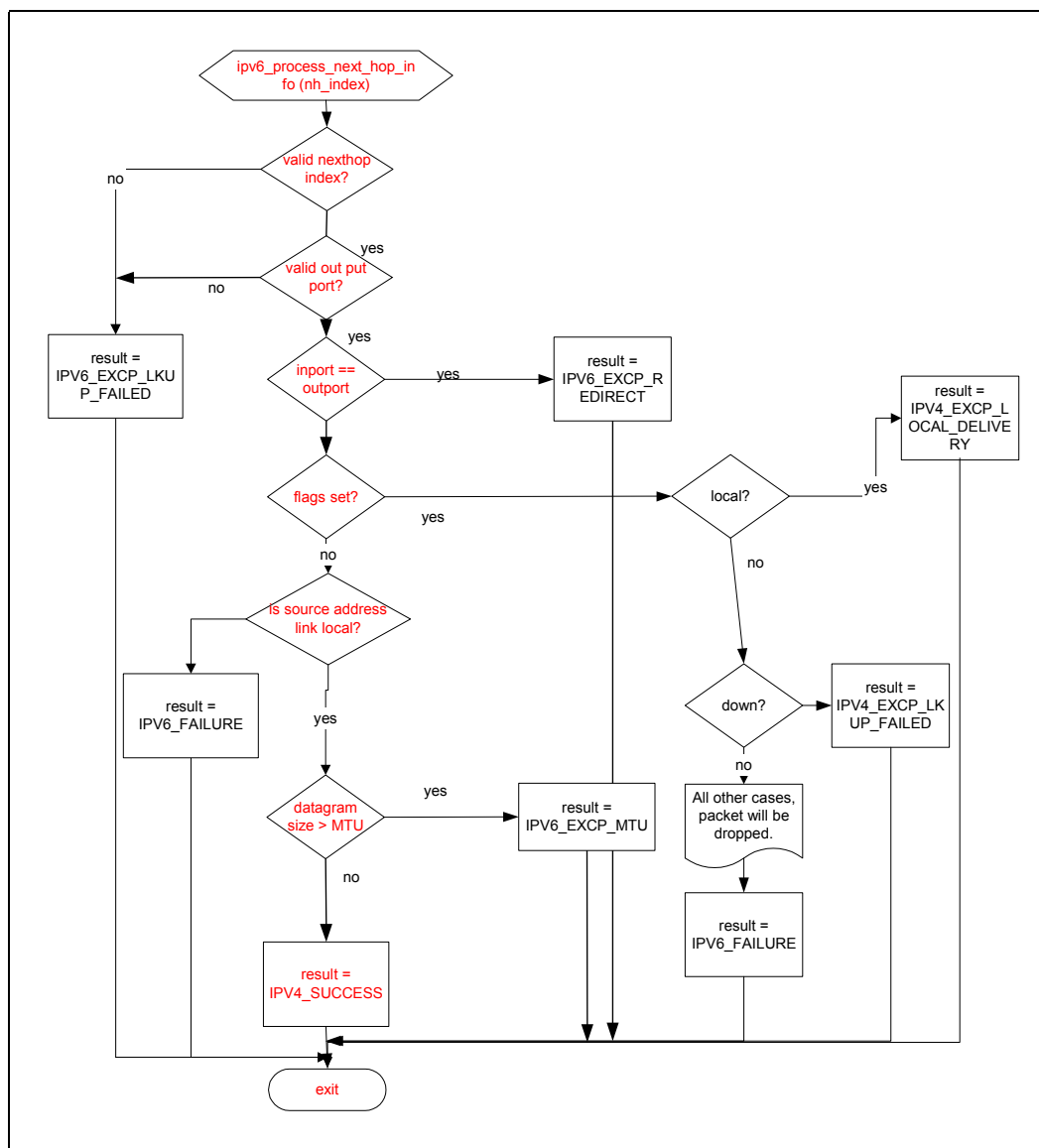


Figure 25-16. IPV6 Microblock Processing Next-Hop Information Flowchart



25.10 Performance Analysis

The IPv6 Forwarder runs as a functional pipeline on four microengines. The compute budget available is 117 times 4 cycles. The I/O latency available for the functional pipeline, to handle a Ethernet min packet is 117 times 8 times 4 cycles.

To understand the worst-case performance numbers we need to analyze the Lookup algorithm in the Forwarder. The data structures and the code are organized in such a way as to limit the number of SRAM accesses to 8 (when hashing is turned ON). The following equation gives the worst case

number of memory accesses (local, SRAM & the hash operation) which includes all the possible paths that a packet can take (with no assumptions on packet direction and the length of the router prefix)

- Packets that come into the router and whose destination matches the 64-bit cached router prefix,
7 Local Mem reads + L SRAM reads where $1 \leq L \leq 8$, L indicates a walk on the lower 64 bits of the address
- Packets that do not match the 64-bit cached value (destination is in a different domain or the router prefix is less than 64 bits)
(8-H) Local Mem reads + H SRAM reads + hash function, where $1 \leq H \leq 7$, H indicates a walk on the higher 64 bits

The above equations show that regardless of packet direction or prefix length, the number of SRAM accesses is bound by 8.

Considering the worst case in each of the scenarios we get the following cycle estimates,

- 7LM + 8 SRAM
ether_classify (60) + metadata setup etc (36) + Lookup (212) + DL_SOURCE (37) + DL_SINK (40) = 385 cycles
- 1LM + 7 SRAM + 1 hash
ether_classify (60) + metadata setup etc (36) + Lookup (174) + DL_SOURCE (37) + DL_SINK (40) = 347 cycles

Table 25-14 summarizes the performance analysis for the IPV6 Forwarder block.

Table 25-14. IPv6 Cycle Count Analysis

Cycle Count Analysis	Values
Available cycle count for 98 byte min Ethernet packet	(117 * 4) cycles (assuming 600 MHz IXP2400)
Estimated cycle count for min packet based on flow chart	~365

Table 25-15. IPv6 I/O Latency Analysis for Min Packet

I/O latency analysis for min packet	Values
Available I/O latency for min packet	$117 * 4 * 8 = 468$ cycles
Read packet header from DRAM	40 bytes (DL_SOURCE)
Trie lookup	15 SRAM reads (4 bytes each), worst case with Hashing turned ON, 8 SRAM reads (4 bytes each), worst case
Read next hop info from SRAM (16 Bytes)	2 SRAM reads (4 bytes each)
Write packet header to DRAM	40 bytes (DL_SINK)
Write packet metadata to SRAM	28 bytes (DL_SINK)
Scratch read	20 bytes (DL_SOURCE)
Scratch write	12 bytes (DL_SINK)

25.10.1 Characterization Data

Table 25-16. IPv6 Forwarder Microblock Characterization Data

Data	Value
General:	
Microblock Name	IPV6_FWDER_UC
Microblock Version Number	1.0
Implementation Language	microcode
Configuration Options use to gather this set of data	1. MICROENGINE 2. MICROCODE 3. CHIP_VERSION=IXP2XXX 5. META_CACHE_SIZE=7 6. IP_HDR_OFFSET=2 7. RFC2644_CHECKS 8. NEXTHOP_INFO_SRAM 9. PROCESS_CONTROL_BLOCK 10. IP_HDR_CACHE_LM 11. V6V4_UPDATE_OFFSET 12. RFC2373_CHECKS 13. V6V4_DECAP_VERIFY_IPV6_SOURCE
Measurement Environment (tool settings)	SDK 3.5. Assembler optimizer enabled
Performance: (cycle counts, latencies, bandwidths)	
Instruction cycle count per packet (common case packet / worst case cycle count)	250
Common-case packet/path assumptions to be documented here	Assumptions <ul style="list-style-type: none"> • The header is cached in local memory. • Packet is forwarded in common case. • Worst-case scenario for lookup is 32bit lookup with both hi64k and hi256 tables have to be traversed. • RFC1812 MUST, SHOULD cases, RFC2644 checks enabled. • PPP Header is 2 bytes, IP HDR offset is fixed at 2 bytes. • All numbers reported are for worst-case normal path.
Scratch Memory	
# of longwords read (for bandwidth calculations)	0
# of longwords written (for bandwidth calculations)	0
# and type of each atomic operation performed (for bandwidth calculations)	0
SRAM	

Table 25-16. IPv6 Forwarder Microblock Characterization Data (Continued)

Data	Value
# of longwords read	Trie Table: 8 NH information: 4 Router information: 10
# of longwords written	0
# and type of each atomic operation performed (for bandwidth calculations)	0
DRAM	
# of quadwords read	0
# of quadwords written	0
Other I/O instructions (i.e. MSF, PCI, CAP, and Hash)	0
List of dependent I/O accesses in the longest latency path	

Per-Microengine Resources:	
Control-Store Usage (# of instructions used)	395
Local Memory Footprint (# of long words used)	IPv6 header: 48 router prefix and trie-entry: 10
Local Memory Configuration (shared, or per-context pointer)	-
Local Memory - # of LM pointers used	2
GPR Usage – minimum, static usage (absolute, static, globals)	-
Transfer Reg. Usage – minimum, static usage	-
Next Neighbor Reg. Usage – minimum, static usage	-
Signal Usage – minimum, static usage	-
CAM used? (yes or no)	No

Global Resources:	
Scratch footprint (# of longwords used) – constant or formula ...	0
SRAM footprint (# of longwords used) – constant or formula ...	256 Kbytes for route table 321 Kbytes for 8 bit Trie Table 48 bytes per port for stats 4 Kbytes for Next Hop Table
DRAM footprint (# of quadwords used) – constant or formula ...	0
Q-Array usage - # of queues used and if they need to be cached	0
CRC Unit used?	no
Hash Unit used? (yes or no)	no

Table 25-16. IPv6 Forwarder Microblock Characterization Data (Continued)

Data	Value
MSF Usage Information:	
Media Bus Configuration	-
RBUF, TBUF usage	-
CBus signals	-
Other Information:	
Critical Section Length (compute cycles + memory accesses)	0
# of phases	1
Packet Metadata - fields read	packet_size, input_port class_id nexthop_id
Packet Metadata - fields written	nexthop_id_type output_port fabric_port
Header - fields read	All
Header - fields written	All
Documentation:	
Thread Ordering Requirements	yes
OS dependencies	VxWorks, Linux
Chip/hardware dependencies (i.e. Crypto Unit in 2850)	IXP2400, IXP2800, IXP2850
Tested on which SDK Release(s)	IXA SDK 3.5
Tested on hardware? Which hardware configuration?	Yes, IXDP 2400, IXDP2800
Tested in which applications (not an all inclusive list)	several SDK 3.5 applications
Possible Configuration Options	
Major limitations (i.e. Rx/Tx microblocks written to use entire ME)	High latency due to numerous dependent read operations
Packet Sequencing Issues (esp. in POT applications)	None
Core Component or Interface requirements or dependencies	IPV6 CC, RTMV6

IPv6 To IPv4 Tunneling Microblock 26

26.1 Overview

The Tunneling microblock on the Ingress IXP2400 consists of two microblocks that provide support for IPv6 over IPv4 tunneling. The V6V4-Tunnel decap microblock handles IPv4 packets that contain an IPv6 packet and have reached a tunnel endpoint. The V6V4-Tunnel encap microblock handles IPv6 packets that require encapsulation in an IPv4 packet in order to reach the next-hop IPv6 node. The block is run in a functional pipeline on four microengines (preferably with two microengines per cluster to balance the usage of the command bus and the S-Push/Pull buses). Each thread executing this block handles one packet at a time. The threads execute in strict order and the ordering is maintained by the `dl_source` and `dl_sink` blocks in the dispatch loop. This ensures that packet ordering is maintained. It is expected that the application includes microblocks for receive, transmit, IPv4 forwarding and IPv6 forwarding. Other microblocks may be present as required by the application.

The tunneling microblocks provide the capability for the node to serve as an endpoint of an IPv6 over IPv4 tunnel. The types of tunneling supported are:

- Configured Tunnels as defined in RFC 2893
- Automatic Tunnels as defined in RFC 2893
- 6to4 tunnels as defined in RFC 3056

The specific checks performed are described in [Section 26.5, “RFC Compliance” on page 455](#). The functionality of the microblock is described in the high-level flow chart in [Section 26.7, “V6V4-Tunnel-Decap Microblock” on page 457](#) and [Section 26.8, “V6V4-Tunnel-Encap Microblock” on page 468](#).

26.2 Assumptions

The following assumptions are made in the design and implementation of the IPv6-IPv4 tunneling microblock:

- The tunneling microblocks are transform microblocks. They must run as part of a microblock group that includes the source and sink microblocks.
- The microblock assumes that packet metadata is available via the dispatch loop macros described in the *Intel® Internet Exchange Architecture Portability Framework Developer's Manual*. The specific variables used by each microblock are specified in the later sections describing the details of the microblocks.
- The microblocks assume that the packet header can be accessed via the `xbuf_` API provided with the Intel® Internet Exchange Architecture Software Development Kit (Intel® IXA SDK). The specific macros assumed by each microblock are specified in the later sections describing the details of the microblocks. The microblock assumes that these `xbuf` macros can operate on transfer registers, general purpose registers (GPRs) and local memory.

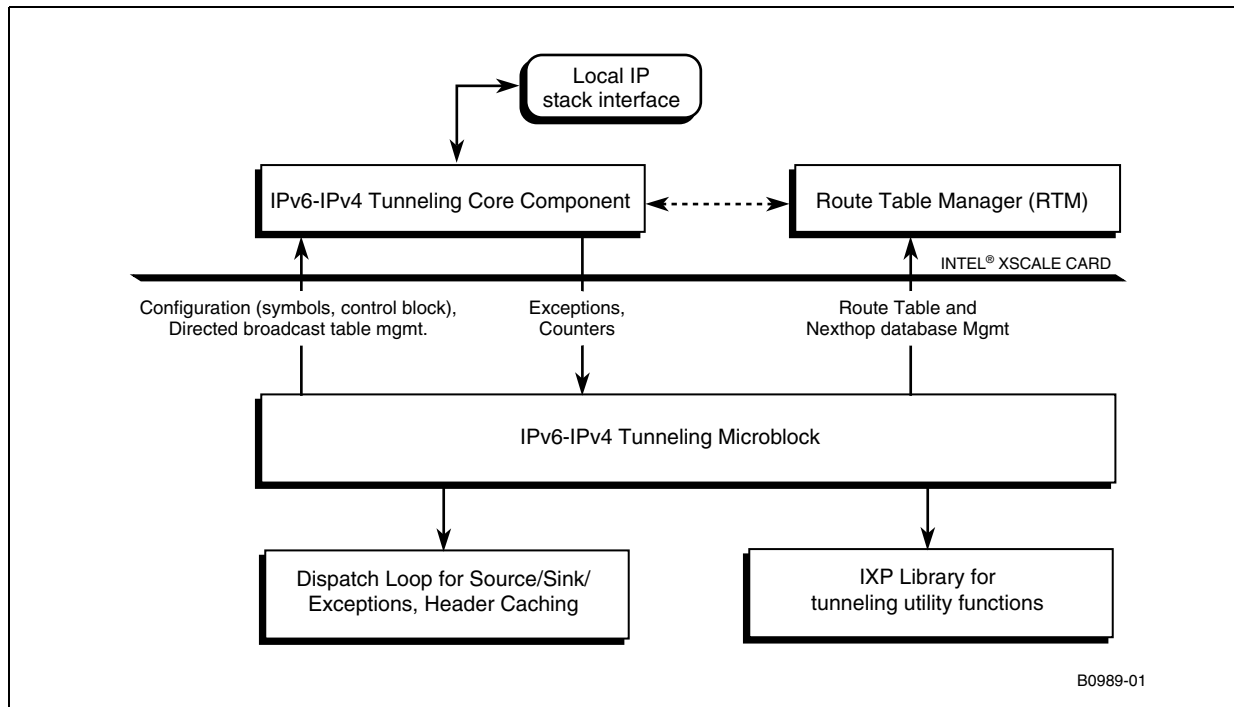
The primary goal of this microblock is to achieve maximum performance. Hence the following assumptions apply:

- The microblock doesn't perform any error checking on the data structures shared with XScale core. The XScale core code has to populate the structures with correct values. Otherwise, the results are unpredictable.
- The microblock tries to take advantage of the layout of the above structures (and the member fields inside). Hence, the XScale core code cannot assume to re-use the reserved fields for some other purpose without affecting the microblock functionality.
- A trade-off is evaluated between efficiency of the microblock and the degree of details provided in the exception codes when the packet has to be sent to XScale core.
- All reserved fields in the data structures must be cleared to zero, unless otherwise specified.
- The threads executing this microblock does not maintain strict ordering (because of the different amounts of processing and I/O required per packet). As a consequence, the packets can get out of order by the end of microblock. Hence, it is the responsibility of the dispatch loop using this microblock to ensure packet order around this block.
- The packet ordering is not maintained when there are exception packets, as these packet are sent to XScale core and the microblock (or the dispatch loop) has no way of ensuring the packet ordering on XScale core.
- The microblock always performs the MUST checks specified in RFC 3056 and RFC 2893 (see section 7.5) It is assumed that the applications always require these checks. The RFC SHOULD checks may be disabled as a compile-time option.
- The XScale core code has to create and maintain/update the shared data structures. The microblock simply reads and processes the information. Only counters are an exception, as they are updated by the microblock.
- All address and offsets specified in this section are byte addresses.
- The values in the data structures shared by this block are in network byte order (big-endian).

26.3 Dependencies

The main dependencies of this microblock are shown in Figure 26-1. The inward arrows indicate dependency on the IPv6-IPv4 tunneling microblock. The outward arrows show what the microblock expects from other modules.

Figure 26-1. IPv6-IPv4 Tunneling Microblock Dependencies



26.4 Configuration Options

26.4.1 Build Switches

Table 26-1 identifies compile time build switches, which can be turned on or off as per the requirements of a specific application.

Table 26-1. IPv6 to IPv4 Tunneling Build Switches (Sheet 1 of 2)

Symbol	Description
MICROENGINE	Enables microengine specific definitions
MICROCODE	Enables definitions specific to microcode. If not enabled, then the definitions apply to microC code.
CHIP_VERSION	Enables IXP library to work on IXP2xxx processors
META_CACHE_SIZE	Defines the number of long words of metadata to be cached
DL_NEXT_ME	Next ME in the functional pipeline w.r.t the ME in question. Needed by dispatch loop.

Table 26-1. IPv6 to IPv4 Tunneling Build Switches (Sheet 2 of 2)

Symbol	Description
V6V4_INGRESS_LIST_MAX_ADDR	Maximum number of addresses allowed in the ingress source list block (1-12). Not needed if V6V4_INGRESS_LIST_TRIE is set.
V6V4_INGRESS_LIST_INDEX_LIST_BLOCK_SIZE	Size of index blocks used with trie table based ingress source list (0 - 32). Not needed if V6V4_INGRESS_LIST_TRIE is not set.
V6V4_UPDATE_OFFSET	Enables updating of metadata variable offset when encapsulating or decapsulating a packet.
V6V4_DECAP_VERIFY_IPV4_SOURCE	Enables validation of IPv4 source address during decapsulation as specified in RFC 2893 section 3.6, if the address is not validated using the ingress source list.
V6V4_DECAP_VERIFY_IPV6_SOURCE	Enables validation of the IPv6 source address during decapsulation as specified in RFC 2893 section 3.6.
V6V4_ENCAP_VERIFY_DEST_AUTO	Enables validation during encapsulation that the destination IPv4 address to be used for an automatic tunnel is not a loopback or unspecified address.
V6V4_INGRESS_LIST_TRIE	Specifies that trie tables should be used for ingress source list instead of simple list.
V6V4_DECAP_COUNTERS	Enables statistics maintenance for decapsulation.
V6V4_ENCAP_COUNTERS	Enables statistics maintenance for encapsulation.

26.4.2 Default Configuration

The build switches with which the microblock is released are as follows:

- MICROENGINE\$
- CHIP_VERSION=IXP2xxx\$
- META_CACHE_SIZE=8
- DL_NEXT_ME=x, where x is given in [Table 26-2](#). (The ME number in 0xAB format implies the ME 'B' in cluster 'A').

Table 26-2. Next ME Values

ME	Next ME
0x01	0x02
0x02	0x11
0x11	0x12
0x12	0x01

- V6V4_INGRESS_LIST_MAX_ADDR = 12
- V6V4_UPDATE_OFFSET
- V6V4_DECAP_VERIFY_IPV6_SOURCE

26.5 RFC Compliance

26.5.1 Decapsulation

The V6V4-Tunnel-Decap microblock performs the following MUST checks from RFC 3056 provided that the packet was received on an interface configured for 6to4 tunnels:

Table 26-3. RFC 3056 *Must* Checks Performed in the Microblock

Serial No.	RFC3056 MUST Check	Action
1	Packet has 2002:/16 source or destination address that embeds an IPv4 broadcast address.	Drop
2	Packet has 2002:/16 source or destination address that embeds an IPv4 multicast address.	Drop
3	Packet has 2002:/16 source or destination address that embeds the IPv4 loopback address (127.0.0.1).	Drop
4	Packet has 2002:/16 source or destination address that embeds the IPv4 unspecified address (0.0.0.0).	Drop
5	Packet has 2002:/16 source or destination address that embeds an IPv4 private address.	Drop
6	Packet has 2002:/16 source or destination address that embeds an IPv4 subnet broadcast address.	Drop

The V6V4-Tunnel-Decap microblock performs the following SHOULD checks from RFC 2893 if the flag `V6V4_DECAP_VERIFY_IPV4_SOURCE` is set:

Table 26-4. RFC 2893 *Should* Items for IPv4 Source Address

Serial No.	RFC2893 SHOULD Check	Action
1	Packet has a multicast address for the IPv4 source address.	Drop
2	Packet has a broadcast address for the IPv4 source address.	Drop
3	Packet has the loopback address (127.0.0.1) for the IPv4 source address.	Drop
4	Packet has the unspecified address (0.0.0.0) for the IPv4 source address.	Drop

The V6V4-Tunnel-Decap microblock performs the following SHOULD checks from RFC 2893 if the flag `V6V4_DECAP_VERIFY_IPV6_SOURCE` is set:

Table 26-5. RFC 2893 *Should* Items for IPv6 Source Address

Serial No.	RFC2893 SHOULD Check	Action
1	Packet has a multicast address for the IPv6 source address.	Drop
2	Packet has the loopback address for the IPv6 source address.	Drop
3	Packet has the unspecified address for the IPv6 source address.	Drop
4	Packet has an IPv4-compatible IPv6 source address, and the embedded IPv4 address is a broadcast address.	Drop

Table 26-5. RFC 2893 *Should* Items for IPv6 Source Address (Continued)

Serial No.	RFC2893 SHOULD Check	Action
5	Packet has an IPv4-compatible IPv6 source address, and the embedded IPv4 address is a multicast address.	Drop
6	Packet has an IPv4-compatible IPv6 source address, and the embedded IPv4 address is the loopback address (127.0.0.1).	Drop
7	Packet has an IPv4-compatible IPv6 source address, and the embedded IPv4 address is the unspecified address (0.0.0.0).	Drop

26.5.2 Encapsulation

The V6V4-Tunnel-Encap microblock always performs the first three items of the following list of *Must* checks from RFC 2893. Since items 4 and 5 are likely to be performed by the IPv4 Forwarder, they are performed only if the compile flag `V6V4_ENCAP_VERIFY_DEST_AUTO` is set.

Table 26-6. RFC 2893 *Must* Items for IPv4 Destination Address

Serial No.	RFC2893 MUST Check	Action
1	Automatic tunnel cannot send to broadcast address.	Drop
2	Automatic tunnel cannot send to multicast address.	Drop
3	Automatic tunnel cannot send to subnet broadcast address.	Drop
4	Automatic tunnel cannot send to loopback address (127.0.0.1).	Drop
5	Automatic tunnel cannot send to unspecified address (0.0.0.0).	Drop

The V6V4-Tunnel-Encap microblock performs the following *MUST* checks from RFC 3056 provided for packets being sent on an interface configured for 6to4 tunnels:

Table 26-7. RFC 3056 *Must* Items for IPv6 Source and Destination Address

Serial No.	RFC3056 MUST Check	Action
1	Packet has 2002::/16 source or destination address that embeds an IPv4 broadcast address.	Drop
2	Packet has 2002::/16 source or destination address that embeds an IPv4 multicast address.	Drop
3	Packet has 2002::/16 source or destination address that embeds the IPv4 loopback address (127.0.0.1).	Drop
4	Packet has 2002::/16 source or destination address that embeds the IPv4 unspecified address (0.0.0.0).	Drop
5	Packet has 2002::/16 source or destination address that embeds an IPv4 private address.	Drop
6	Packet has 2002::/16 source or destination address that embeds an IPv4 subnet broadcast address.	Drop

26.6 Statistics Counters

The V6V4-Tunnel-Decap block can be configured to maintain counters for statistics gathering. If the compiler option `V6V4_DECAP_COUNTERS` is set, then block maintains 32-bit counters for the following:

- The number of packets dropped
- The number of packets sent to the core component as exceptions

The V6V4-Tunnel-Encap block can be configured to maintain counters for statistics gathering. If the compiler option `V6V4_ENCAP_COUNTERS` is set, the block maintains 32-bit counters for the following:

- The number of packets dropped
- The number of packets sent to the core component as exceptions.

The microblock always maintains 32-bit counters. Code running on the XScale core can keep 64-bit counters if required (see [Section 2.5, “Statistics and Handling of 64-bit Counters” on page 63](#)).

26.7 V6V4-Tunnel-Decap Microblock

26.7.1 Introduction

The V6V4-Tunnel-Decap microblock performs the packet processing required for IPv6 packets that have been tunneled over IPv4 and have reached the tunnel endpoint.

Upon entry, the V6V4-Tunnel-Decap microblock assumes that a previous microblock in the group, presumably an IPv4 forwarder, has performed a lookup on the IPv4 destination address of the packet and has set the packet metadata variables `next_hop_id` and `next_hop_id_type`. The variable `next_hop_id` is interpreted as the index of some information that is pertinent to the next stage of packet processing. The variable `next_hop_id_type` specifies the type of information indexed by `next_hop_id`. If the value of `next_hop_id_type` does not indicate IPv6-over-IPv4 tunneling, then the V6V4-Tunnel-Decap block sets `dl_next_block` to `V6V4_NEXT_NO_DECAP` and no further processing is performed. If the value of `next_hop_id_type` indicates IPv6-in-IPv4 tunneling, then the microblock continues to process the packet.

The value of `next_hop_id` is used to index an array in memory referred to as the tunnel next hop table for ending tunnel endpoints. Each entry in this array describes what type of tunneling is supported for the destination and what type of header validation, if any, is required. The V6V4-Tunnel-Decap microblock reads the appropriate entry and performs the required validation. If the validation fails, the packet is dropped. If the validation succeeds, the metadata variables `packet_size` and `buffer_size` are decremented by the size of an IPv4 header, the metadata variable `header_type` is set to `V6V4_HDR_TYPE_DECAP` to indicate a decapsulated IPv6 packet, and `dl_next_block` is set to `V6V4_NEXT_IPV6`. The metadata variable `buffer_offset` may optionally be updated as a result of executing this microblock, as described in the [Section 26.7.1.1, “Header Cache and Metadata Requirements” on page 458](#).

The type of header validation performed depends on the information configured in the tunnel next-hop entry. If required, the IPv4 source address of the packet is checked against a list of approved IPv4 source prefixes. This list is referred to as the ingress source list in this document. If the source

address is not validated against the ingress source list, it may still be checked to verify that it is not a broadcast address or other address type that is not acceptable for the tunnel. The destination address may be checked for acceptability as well.

Note that `V6V4_NEXT_NO_DECAP` and `V6V4_NEXT_IPV6` are defined outside the microblock and must be bound to actual block identifiers by the dispatch loop. To allow the application flexibility in assigning header types, the header type value `V6V4_HDR_TYPE_DECAP` is also defined outside the block.

26.7.1.1 Header Cache and Metadata Requirements

The V6V4-Tunnel-Decap microblock assumes that the fields in the IPv4 header can be accessed via the `xbuf_extract` API, and assumes that the header is available in the appropriate buffer at the time the block is invoked. The name of this buffer and the offset of the start of the IPv4 header are passed to the microblock as parameters. The V6V4-Tunnel-Decap microblock assumes that the encapsulated IPv6 header has not yet been loaded into the header cache, and requires the application to provide a dispatch loop macro that allows the IPv6 header to be loaded in a manner consistent with the application's header caching scheme. The name of the buffer into which the IPv6 header should be loaded is passed to the microblock as a parameter, along with the buffer offset at which to load the header.

The V6V4-Tunnel-Decap block executes this macro after it has determined that the packet requires decapsulation. The macro has the following format:

```
dl_load_ipv6_hdr[xbuf_name, xbuf_offset, BYTE_COUNT]
```

where:

`xbuf_name` is the name of the buffer into which the header should be loaded

`xbuf_offset` is the offset in `xbuf_name` at which the header should be loaded

`BYTE_COUNT` is the number of bytes to be loaded

After the microblock executes this macro, it reads the header fields using `xbuf_extract`.

The V6V4-Tunnel-Decap microblock may optionally update the metadata value `buffer_offset` after it decapsulates a packet. A compiler flag controls this option. If the flag `V6V4_UPDATE_OFFSET` is defined, the microblock increments `buffer_offset` by the size of the IPv4 header. If this flag is not defined, the block does not modify `buffer_offset`.

26.7.2 Data Structures

26.7.2.1 Tunnel Next-Hop information

The V6V4-Tunnel-Decap microblock uses the metadata variable `next_hop_id` as an index into an SRAM array of tunnel next-hop structures. Each entry describes the ending endpoint of a tunnel. The format of the tunnel next-hop information for ending tunnel endpoints is specified in Table 26-8.

Table 26-8. Tunnel Next-Hop Entry for V6V4-Tunnel-Decap Microblock

LW	Bits	Size	Field	Description
0	31:16	16	Flags	<p>Indicates how decapsulated packets are handled:</p> <p>V6V4_DECAP_FLAGS_VALID (bit 31): Indicates a valid entry.</p> <p>V6V4_DECAP_FLAGS_AUTO (bit 16): Indicates the node should accept packets addressed to the IPv4-compatible address.</p> <p>V6V4_DECAP_FLAGS_SRC_VALIDATE (bit 17): Indicates whether the node should perform ingress source validation.</p> <p>V6V4_DECAP_FLAGS_6TO4 (bit 18): Indicates whether the node should check the source and destination addresses for invalid 6to4 addresses.</p> <p>V6V4_DECAP_FLAGS_TOS_V6 (bit 19): Indicates whether the node should set the IPv6 TrafficClass field to the Type of Service field in the encapsulating header.</p> <p>The remaining bits (30:20) are reserved and must be zero.</p>
0	15:0	16	IngressList	Index of Ingress Source List block.

Each entry in the tunnel next hop table for ending tunnel endpoints contains a single 32-bit longword. Bit 31, `V6V4_DECAP_FLAGS_VALID`, is set to indicate a valid entry.

The remaining flags control how a packet that needs decapsulation is processed. If `V6V4_DECAP_FLAGS_AUTO` is set, this means the packets with an IPv4-compatible destination address that matches the local interface address should accepted and sent to the core component for further processing. If this flag is not set then all packets received on the tunnel that have an IPv4-compatible destination address are dropped.

`V6V4_DECAP_FLAGS_SRC_VALIDATE` controls whether the packet's IPv4 source address is validated prior to decapsulation. For a configured tunnel, typically this flag is set. If set, then only those packets whose IPv4 source address matches a prefix in the ingress source list is decapsulated and passed on. If this flag is set and the ingress source list is empty then all packets addressed to the tunnel is dropped except for automatic tunneled packets (if `V6V4_DECAP_FLAGS_AUTO` is set).

For a 6to4 tunnel, `V6V4_DECAP_FLAGS_SRC_VALIDATE` may or may not be set according to the application's needs. For example, if the application wants to restrict the tunnel to accept packets only from a limited number of 6to4 routers, then it can set this flag and configure the ingress list accordingly. Bits 0-15 of the longword contain an index to the ingress source list, which is described in the following section.

V6V4_DECAP_FLAGS_6TO4 controls whether the tunnel decapsulation block checks the packet for 6to4 source and destination addresses. If this flag is set, the block checks the packet for a 6to4 source or destination address, and if detected, it verifies that the address is acceptable as defined in RFC 3056 section 9.

V6V4_DECAP_FLAGS_TOS_V6 controls whether the Type of Service field in the IPv4 header should be copied to the Traffic Class field in the IPv6 header. RFC 2893 section 3.6 states that the decapsulating node does not modify the IPv6 header, but future RFCs might define different behavior with respect to the Type of Service byte. This flag allows the node to override the default behavior if it becomes necessary at some future time.

26.7.2.2 Ingress Source List

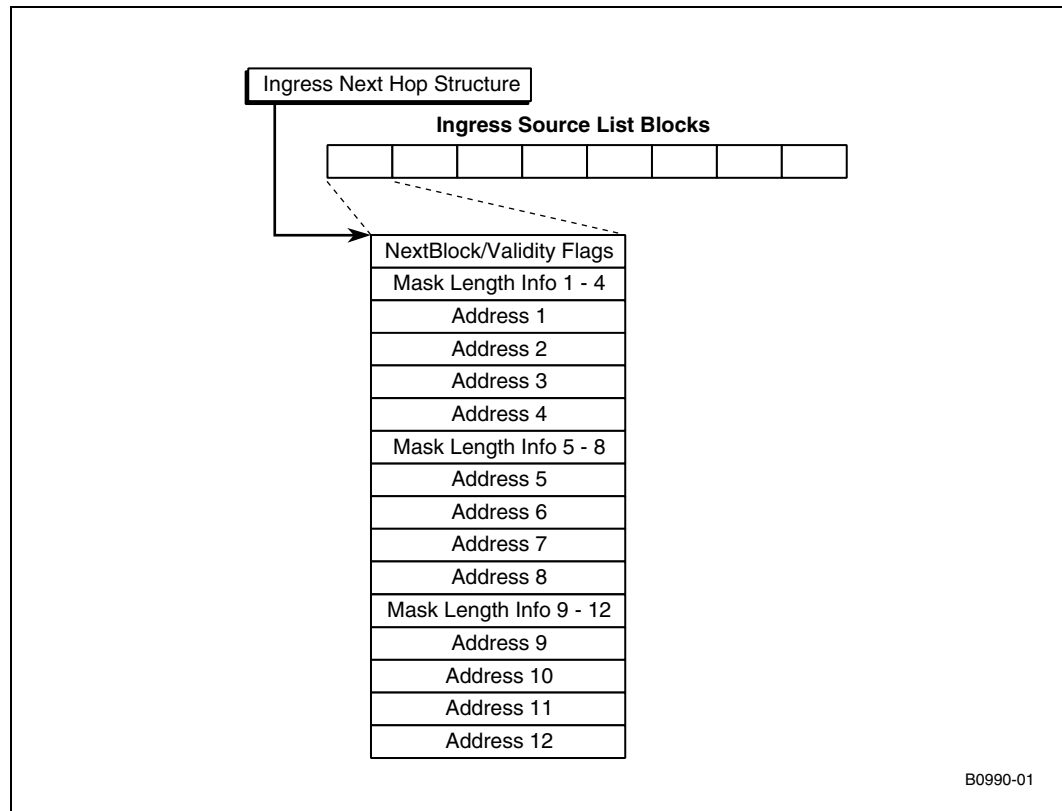
The tunnel ingress source list is a list of acceptable source prefixes for an incoming tunnel endpoint. Figure 26-2 shows the layout of the ingress source list.

The tunnel ingress source list is stored in an ingress list block. Ingress list blocks for the application as a whole are stored as an array in SRAM. The block for a specific tunneling endpoint is found in the ingress tunnel next-hop structure for that endpoint. Each ingress list block is an array of 16 longwords. Table 26-9 summarizes the format of the ingress list block.

Table 26-9. Format of Ingress Source List Block

LW	Bits	Size	Field	Description
0	31:16	16	Next Block	Index of next block
0	15:12	4	Reserved	Must be zero.
0	11:0	12	ValidityFlags	Flags indicating whether each address entry is valid. Bit 0 corresponds to address 1, bit 1 to address 2, etc.
1	31:24	8	Mask1	Number of zero bits in mask for Address 1.
1	23:16	8	Mask2	Number of zero bits in mask for Address 2.
1	15:8	8	Mask3	Number of zero bits in mask for Address 3.
1	7:0	8	Mask4	Number of zero bits in mask for Address 4.
2-5	31:0	32x4	Address1_4	Addresses 1 - 4
6	31:24	8	Mask5	Number of zero bits in mask for Address 5.
6	23:16	8	Mask6	Number of zero bits in mask for Address 6.
6	15:8	8	Mask7	Number of zero bits in mask for Address 7.
6	7:0	8	Mask8	Number of zero bits in mask for Address 8.
7-10	31:0	32x4	Address5_8	Addresses 5 - 8
11	31:24	8	Mask9	Number of zero bits in mask for Address 9.
11	23:16	8	Mask10	Number of zero bits in mask for Address 10.
11	15:8	8	Mask11	Number of zero bits in mask for Address 11.
11	7:0	8	Mask12	Number of zero bits in mask for Address 12.
12-15	31:0	32x4	Address9_12	Addresses 9 - 12

Figure 26-2. Ingress Source List Blocks



Longword 0 of the array contains validity flags in the low order 12 bits. If a bit is set, the corresponding address and prefix length specifier indicate a valid source prefix for the incoming tunnel. If a bit is not set, the corresponding address and prefix length specifier should be ignored. The upper 16 bits of longword 0 contain the index of the next block for the same tunnel, if there is one. A value of zero indicates there is no next block.

Longword 1 of the block contains mask length information for the first 4 entries in the block. Each entry uses 8 bits, and can assume values from zero to 31. The value specifies the number of “don't care” bits at the end of the address, hence the value is actually 32 minus the prefix length. Each address entry contains a prefix that is acceptable for the tunnel, with the non-prefix bits set to zero.

Longwords 6 and 11 contain the prefix length information for addresses 5-8 and 9-12, respectively.

This structure is set up so that an application can be configured to read a smaller number of longwords of this structure. If the number of source address prefixes for a given local address is limited to 12 or less, then the application might want to avoid reading and checking the unused words. To use this option, the application defines the value `V6V4_INGRESS_LIST_MAX_ADDR` to the appropriate value. This value should be the maximum number of addresses that can exist in the structure, and therefore must be a number from 1 through 12. If this value is set, then the NextBlock field in the ingress list block is not used. If more than 12 prefixes are possible, then `V6V4_INGRESS_LIST_MAX_ADDR` must not be defined. This enables the use of multiple blocks.

The ingress source list is configured and updated by the Intel XScale[®] core component. To avoid invalidating the structure while adding an address to a block, the address and prefix longwords are updated first, and lastly the validity flags are updated with the corresponding bit set. To delete an address, the validity flags are simply updated to clear the corresponding bit.

26.7.2.3 Trie Table Option

The format for the ingress source list was chosen under the assumption that the number of address prefixes in the ingress source list for any given interface is small, so that the vast majority of validations should need to read just one block.

If larger lists are expected, then a more scalable approach, such as the trie tables similar to those used in the IPv4 and IPv6 Forwarders can be used instead. Trie tables are described in detail in section 6. To choose the trie-table option, the application sets the compiler flag `V6V4_INGRESS_LIST_TRIE`. To conserve memory usage, trie tables used for tunneling are shared among all ending tunnel endpoints, and have an 8-bit root and 4 bit extensions. A successful lookup yields an index that identifies an entry in another table of index blocks. This index block contains a list of IDs of tunnels that allow the prefix. The size of each index block is derived from the compiler define `V6V4_INGRESS_LIST_INDEX_BLOCK_SIZE`. Acceptable values for `V6V4_INGRESS_LIST_INDEX_BLOCK_SIZE` are as follows:

- 8—index block holds 3 tunnel IDs
- 16—index block holds 7 tunnel IDs
- 32—index block holds 15 tunnel IDs
- 64—index block holds 31 tunnel IDs

26.7.3 Process Flow

The flow chart in [Figure 26-3](#) shows the high-level process flow for the microblock.

Once it determines (by examining `next_hop_id_type`) that a packet needs to be processed, the V6V4-Tunnel-Decap microblock compares the IPv4 protocol to the known value for IPv6 over IPv4 tunneling (41). If there is a mismatch, the packet is sent to the core component as an exception. Recall that the lookup that was performed prior to execution of the tunneling decapsulation block has already determined that the packet's destination address matches a local interface. Therefore, if the packet is not a tunneled packet, it is a locally addressed packet and is handled by the core component. The exception is handled in this microblock instead of passing the packet on to another microblock to handle local packets so that the core component can check for ICMP error messages received for outgoing tunneled packets that were sent from the same local address.

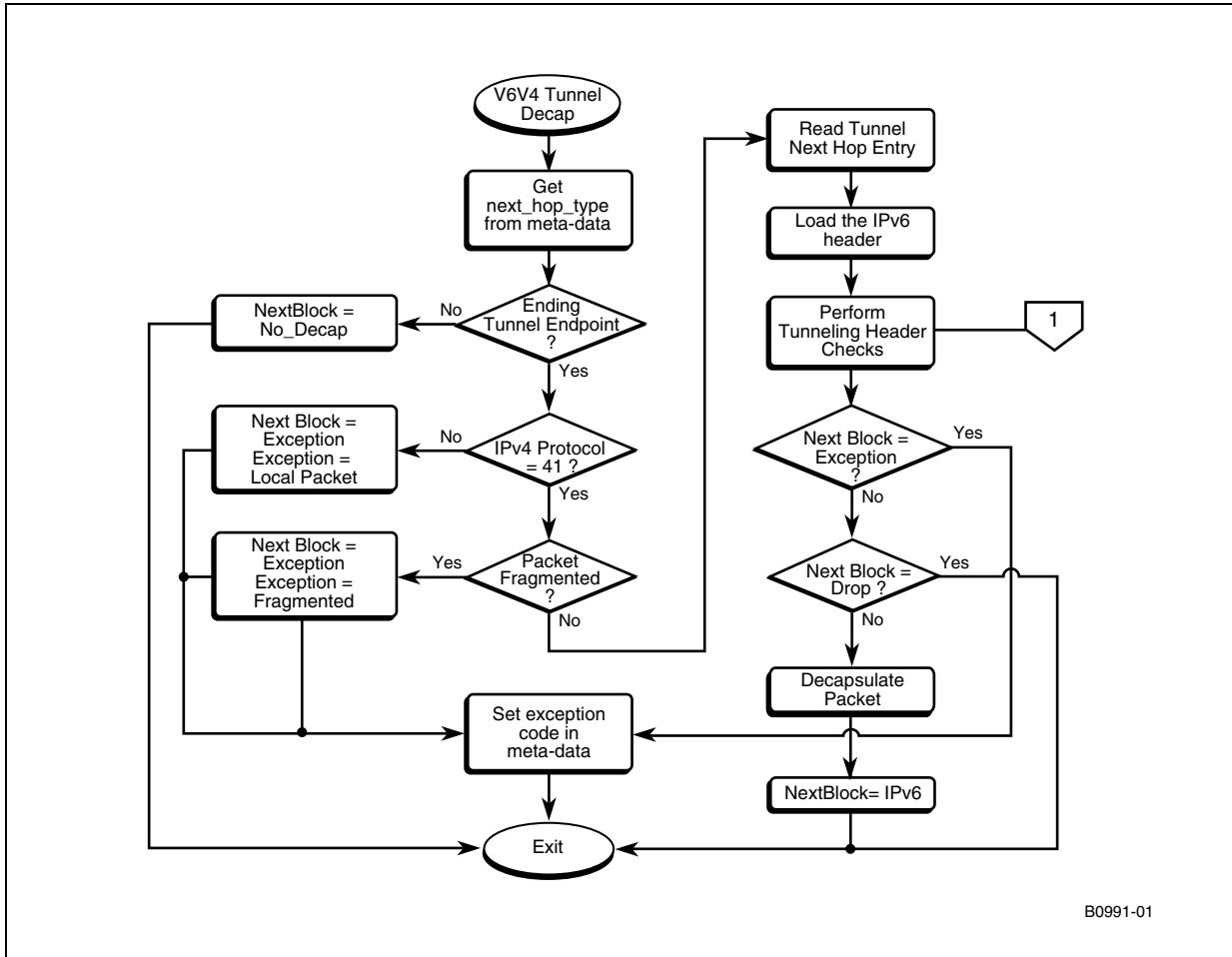
If the IPv4 protocol field is equal to 41, which indicates an encapsulated IPv6 packet, the microblock checks for a fragmented packet, and if detected, the packet is sent to the core component as an exception. If the packet is not fragmented, the microblock proceeds with header validation.

If header validation does not indicate the packet should be dropped or sent to the core component as an exception, the packet is decapsulated. Decapsulation consists of:

1. Decrementing the metadata variables `packet_size` and `buffer_size` by the size of the IPv4 header.

2. Incrementing the metadata variable offset by the size of the IPv4 header, if the compiler switch V6V4_UPDATE_OFFSET is set.
3. Setting the metadata variable header_type to V6V4_HDR_TYPE_DECAP.

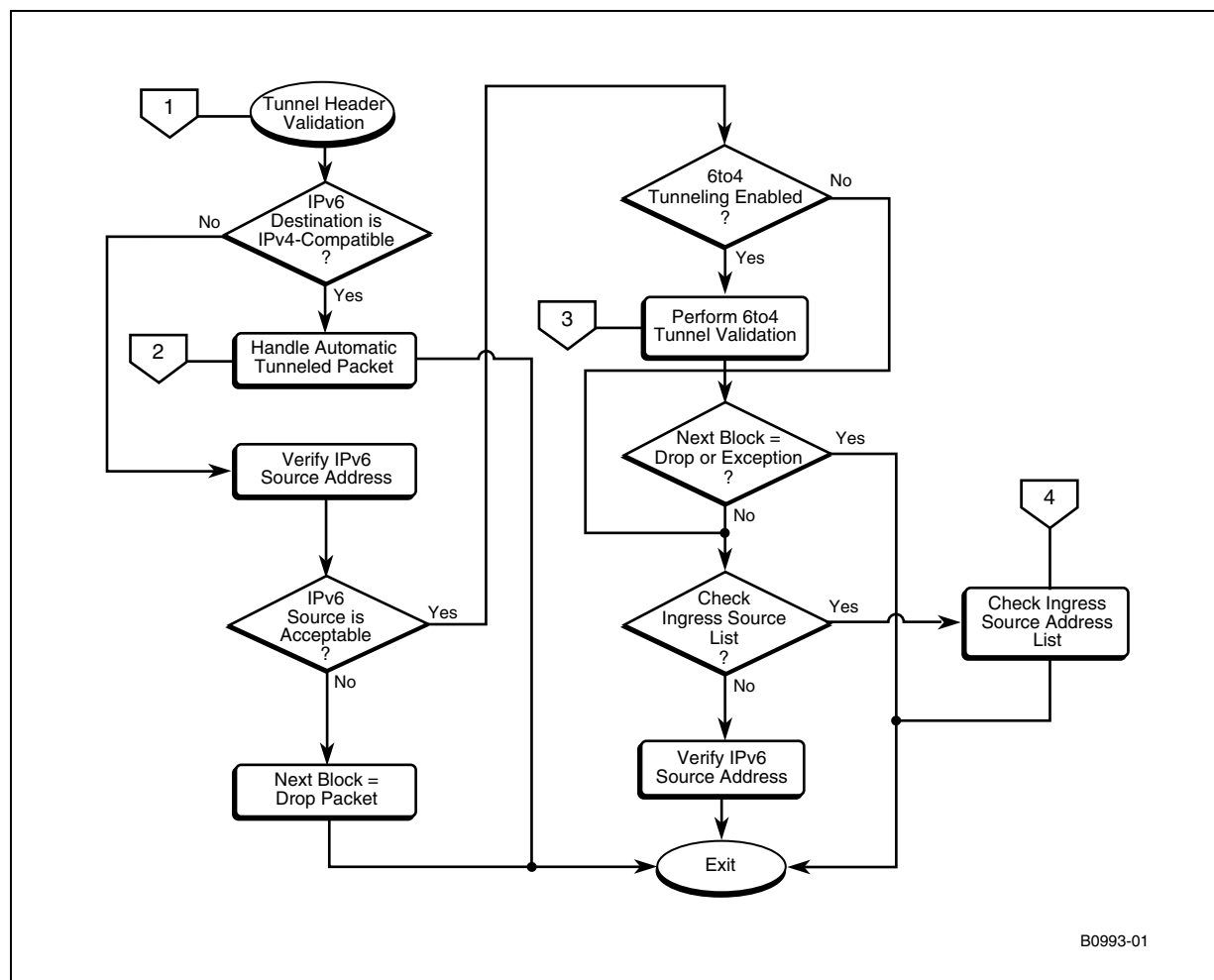
Figure 26-3. Process Flow for V6V4-Tunnel-Decap Microblock



26.7.3.1 Tunneling Header Checks

The high-level header validation process is described in Figure 26-4.

Figure 26-4. Process Flow for V6V4-Tunnel-Decap Header Validation



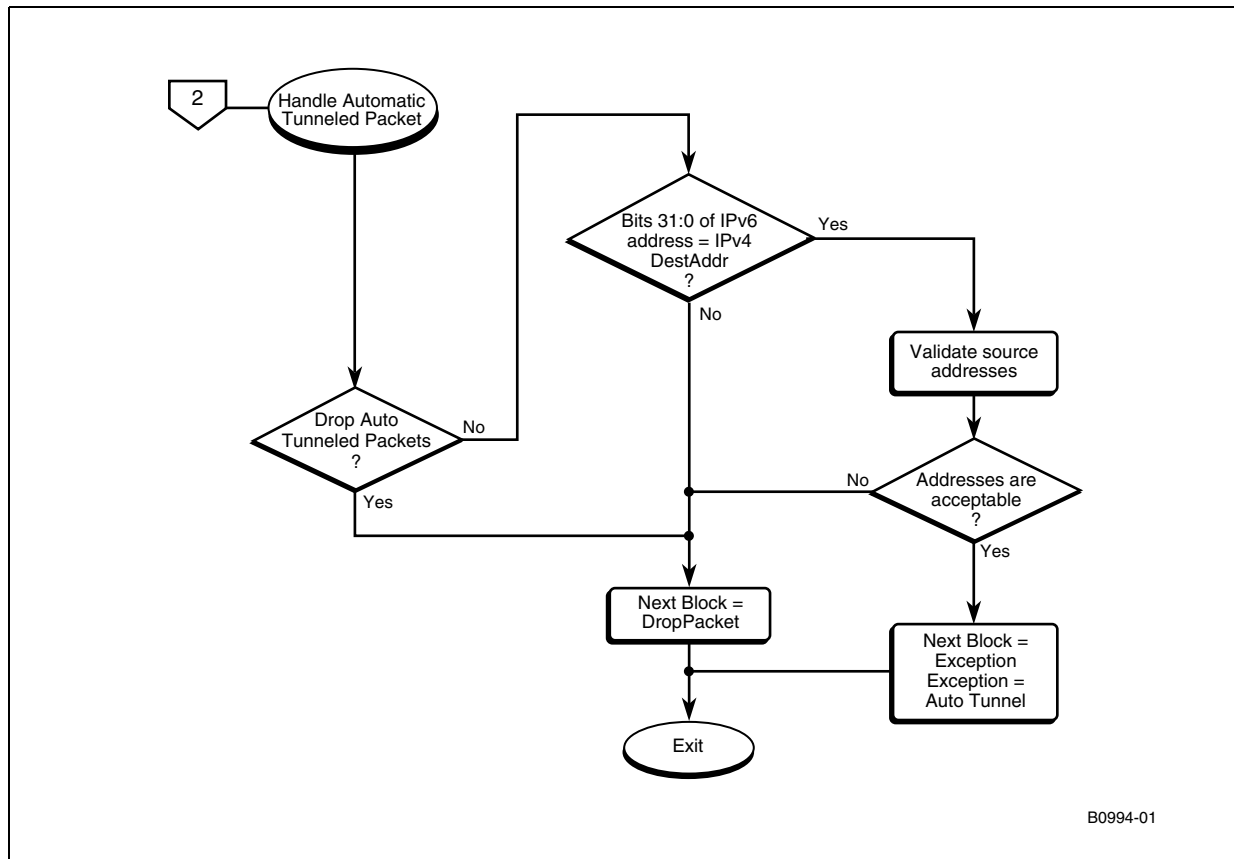
The header check process begins with checks for an automatic tunneled packet. If an IPv4-compatible destination address is detected, the automatic tunnel validations are performed. These checks result in either a dropped packet or a packet sent to the core component as an exception. If the packet does not have an IPv4-compatible destination address, the IPv6 source address is validated according to RFC 2893 section 3.6 if the compiler switch `V6V4_DECAP_VERIFY_IPV6_SOURCE` is set. Specifically, it is checked to ensure it is not a multicast, loopback or unspecified address or an IPv4 compatible address that embeds a multicast, broadcast, loopback or unspecified IPv4 address. This step is bypassed if the automatic tunneling checks are done since they include those same validations. If the packet was not dropped due to these validations, then the 6to4 validations are performed if the tunnel is enabled for 6to4. If those checks did not yield a dropped packet, then the IPv4 source address is checked against the ingress source list, if the tunnel endpoint is configured to do so. If the tunnel endpoint is not configured to verify the IPv4 source address against the ingress source list, then the IPv4 source address is

validated according to RFC 2893 section 3.6 if the compiler switch V6V4_DECAP_VERIFY_IPV4_SOURCE is set. Specifically, the packet is dropped if the IPv4 source address is a broadcast, multicast, loopback or unspecified address.

26.7.3.2 Automatic Tunneling Verification

Figure 26-5 shows the verification process for automatic tunneling.

Figure 26-5. Process Flow for V6V4-Tunnel-Decap Validation for Automatic Tunneling



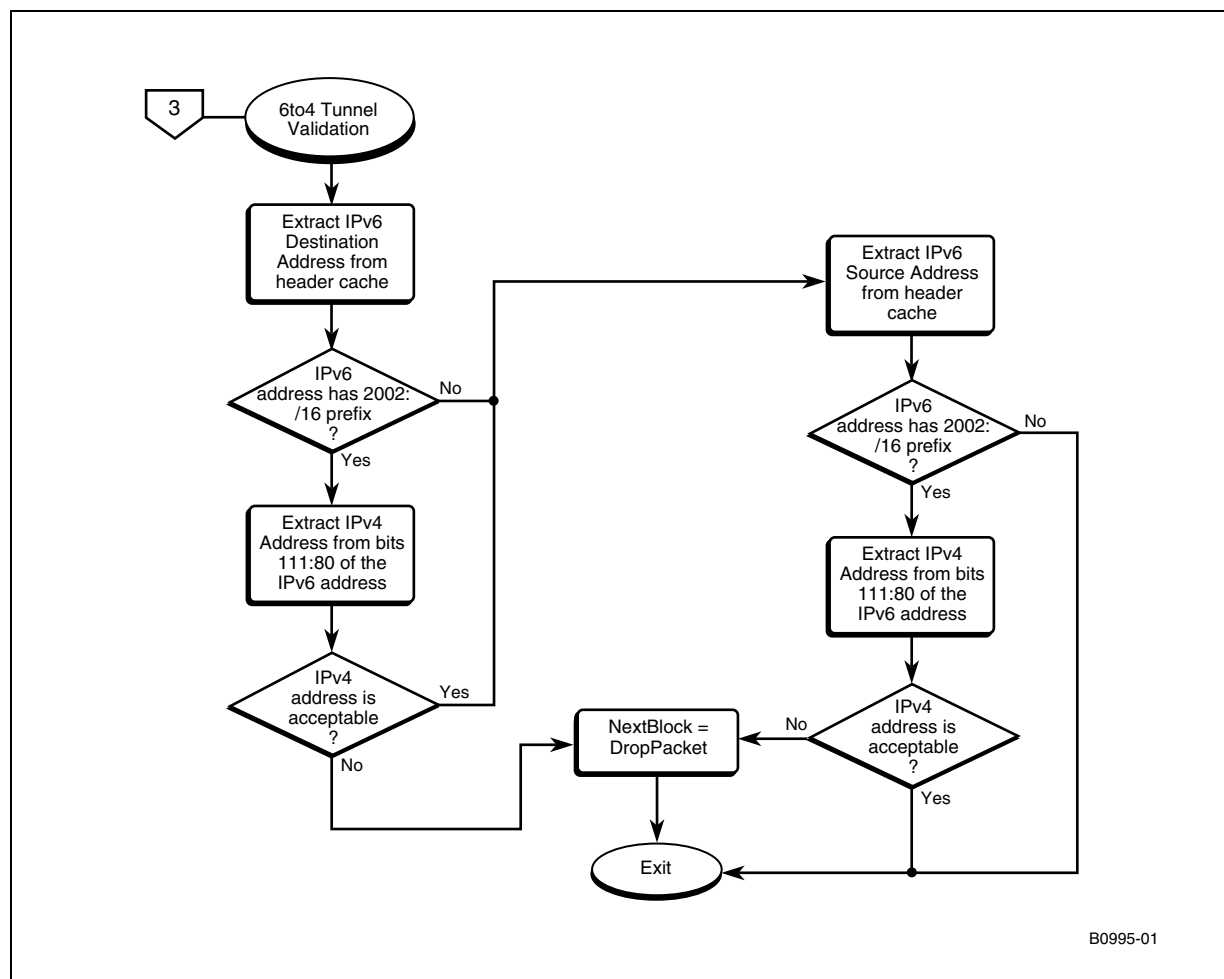
If the tunnel is configured to accept such packets, the embedded IPv4 address is extracted from the IPv4-compatible destination address and compared to the IPv4 destination address. This is required since automatic tunnels must encapsulate to the final destination (see RFC 2893 section 5.6). If the addresses match, the packet's source addresses are validated as described previously. If the source addresses are acceptable, the packet is accepted. Notice that if a packet is accepted for the local node over an automatic tunnel, it is sent to the core component as an exception, since it must be a locally addressed packet (the lookup process that occurred prior to the execution of the decapsulation block matched to a local tunnel endpoint).

26.7.3.3 6to4 Tunneling Verification

Figure 26-6 shows the validation process for 6to4-enabled tunnel endpoints.

The packet's destination IPv6 address is extracted and if the prefix matches the 6to4 prefix (2002:/16), the embedded IPv4 address is extracted and checked for acceptability according to RFC 2893 section 9. The packet's IPv6 source address is extracted and validated in the same manner. To pass validation, the embedded IPv4 source or destination address must not be an IPv4 broadcast, subnet broadcast, multicast, unspecified, loopback, or private address as defined in RFC1918.

Figure 26-6. Process Flow for V6V4-Tunnel-Decap Validation for 6to4 Tunneling

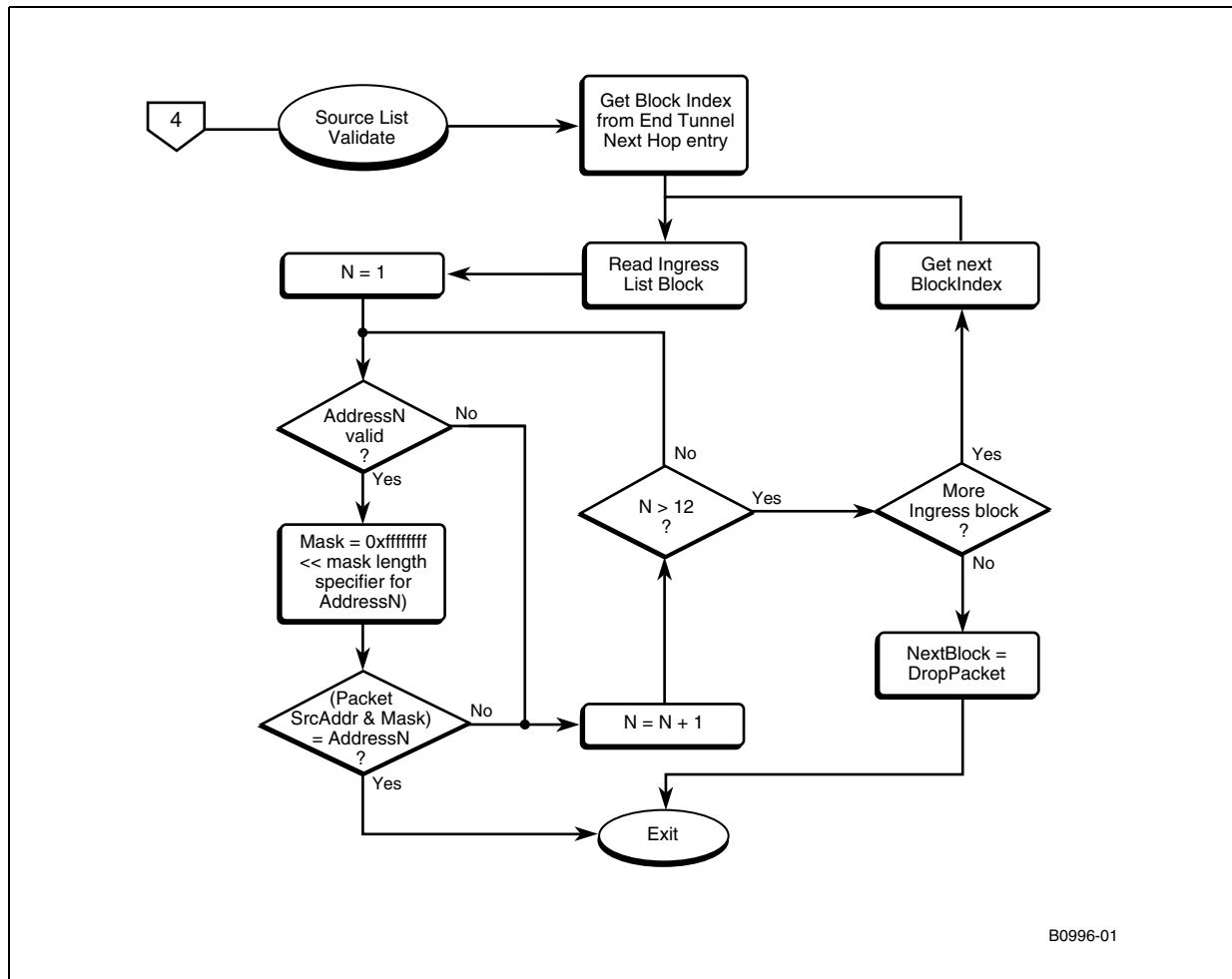


26.7.3.4 Source Address Validation Using the Ingress Source List

The processes for source address validation is described in [Figure 26-7](#).

Ingress source address validation is performed using the ingress source list that was described previously. If the packet's source address matches a prefix in the ingress source list for the destination the packet is accepted for further processing by the IPv6 Forwarder. If not, the packet is dropped.

Figure 26-7. Process Flow for Tunnel Ingress Source Validation



26.8 V6V4-Tunnel-Encap Microblock

26.8.1 Introduction

The V6V4-Tunnel-Encap microblock performs the packet processing required for IPv6 packets that need to be tunneled over IPv4.

Upon entry, the V6V4-Tunnel-Encap microblock assumes that a previous microblock in the group, presumably an IPv6 forwarder, has performed a lookup on the IPv6 destination address of the packet and has set the packet metadata variables `next_hop_id` and `next_hop_id_type`. The variable `next_hop_id` is interpreted as the index of some information that is pertinent to the next stage of packet processing. The variable `next_hop_id_type` specifies the type of information indexed by `next_hop_id`. If the value of `next_hop_id_type` does not indicate IPv6-in-IPv4 tunneling, then the V6V4-Tunnel-Encap block sets `dl_next_block` to `V6V4_NEXT_NO_ENCAP` and no further processing is performed. If the value of `next_hop_id_type` indicates IPv6-in-IPv4 tunneling, then the microblock continues to process the packet.

The value of `next_hop_id` is used to index an array in memory referred to as the tunnel next-hop table for starting tunnel endpoints. Each entry in this array contains the information necessary for the packet to be encapsulated in an IPv4 header. The V6V4-Tunnel-Decap microblock reads the indicated entry and attaches an appropriate IPv4 header to the packet. After encapsulation, the metadata variable `packet_size` is incremented by the size of an IPv4 header and `dl_next_block` is set to `V6V4_NEXT_IPV4`. The metadata variables `header_type` and `buffer_offset` are updated as described in [Section 26.8.1.1, “Header Cache and Metadata Requirements” on page 468](#). Note that `V6V4_NEXT_NO_ENCAP` and `V6V4_NEXT_IPV4` are defined outside the microblock and must be bound to actual block identifiers by the dispatch loop.

26.8.1.1 Header Cache and Metadata Requirements

The V6V4-Tunnel-Encap microblock assumes that the fields in the IPv6 header can be accessed via the `xbuf_extract` API, and assumes that the header is available in the appropriate buffer at the time the block is invoked. The name of this buffer and the offset of the start of the IPv6 header are passed to the microblock as parameters, as described in the next section.

The V6V4-Tunnel-Encap block writes the encapsulating IPv4 header to the appropriate buffer using the `xbuf_offset` API. The name of this buffer and the offset at which the header should begin are passed to the microblock as parameters, as described in the next section.

After encapsulating a packet, the V6V4-Tunnel-Encap block updates the metadata variable `header_type` by ORing the value `V6V4_HDR_TYPE_ENCAP` with the existing value of `header_type`. The value `V6V4_HDR_TYPE_ENCAP` is defined outside of the microblock.

The V6V4-Tunnel-Encap microblock may optionally update the metadata value `buffer_offset` after it encapsulates a packet. A compiler flag controls this option. If the flag `V6V4_UPDATE_OFFSET` is defined, the microblock decrements `buffer_offset` by the size of the IPv4 header. If this flag is not defined, the block does not modify `buffer_offset`. Note this is the same flag that is used to control the modification of `buffer_offset` in the V6V4-Tunnel-Decap block.

26.8.2 Data Structures

26.8.2.1 Tunnel Next-Hop Information

The V6V4-Tunnel-Encap microblock uses the metadata variable `next_hop_id` as an index into an SRAM array of tunnel next-hop structures. Each entry describes the starting endpoint of a tunnel. The format of the tunnel next-hop information for starting tunnel endpoints is specified in [Table 26-10](#).

Table 26-10. Next Hop Fields

LW	Bits	Size	Field	Description
0	31:16	16	Flags	Indicates how encapsulated packets are handled: <ul style="list-style-type: none"> V6V4_ENCAP_FLAGS_VALID (bit 31)—entry is valid. V6V4_ENCAP_FLAGS_AUTO (bit 16)—Automatic tunneling endpoint V6V4_ENCAP_FLAGS_6TO4 (bit 17)—6to4 tunneling endpoint. V6V4_ENCAP_FLAGS_TOS_V6 (bit 18)—TOS field is to be copied from IPv6 traffic class. V6V4_ENCAP_FLAGS_NO_DF (bit 19)—the DON'T FRAGMENT bit should never be set in an encapsulated packet. The remaining bits (30:20) are reserved and must be zero.
0	15:0	16	MTU	MTU for the tunnel.
1	31:0	32	LocalAddress	IPv4 address of the local tunnel endpoint.
2	31:0	32	RemoteAddress	IPv4 address of the remote tunnel endpoint.
3	31:24	8	IPv4 TTL	TTL value to place in encapsulating IPv4 header.
3	23:16	8	TrafficClass	Type of service value. Meaningful only if V6V4_ENCAP_FLAGS_TOS_V6 is not set.
3	15:0	24	Reserved	Must be zero.
4	31:0	32	SubnetBdcast	Subnet broadcast address that cannot be used as packet destination address. Meaningful only if V6V4_ENCAP_FLAGS_AUTO is set.
5-7	31:0	32x3	Reserved	Must be zero.

Each entry in the tunnel next-hop table for ending tunnel endpoints contains eight 32-bit longwords, the first of which contains flags that affect how the packet is encapsulated. Bit 31, `V6V4_ENCAP_FLAGS_VALID`, is set to indicate a valid entry. The flags

V6V4_ENCAP_FLAGS_AUTO and V6V4_ENCAP_FLAGS_OUT_6TO4 are used to determine where to obtain the remote IPv4 address for the encapsulated. [Table 26-11](#) summarizes the behavior for the possible combinations of these flags.

Table 26-11. Tunnel Next-Hop Flags for V6V4-Tunnel-Encap Microblock

Auto	6-to-4	Interpretation
1	0	Automatic tunnel. IPv4 destination address is obtained from the last 32 bits of packet IPv6 destination address.
0	X	Configured tunnel. IPv4 destination address is obtained from RemoteAddress in tunnel next-hop entry.
1	1	6to4 tunnel. IPv4 destination address is obtained from the second two words of packet IPv6 destination address

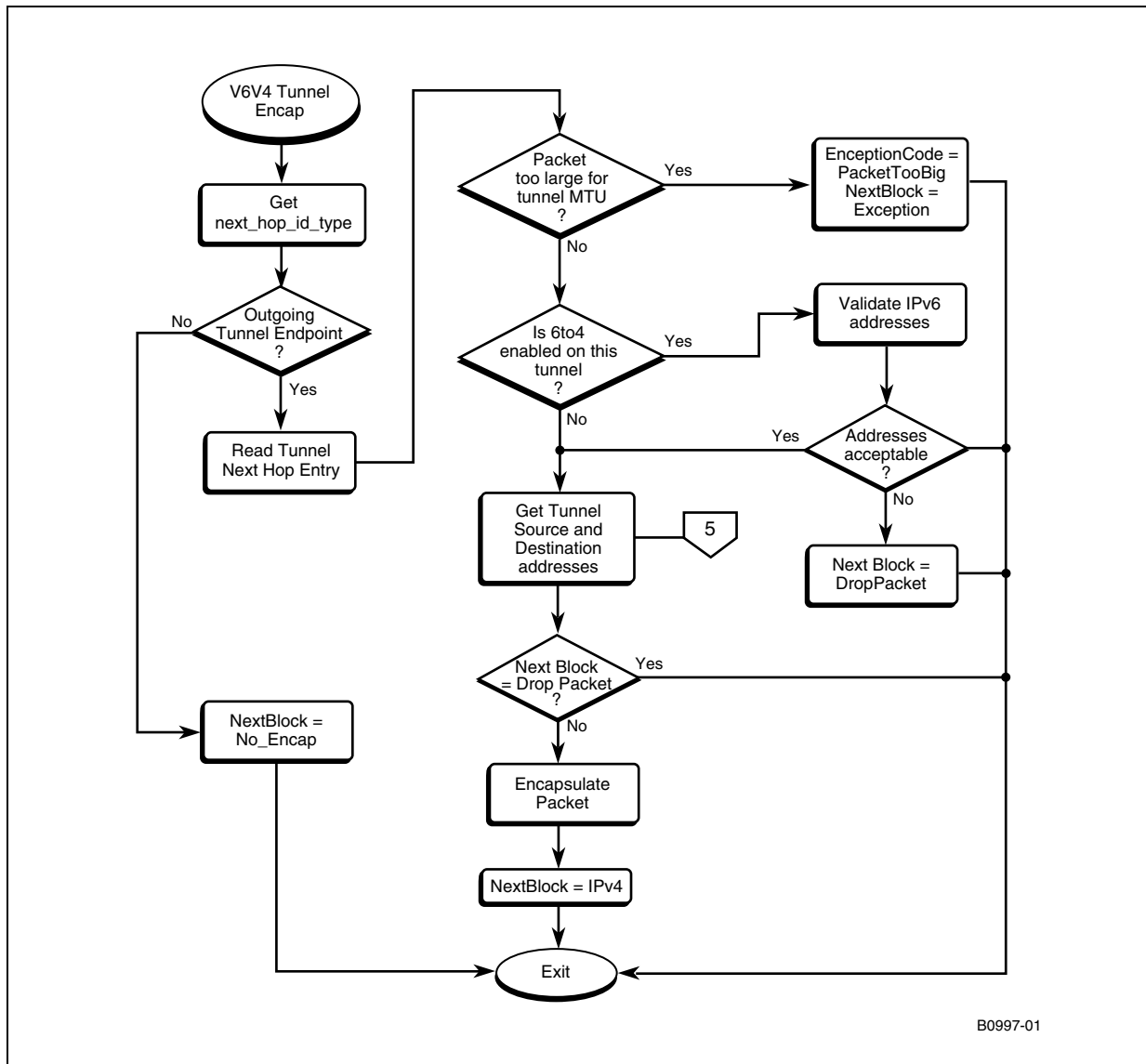
The flag V6V4_ENCAP_FLAGS_TOS_V6 can be used to cause the Type of Service field to be set to the same value as the Traffic Class field in the IPv6 header. If this flag is not set then the Type of Service field is set to the value of the TrafficClass field in the tunnel next-hop entry (bits 23:16 of longword 2). Note that [RFC 2893—Transition Mechanisms for IPv6 Hosts and Routers] specifies that the Type of Service field should be zero, but future RFCs might define different behavior for this field. Setting the flag V6V4_ENCAP_FLAGS_TOS_V6 or setting TrafficClass to a non-zero value overrides the default setting for the Type of Service field.

The flag V6V4_ENCAP_FLAGS_NO_DF is used to control whether the DON'T FRAGMENT flag is set in the encapsulating header. For 6to4 tunnels, this flag should be set, since [6TO4] specifies that encapsulating nodes should not set this flag. For other types of tunnels, the algorithm in [TRMECH], section 3.2 is used to determine whether to set the DON'T FRAGMENT flag. According to [RFC 2893], the DON'T FRAGMENT bit must not be set for tunnels for which the IPv4 path MTU algorithm is not used. The flag V6V4_ENCAP_FLAGS_NO_DF can be set for such tunnels to cause the encapsulation process to avoid setting the DON'T FRAGMENT flag in cases where it otherwise would have.

26.8.3 Process Flow

The flow chart in [Figure 26-8](#) describes the behavior of the V6V4-Tunnel-Encap microblock. As explained previously, the V6V4-Tunnel-Encap block uses the metadata variable next_hop_id_type to determine whether the packet needs to be encapsulated in an IPv4 packet. If encapsulation is required, the tunnel next-hop information is read from memory. Prior to encapsulation, the block performs any required header validation. If the tunnel supports 6to4 traffic (as indicated by the V6V4_ENCAP_FLAGS_6TO4 flag) then the source address is validated according to [RFC 3056: Connection of IPv6 Domains via IPv4 Clouds]. Specifically, the IPv6 source address is checked to determine if it has a 2002::/16 prefix, and if so the embedded IPv4 address is extracted and checked for acceptability. If the extracted IPv4 address is a broadcast, subnet broadcast, multicast, loopback, unspecified or private address as defined by RFC 1918, the packet is dropped. The IPv6 destination address is validated in the same manner.

Figure 26-8. Process Flow for V6V4-Tunnel-Encap Microblock



As explained previously, the V6V4-Tunnel-Encap block uses the metadata variable `next_hop_id_type` to determine whether the packet needs to be encapsulated in an IPv4 packet. If encapsulation is required, the tunnel next-hop information is read from memory. Prior to encapsulation, the block verifies that the packet size is not too large for the tunnel MTU, according to [RFC 2893]. Specifically, packets whose length after encapsulation exceeds the tunnel MTU and whose length also exceeds the minimum IPv6 MTU (1280 bytes) are sent to the core as exceptions.

The V6V4-Tunnel-Encap block also performs any required header validation before it encapsulates the packet. If the tunnel supports 6to4 traffic (as indicated by the `V6V4_ENCAP_FLAGS_6TO4` flag) then the source address is validated according to [RFC 3056]. Specifically, the IPv6 source address is checked to determine if it has a 2002:/16 prefix, and if so the embedded IPv4 address is extracted and checked for acceptability. If the extracted IPv4 address is a broadcast, subnet broadcast, multicast, loopback, unspecified or private address as defined by RFC 1918, the packet is dropped. The IPv6 destination address is validated in the same manner.

After the required validations are performed, the IPv4 endpoint addresses are obtained and if no errors were found the packet is encapsulated. The encapsulation process consists of the following:

- Constructing an IPv4 header in the header cache.
- Incrementing the metadata variables `buffer_size` and `packet_size` by the size of the IPv4 header.
- Decrementing the metadata variable `offset` by the size of the IPv4 header, if the compiler switch `V6V4_UPDATE_OFFSET` is set.
- ORing the metadata variable `header_type` with the application-defined value `V6V4_HDR_TYPE_ENCAP`.

26.8.3.1 Obtaining the Tunnel Endpoint Addresses

The V6V4-Tunnel-Encap microblock obtains the tunnel source address from the `LocalAddress` field in the tunnel next-hop structure, and obtains the tunnel destination address either from the `RemoteAddress` field of the tunnel next-hop structure or from the IPv6 destination address in the packet header. This behavior is shown in the flow chart in [Figure 26-9](#).

All automatic tunneled packets (those for which the destination IPv4 address must be extracted from the IPv4-compatible IPv6 destination address) are subject to the following validations on the extracted IPv4 address (see RFC 3056 section 5.3):

1. The address must not be a broadcast, multicast or subnet broadcast address.
2. The address must not be a loopback or unspecified address.

The validations described in (1) above are always performed on the extracted address. The validations described in (2) are performed only if the compile flag `V6V4_ENCAP_VERIFY_DEST_AUTO` is set, since these checks are typically performed by the IPv4 forwarder.

These checks are not performed if the destination address is extracted from a 6to4 (2002:/16) address, because they were included in the 6to4 validations previously performed as specified in RFC 3056.

26.8.3.2 Creating the IPv4 Header

The IPv4 header is created as specified in RFC 3056 section 3.5. This is summarized in Table 26-12. Figure 26-9 illustrates the process flow for obtaining the IPV4 tunnel endpoint addresses.

Figure 26-9. Obtaining the IPv4 Tunnel Endpoint Addresses

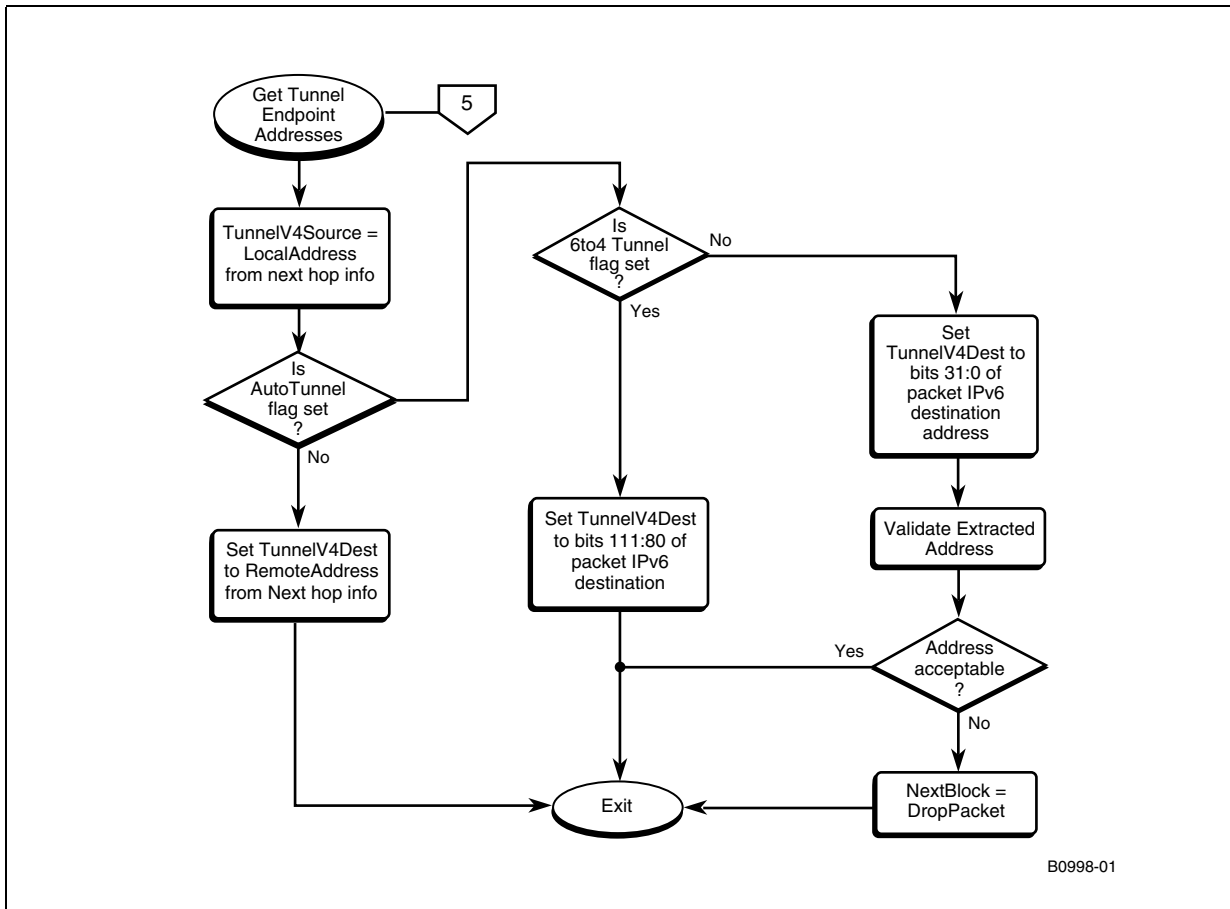


Table 26-12. IPv4 Header Creation for Encapsulated Packets (Sheet 1 of 2)

Field	Value
Version	4
IP Header Length (32-bit words)	5
Type of Service	IPv6 traffic class if tunnel next-hop flag V6V4_ENCAP_FLAGS_TOS_V6 is set OR TrafficClass from next-hop entry
Total Length	Length of IPv6 payload + 60

Table 26-12. IPv4 Header Creation for Encapsulated Packets (Sheet 2 of 2)

Field	Value
Identification	Assuming Bit 15 is most significant bit:High-order 8 bits contains base value that identifies microengine context:Bits 15:11 = ME numberBits 10:8 = thread numberLow-order 8 bits is a sequence number for the packet:Bits 7:1 = sequence numberBit 0 = 0 (indicates Microengine; 1 is used for packets processed in the core component)Base value (bits 15:8) is calculated by the thread when the microblock is initialized; sequence number is initialized to zero and incremented by 2 (to ensure low order bit is always zero) each time a packet is processed.
Flags	If V6V4_ENCAP_FLAGS_NO_DF flag is set, flags are zero. Otherwise Don't Fragment bit is set according to RFC2893 section 3.2.
Fragment Offset	0
Time to Live	Set TTL value in tunnel next-hop entry.
Protocol	41
Header Checksum	Calculated as header is being constructed (sum the 16-bit words in header and take ones complement)
Source Address	LocalAddress from tunnel next-hop structure
Destination Address	Tunnel Destination from next-hop structure RemoteAddress or extracted from packet IPv6 destination address

26.9 XScale Interface

26.9.1 Symbols

The tunneling microblocks share the following data structures with the core components:

- Tunnel next-hop information
- Ingress source block array

The information about the location of these structures are passed to the microengines by dynamically patching the following symbols at load time.

Table 26-13. Symbols

Symbol	Description
_TUNNEL_DECAP_NEXTHOP_SRAM_BASE	Start address of ending tunnel next hop entries in SRAM.
_TUNNEL_ENCAP_NEXTHOP_SRAM_BASE	Start address of starting tunnel next hop entries in SRAM.
_TUNNEL_INGRESS_LIST_SRAM_BASE	Start address of ingress list array in SRAM, if used.
_TUNNEL_INGRESS_LIST_SRAM_TRIE_BASE	Start address of ingress list trie table base in SRAM, if used

26.9.2 Exception Codes

Table 26-14 identifies the exception codes generated by IPv4 Forwarder block that are passed along with the packet to the Intel XScale® core counterpart.

Table 26-14. Exception Codes

Exception code	Value	Description
V6V4_ENCAP_EXCEPTION_MTU	0x90	Packet too big for tunnel path MTU.
V6V4_DECAP_EXCEPTION_AUTO	0x81	Decapsulated packet received on an automatic tunnel is for local delivery.
V6V4_DECAP_EXCEPTION_LOCAL	0x80	Local packet received.
V6V4_DECAP_EXCEPTION_FRAGMEN T	0x82	Tunneled packet was fragmented and must be reassembled before decapsulation

26.10 Performance Analysis

The IPv6-IPv4 Tunneling microblock runs as a functional pipeline on four microengines. The I/O latency available for the functional pipeline, to handle a ethernet min packet is $141 * 8 * 4$ cycles.

Table 26-15 and Table 26-16 summarize the performance analysis for the Tunneling block.

Table 26-15. Cycle Count Analysis

Cycle Count Analysis	Values
Available cycle count for 118 byte min Ethernet packet	$(141 * 4)$ cycles (assuming 600 MHz IXP2400)
Estimated cycle count for min packet based on flow chart to Decapsulate automatic tunneled packet	70
Estimated cycle count for min packet based on flow chart to Decapsulate packet with ingress source validation (linear list assuming maximum of 12 entries in ingress source list), not counting load of IPv6 header	90 (best case) 160 (worst case)
Estimated cycle count for min packet based on flow chart to Decapsulate packet without ingress source validation	70
Estimated cycle count for min packet based on flow chart Encapsulate packet	100

Table 26-16. I/O Latency Analysis for Min Packet

I/O latency analysis for min packet	Values
Available I/O latency for min packet	$4 * 141 * 8$ cycles
Decapsulate automatic tunneled packet	4 bytes

Table 26-16. I/O Latency Analysis for Min Packet

I/O latency analysis for min packet	Values
Decapsulate packet with ingress source validation (linear list assuming maximum of 12 entries in ingress source list), not counting load of IPv6 header	2 (1 LW, 16 LW)
Decapsulate packet without ingress source validation	1 (1 LW)
Encapsulate packet	1 (8 LW)

IPv6 To IPv4 Translation Microblock 27

This section describes the design and implementation of the IPv6-IPv4 Translation microblock.

27.1 Overview

The IPv4-to-IPv6 transition mechanism supports the Network Address Translation-Protocol Translation (NAT-PT) mechanism. NAT-PT allows end hosts in a v6 realm to communicate with end hosts in a v4 realm and vice versa. An address realm is a network domain where the entities have unique network addresses such that datagrams can be routed to them [RFC2663]. A NAT-PT device straddles different address realms and allows the end hosts in the different realms to communicate via a technique known as transparent routing [RFC2663]. Transparent routing involves changing the header contents of a datagram in order to route datagrams between disparate address realms. Specifically NAT-PT changes the addresses and the protocol—that is, the header format, to make a datagram valid and routable in the address realm into which it is sent.

The NAT-PT does not mandate dual-stack—that is, IPv4 as well as IPv6 protocol support, or special-purpose routing requirements (for example, tunneling support) on end nodes. It also does not require a host to be assigned a permanent IPv4 address. Thus from the host's point of view, an IPv6-only host can communicate with a host that only supports IPv4, just like it would communicate with any other IPv6 host.

Translation of headers on-the-fly is, however, a time consuming process and hence, a fundamental assumption of NAT-PT is that it be used only when no other means (like native IPv6 or tunneling mechanisms) of communication is possible.

The following sections describe the various options of NAT-PT.

27.1.1 Traditional (Outbound) NAT-PT

Traditional NAT-PT supports unidirectional sessions, initiated by hosts within the v6 realm. That is, from the point of the v6 realm an “outbound” session is allowed, which allows hosts within the v6 realm to access hosts in a v4 realm. There are two variations to traditional NAT-PT:

- Basic NAT-PT—Network Address Translation-Protocol Translation
- NAPT-PT—Network Address Port Translation-Protocol Translation

27.1.1.1 Basic NAT-PT

Basic NAT-PT involves setting aside a block of v4 addresses for translating v6 host addresses. A v4 address is bound to a v6 address at least for the duration of a session. For outbound packets, source IP address & IP/TCP/UDP checksum fields are translated. For inbound packets, destination IP address & checksum are translated.

27.1.1.2 NAPT-PT

Network Address Port Translation-Protocol Translation (NAPT-PT), extends the notion of translation to transport identifiers (for example, TCP/UDP port numbers), thereby allowing transport identifiers from various v6 hosts to be multiplexed into transport identifiers of a single assigned v4 address. Thus, NAPT-PT allows a number of v6 hosts to share a single v4 address (a very useful property considering the problem of v4 address space exhaustion).

Note: NAPT-PT can be combined with Basic-NAT-PT so that a pool of addresses can be used along with port translation.

27.1.2 Two-way (Bi-directional) NAT-PT

Two-way NAT-PT supports sessions initiated from hosts in v4 networks and in v6 networks. Hosts in the v4 network initiate sessions to v6 nodes using their domain names. An entity called a DNS-Application Level Gateway (ALG) is needed to support two-way NAT-PT. The DNS-ALG snoops on DNS packets traversing the v6 and v4 realms and converts v6 addresses in the packets to their v4 address bindings.

27.1.3 Application Level Gateway (ALG)

Some applications contain network addresses in payloads. If NAT-PT translates the addresses in the network headers, these applications will not be able to function correctly. NAT-PT is application unaware and does not look at the payload. Hence, to allow such applications to work correctly when a v6 node is communicating with a v4 node, an application-specific entity called an Application Level Gateway (ALG) is defined. ALGs are specific to an application, aware of the payload details and take on the task of translating the payload. They need to coordinate with NAT-PT to obtain the appropriate address bindings.

27.2 Assumptions

The following assumptions are made in the design and implementation of the IPv6-IPv4 translation microblock:

- The translation microblock is a transform microblock. It must run as part of a microblock group that includes the source and sink microblocks.
- The packet meta-data is available via the dispatch loop macros described in [IXAPF].
- The packet header can be accessed via the xbuf API provided with the Intel® Internet Exchange Architecture Software Development Kit (Intel® IXA SDK). The microblock assumes that these xbuf macros can operate on transfer registers, general-purpose registers (GPRs) and local memory.
- The packet buffers passed in to it are large enough to contain the main IP header (40 bytes in case of IPv6 and 20 bytes for IPv4)
- The packet buffers in DRAM have some headroom. Based on this assumption, when the translated header is longer than the original packet, the start offset (in the meta-data) of the packet is moved back, such that when the header is flushed back it lies right before the payload.

The primary goal of this microblock is to achieve maximum performance. Hence, the following assumptions apply:

- The microblock doesn't perform any error checking on the data structures shared with the XScale core. The XScale core code has to populate the structures with correct values. Otherwise, the results are unpredictable.
- The threads executing this microblock will not maintain strict ordering (because of the different amounts of processing and I/O required per packet). As a consequence, the packets can get out of order by the end of the microblock. Hence, it is the responsibility of the dispatch loop using this microblock to ensure packet order around this block.
- The packet ordering is not maintained when there are exception packets, as these packets are sent to the XScale core, and the microblock (or the dispatch loop) has no way of ensuring the packet ordering on the XScale core.
- The XScale core code has to create and maintain/update the shared data structures. The microblock simply reads and processes the information. Only the counters are different, as they are updated by the microblock.
- All address and offsets specified in this section are byte addresses.
- The values in the data structures shared by this block are in network-byte order (big-endian).

27.2.1 Dependencies

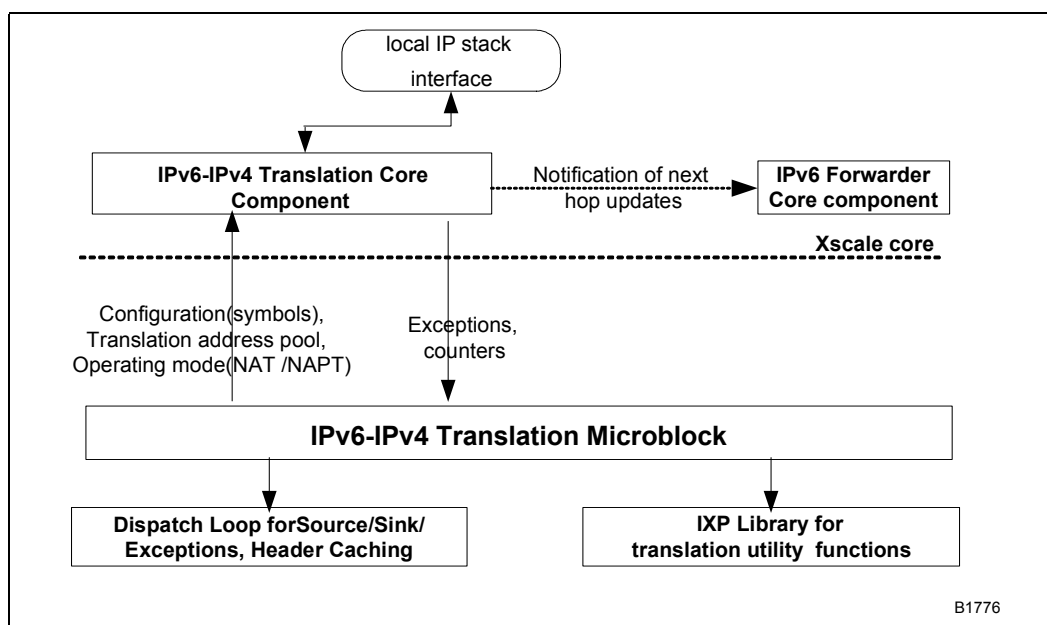
The translation microblock requires:

- the microblock preceding it in the dispatch loop to remove the L2 header, so it can work directly with the L3 header
- the IPv6 and IPv4 forwarders follow so the translated packet can be forwarded

There are no other restrictions on where the microblock can be placed in the dispatch loop of the packet processing stage.

Figure 27-1 illustrates the other components the translation microblock depends on during its operation. The inward arrows indicate dependency on the IPv6-IPv4 translation microblock. The outward arrows show what the microblock expects from other modules.

Figure 27-1. Dependencies of the IPv6 Translation Microblock and Related Modules



The IPv6-IPv4 Translation Microblock expects configuration parameters, the pool of addresses to use for translation to be supplied by its core component. The core component also specifies the mode of operation (NAT or NAPT) of the microblock. The microblock expects standard macros from other components for sourcing and sinking packets, caching headers and sending exceptions to the core. It uses helper routines from the IXP library for performing packet translation.

The Translation microblock requires a local memory buffer, primarily for caching IPv6 extension headers and transport headers if needed. This buffer is also used as a work area for storing other parts of the packet that need to be read by the microblock. Since the microblock is unaware of the local memory map for the application, it depends on the application code to define the required buffer parameters. Table 27-1 shows symbols the microblock assumes are defined.:

Table 27-1. Buffer Parameters Defined

Symbol	Description
LM_NATPT_WORK_BUF	Name of the xbuf buffer in local memory
LM_NATPT_WORK_BUF_BASE	Base address of the buffer in local memory
LM_NATPT_WORK_BUF_SIZE	Size of the buffer in bytes
LM_BUF_LWSIZE_NATPT_WORK	Size of the buffer in long-words
LM_BUF_OFFSET_NATPT_WORK_BUF	Offset of the buffer in bytes. (The xbuf API allows a pool of buffers to be created. When more than one buffer is defined in the pool, this value indicates the offset of a specific buffer in the pool. Currently only one buffer is defined and the offset can be set to 0 by the application)

The Translation microblock is responsible for sending exceptions to the Translation core component and maintaining counters for the packets translated. The Translation core component can pick up these counters when required to gather statistics for the translation module.

The Translation core component depends on the IPv6 forwarder core component for providing notifications when the forwarding table is updated. It interacts with the Stack interface sending appropriate exception packets or locally destined packets up the stack. The stack driver sends down locally generated packets to the Translation core component.

27.2.2 Configuration Options

27.2.2.1 Build Switches

Table 27-2 identifies compile time build switches, which can be turned on or off as required for a specific application.

Table 27-2. Compile Time Build Switches

Symbol	Description
TRANSLATE_TRAFFIC_CLASS	Enables setting of TOS & Precedence fields of a translated IPv4 packet based on the traffic class field in the original IPv6 packet. If not set, these fields are set to 0.
NATPT_V6_OPTIMIZE_LOOKUP	Enables optimization where the address lookup for translated v6 packets is avoided.
NATPT_DEBUG_COUNTERS	Enables counters to gather NATPT statistics.
NO_GPR_OVERFLOW	Enables some buffers in general-purpose registers instead of in local memory. These buffers were allocated in local memory as the application (that is, the Ethernet IPv6) results in a GPR overflow
NO_CODESTORE_OVERFLOW	Enables “in-lining” some function calls, that is, macro calls are made instead of jumping to a specific address. The Ethernet IPv6 application overflows the codestore. Hence some of the code is associated with <code>ifdef</code> under this switch. Specific applications can turn on this switch for faster execution, provided code store availability is not an issue.
ENHANCED_XBUF_API	Switch this flag on if enhanced <code>xbuf_copy</code> API support is available.

27.2.3 Default Configuration

By default all the build switches listed in Table 27-2 are turned OFF.

27.2.4 RFC Compliance

The translation microblock performs the packet translation based on the recommendations in RFC 2766, RFC 2765 and RFC 1631.

27.2.5 Statistics Counters

The NAT-PT microblock maintains some counters to indicate how the incoming packets are handled by the microblock. The Translation core component allocates the space for these counters and patches the base address to be used by the microblock. This allows the core component to inspect the microblock statistics when needed. [Table 27-3](#) lists the counters and their offsets from the base address.

Table 27-3. Counters and Offsets from the Base Address

Counter	Offset from Base	Description
NATPT_RECV	0	Count of packets received by NAT-PT
NATPT_PASS_THROUGH	4	Count of packets that were not translated by NAT-PT, but simply passed on to the next microblock
NATPT_TRANSLATED_TO_V6	8	Count of packets that were translated to IPv6
NATPT_TRANSLATED_TO_V4	12	Count of packets that were translated to IPv4
NATPT_SENT_TO_CORE	16	Count of packets for which an exception was raised and the packet sent to the core

27.3 Overview of NAT-PT Microblock

NAT-PT translation is done on a session basis. There are three distinct phases of operation that a NAT-PT implementation must support in order to implement transparent routing. These phases result in the creation, use and removal of state for the sessions requiring translation.

- **Address Binding**—In this phase, a session requiring translation is identified and a state is created to support the translation process. This involves storing an association (“binding”) between an IPv6 address and an IPv4 address. The operation is performed on session initiation packets, that is, a session initiation packet is an IPv6 packet with a v4-mapped-v6 destination address. It is the very first packet in a packet stream from an IPv6 source to an IPv4 destination. Refer to [Section 27.4.3, “Process Flow” on page 490](#) for details on identifying a session initiation packet.
- **Address lookup and translation**—In this phase, packets are examined to determine if their contents (address, transport ids) match the stored state. Packets that match are classified as part of that session and the stored state is used to modify their headers so that they are valid in the other realm. This operation is performed on all the packets belonging to the session.
- **Address unbinding**—The last phase in address translation involves determining when a session is terminated and clearing up the state stored for the session. This operation involves timing out the stored state when the packets belonging to a session are no longer seen. When a session terminates, the session address is released for use by other sessions.

The NAT-PT functionality is distributed between a core component and another on the ME (microblock). The separation of functionality between the core component and the microblock is guided primarily by the speed and frequency at which a certain operation needs to be performed.

The following describes the separation of functionality in the current implementation:

- Address binding—The microblock is responsible for identifying session initialization packets. Once such a packet is received, the microblock passes it to the Translation core component for processing. The Translation core component is responsible for the actual binding and state creation. The core component creates the bindings in a well-known location accessible by the microblock. This separation ensures that the microblock is not tied up for too long creating the address binding. Also, binding only needs to be performed at the start of a session, and hence, the performance hit for operation in the core component is acceptable.
- Address lookup and translation—The microblock looks at all received packets, compares with existing bindings, and if a packet is found to be part of a session, translates it. The microblock also handles fragments, as long as it can translate the fragment itself. However, some packets need to be reassembled before they can be translated. This is a time consuming process since all fragments must be received prior to the translation. Such fragments are sent up to the Translation core component to handle.
- Address unbinding—This operation is handled by the core component. The microblock assists the core component in finding when a session is terminated, as described in [Section 27.4.2.1, “NAT Table” on page 484](#).

Most of the Application Level Gateway (ALG) (see [Section 27.1.3, “Application Level Gateway \(ALG\)” on page 478](#)) functionality is handled by the Translation core component. However, in order for the specific application packets to reach the core component, there needs to be some entity in the fast path, which looks at and identifies these packets. Support for ICMP, FTP and DNS packet translation is currently targeted. Macros that identify the appropriate ICMP, FTP and DNS packets and send them up to the core component need to be defined. Although these macros are called by the NAT-PT microblock, they are not considered to be part of the NAT-PT implementation. Rather, they should be viewed as ALG hooks. The NAT-PT microblock defines the input and output parameters to these macros. An area in the code where ALG hooks are invoked is specified and the hooks for the currently supported ALGs is called from this area. Based on the output of each hook, the NAT-PT microblock decides whether a packet should be sent up to the core component. Defining an interface and indicating the area where they are invoked allows addition of new ALGs. When a new ALG has to be added, the hook for that ALG is defined and invoked from this area.

27.4 Data Structures

This section describes the data structures used by the NAT-PT microblock. The data structures used to interact with the microblock, that is, pass in data and receive the output of the microblock, are explained. This is followed by a description of the data structures used by NAT-PT to translate packets.

27.4.1 Header Cache and Meta-Data Requirements

NAT-PT assumes that the fields in the IP header can be accessed via the `xbuf_extract` API, and assumes that the header is available in the appropriate buffer at the time the block is invoked. The name of this buffer and the offset of the start of the IP header are passed to the microblock as parameters. In addition, the buffer and the offset at which the translated header should be written are also passed as parameters.

Port translation requires NAT-PT to look into transport headers. If the transport header is not in the header cache passed, NAT-PT loads the header into transfer registers by issuing a read from DRAM (The DRAM buffer address is available in the meta-data). NAT-PT then modifies the header as needed and flushes the modified header back to DRAM.

Note: The transport headers are not cached. This is based on the assumption that transport headers are unlikely to be read-modified-written by numerous microblocks.

Translated packets might be larger than the original packets. The headroom in DRAM buffers (Section 2.2, “[Packet Meta Data \(Buffer Descriptor\)](#)” on page 58) is used to accommodate the growth in the packet size. The offset in the buffer where the packet actually begins is maintained in packet meta-data. NAT-PT updates the values in the meta-data to correctly reflect the offset at which the translated packet lies. For instance, if the translated IP header is larger than the original header by some amount, NAT-PT subtracts the offset value in meta-data by the same amount. This ensures that when the header is flushed back, it lines up correctly with the rest of the packet data

27.4.2 Translation Data Structures

NAT-PT uses the state stored in two tables to translate packets. One of these tables holds NAT specific state, while the other holds state useful for the NAPT operation. The NAPT table is created only when the compile time flag, NAPT_ON, is set. The tables are maintained by the core component. The format of each of the tables, the indexing mechanism and how the data is used is explained below.

27.4.2.1 NAT Table

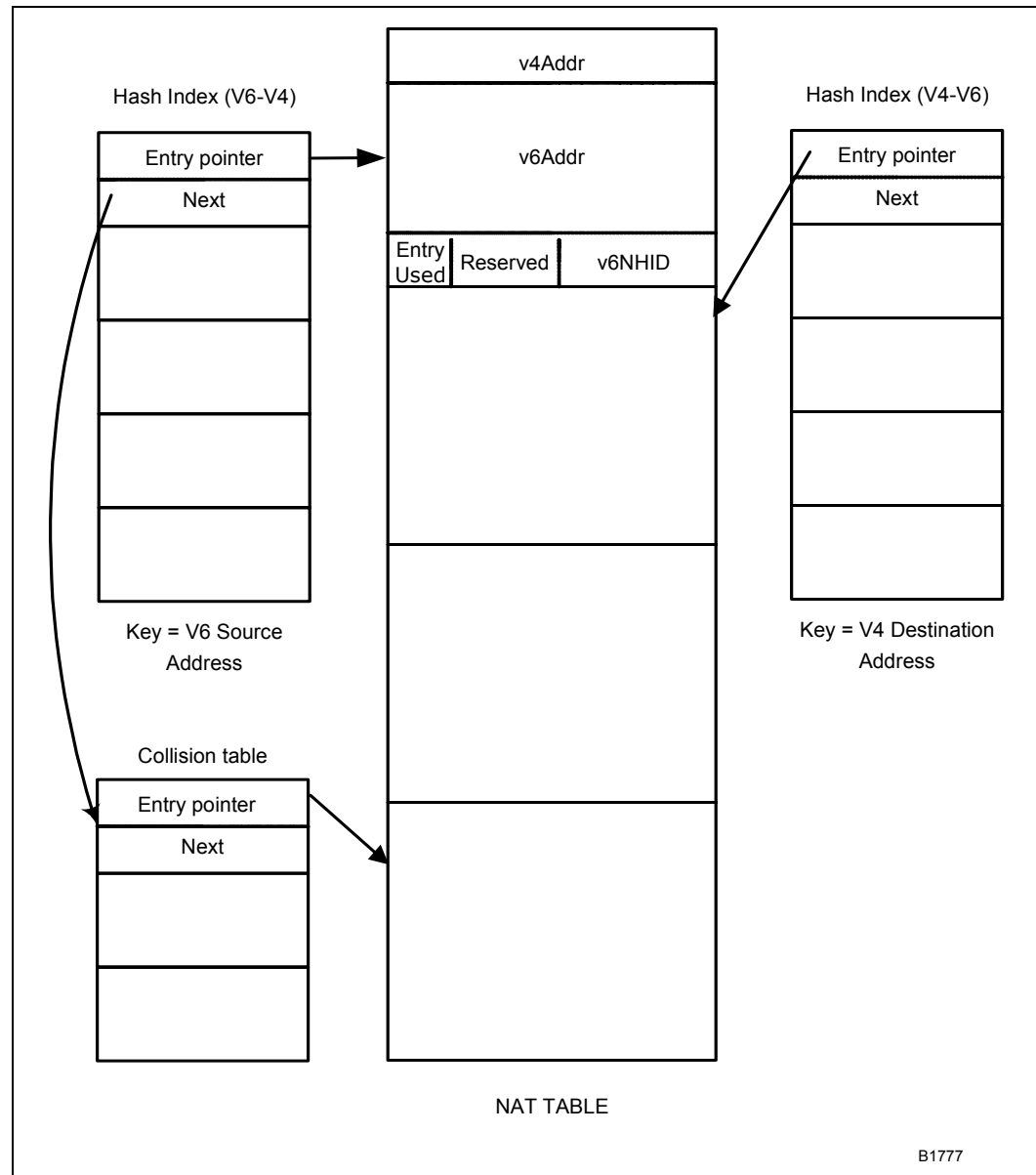
State in [Table 27-4](#) is used when the microblock needs to translate a packet by modifying the address only.

Table 27-4. NAT Table Data Format

LW	Bits	Size	Field	Description
0	31:0	32	v4Addr	The IPv4 address to use when translating an IPv6 packet with the following source v6 address.
1-4	31:0	32x4	v6Addr	The IPv6 address for which a binding has been created.
5	31	1	entryUsed	Used for determining when to unbind the state. Whenever an entry is accessed by the microblock this bit is set. The Translation core component periodically checks this bit and sets the bit to 0. If the bit is still 0 the next time the Translation core component checks this entry, it means the entry has not been used since the last check, implying the session for which the binding was created is terminated and the entry may be freed.
5	30:16	15	Reserved	
5	15-0	16	v6NHID	The next hop ID one would get if a lookup were done on the IPv6 address stored in this entry. Used to avoid a lookup that would otherwise be needed to forward the translated v6 packet. Note: It is the responsibility of the IPv6 forwarder core component to inform the NAT core component of any updates to the forwarding tables.

Figure 27-2 illustrates the main NAT table and the indices used to access it.

Figure 27-2. NAT State



27.4.2.1.1 NAT Table Indexing Mechanism

The state in the NAT table needs to be accessed for translating IPv6 packets entering a IPv4 domain, as well as for IPv4 packets entering a IPv6 domain. For IPv6 packets the IPv6 source address is used as a key to index to the appropriate state, while the IPv4 destination address is used for translating IPv4 packets. That is, to get to the same state, different keys are used to index based on the direction in which the translation occurs. This requirement is supported by having a separate hash index table for each direction.

27.4.2.1.2 V6-V4 NAT Hash Index

Table 27-5, “V6-V4 NAT Hash Index” (the left table in Figure 27-2) is used while translating v6 packets. To get to a particular entry in this index table, a hash is applied on the v6 source address. Each entry in this table holds a pointer (Entry pointer) to a row in the NAT table. In case the hash value of another v6 address ends up on the same entry, the new entry is stored in an area set aside for collision entries (Collision table in Figure 27-2) and the Next field is updated with its address, which creates a chain of collision entries.

Table 27-5. V6-V4 NAT Hash Index

LW	Bits	Size	Field	Description
0	31:0	32	Entry pointer	Points to an entry in the NAT table.
1	31:0	32	Next	In case of a collision this pointer is used to chain the collision entries.

27.4.2.1.3 V4-V6 NAT Hash Index

This index table (the table on the right-hand side in Figure 27-2) is used while translating v4 packets. To get to a particular entry in this index table, a hash is applied on the v4 destination address. Other than this, the format of an entry in this table is the same as Table 27-5.

Note:

1. The format of a collision table entry (in Figure 27-2) is exactly the same as an entry in the Index table.
2. Suppose a hash operation on a packet results in an entry that is part of a collision chain (that is the “Next” pointer is not NULL). In order to find the exact entry in the chain that matches the packet, the Entry pointer is used to get to a row in the NAT table. The fields in the NAT table row are then compared with the corresponding fields in the packet. If they do not match, the search continues down the chain using the Next pointer in the hash index entry.

Separating the hash index from the main state results in the need for an extra SRAM access whenever an entry has to be retrieved. However, the advantage of this approach is that the same state table can be used for translation in either direction, which saves a lot of space.

27.4.2.2 NAPT Table

State in Table 27-6 is used when the microblock needs to translate a packet by modifying the address as well as the transport identifier.

Table 27-6. Data Format

LW	Bits	Size	Field	Description
0-3	31:0	32x4	v6Addr	The IPv6 address and port for which a binding has been created.
4	31:16	16	PortInV6	
4	15:0	16	PortInV4	The port that is bound to the v6 address port combination. This is the transport identifier that is used in the payload of the v4 packet.

Table 27-6. Data Format

LW	Bits	Size	Field	Description
5	31	1	entryUsed	Used to determine when to unbind the state. Whenever an entry is accessed by the microblock this bit is set. The core component periodically checks this bit. Whenever it does this it sets the bit to 0. If the bit is still 0 the next time the core component checks this entry, it means the entry has not been used since the last time the Translation core component checked this entry, implying that the session for which the binding was created has terminated and the entry can be freed.
5	30:24	7	Reserved	
5	23-16	8	Protocol	The transport protocol for which this binding has been created.
5	15-0	16	v6NHID	The next hop ID one would get if a lookup were done on the IPv6 address stored in this entry. Used to avoid a lookup that would be needed to forward the translated v6 packet.

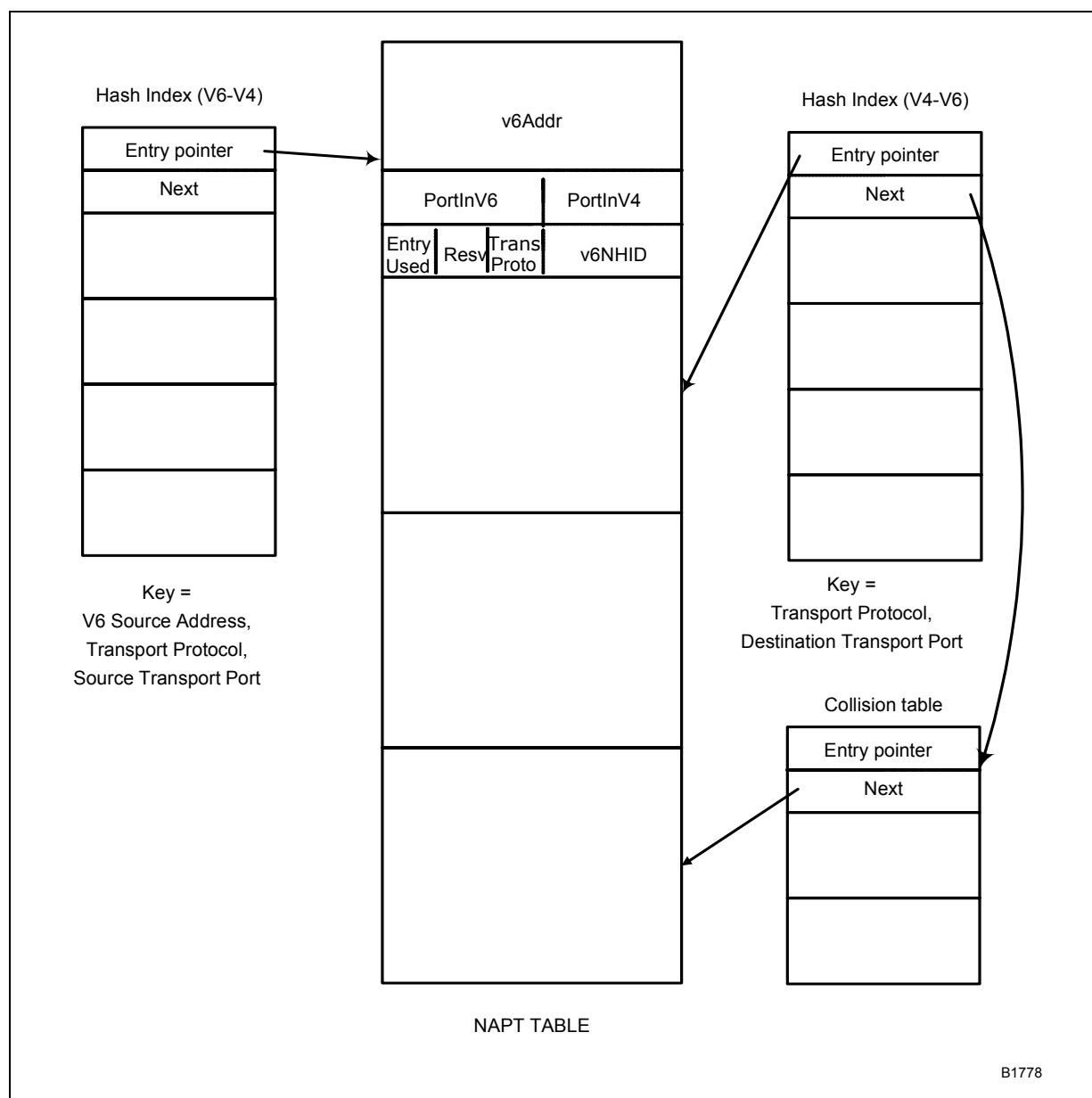
Note: Only one v4 address is used for NAPT-PT. The Translation core component sets the address to use in a well-known memory location

Table 27-7. IPv4 Address Used for NAPT-PT

LW	Bits	Size	Field	Description
0	31:0	32	NAPT v4 address	The IPv4 address to use for NAPT-PT translation

Figure 27-3 illustrates the main NAPT table and the indexes used to access it.

Figure 27-3. NAPT Table and Indexes Used for Access



27.4.2.2.1 NAPT Table Indexing Mechanism

The state in the NAPT table needs to be accessed for translating v6 packets entering a v4 domain as well as for v4 packets entering a v6 domain. For v6 packets the v6 source address, transport protocol and the source transport identifier are used as a key to index to the appropriate state; while the destination transport identifier and transport protocol are used for translating v4 packets. Essentially, to get to the same state, we index using different keys based on the direction in which the translation is occurring. This requirement is supported by having a separate hash index table for each direction.

27.4.2.2.2 V6-V4 NAPT Hash Index

This index table (the left table in [Figure 27-3](#)) is used while translating v6 packets. To get to a particular entry in this index table, a hash is done on the v6 source address, transport protocol and source transport identifier. Each entry in this table holds a pointer (Entry pointer) to a row in the NAPT table. In case the hash value of another address, port, protocol tuple ends up on the same entry, the new entry is stored in an area set aside for collision entries (Collision table in [Figure 27-3](#)) and the Next field is updated with its address, which creates a chain of collision entries.

Table 27-8. V6-V4 NAPT Hash Index

LW	Bits	Size	Field	Description
0	31:0	32	Entry pointer	Points to an entry in the NAPT table
1	31:0	32	Next	In case of a collision this pointer is used to chain the collision entries.

27.4.2.2.3 V4-V6 NAPT Hash Index

This index table (the table on the right-hand side in [Figure 27-3](#)) is used while translating v4 packets. To get to a particular entry in this index table, a hash is applied on the transport protocol and destination transport identifier. Other than this, the format of an entry in this table is the same as the v6-v4 hash index table explained [Table 27-8](#).

Note:

1. A collision table entry is exactly the same as an entry in the Index table.
2. Separating the hash index from the main state results in the need for an extra SRAM access whenever an entry has to be retrieved. However, the advantage of this approach is that the same state table can be used for translation in either direction, thereby saving a lot of space.

Design note—This microblock is not integrated with the forwarders using the NH-type to identify if the packet needs to be translated for the following reasons:

1. Translation entries are end-host addresses. Adding host routes into the forwarding table could impact the performance of the forwarders (the presence of host routes requires LPM to look at more bits in the address before it can determine the longest matching prefix).
2. Also, since the length of the translation address is fixed (either 32-bits or 128-bits) hash can be applied to locate the associated state.

Separating From forwarders	Adding translation entries to the forwarding table
Separating from the forwarders adds an extra hash to every packet. For example, if 10 out of 100 packets are to be translated, 90 hashes would be applied that are of no use.	Including translation entries with the forwarder entries adds an extra SRAM access for every packet. If there is a 64-bit prefix added by the routing protocol, and a 128-bit translation entry is added, all 100 packets require more than 64 bits look-up. In the best case one more SRAM read (to compare the next 8 bits) for all 100 packets.

27.4.3 Process Flow

The flow charts (figures 27-4, 27-5, 27-6, 27-7, and 27-8) show the high-level process flow of the microblock. Figure 27-4 shows how the microblock determines if a packet can be translated and how a session initialization packet is identified. Figures 27-5, 27-6, 27-7, and 27-8 explain the details of the translation process.

The microblock begins by identifying if the packet is inbound (v4 realm to v6 realm) or outbound (v6 realm to v4 realm). If the packet is IPv6, and the destination address is a v4-mapped-v6 address, it must be translated. The microblock now checks if it is an ICMP packet or if any of the ALG hooks match it as an ALG packet. If so, the packet is sent to the Translation core component as an exception. If not, the microblock checks to see if a binding for the session (as identified by the source fields) to which this packet belongs was created earlier. If so, the packet is translated as explained below. If not, then the packet is treated as a session initialization packet. An exception is raised and the packet is sent to the Translation core component for processing. The Translation core component can now create a binding for this session. If the destination v6 address is not a v4-mapped-v6 address, it is a regular v6 packet and is simply sent on by the microblock without any translation.

If the packet is IPv4, the microblock checks to see if a binding for the session (as identified by the destination fields) to which the packet belongs, exists. If so, the packet must be translated. Like in the IPv6 path, an ICMP or a packet for an application that needs ALG translation is sent up to the Translation core component as an exception. Otherwise, it is translated as explained in the subsequent flowcharts. If no binding exists, it means it is a regular v4 packet and is simply sent on to the next microblock in the chain.

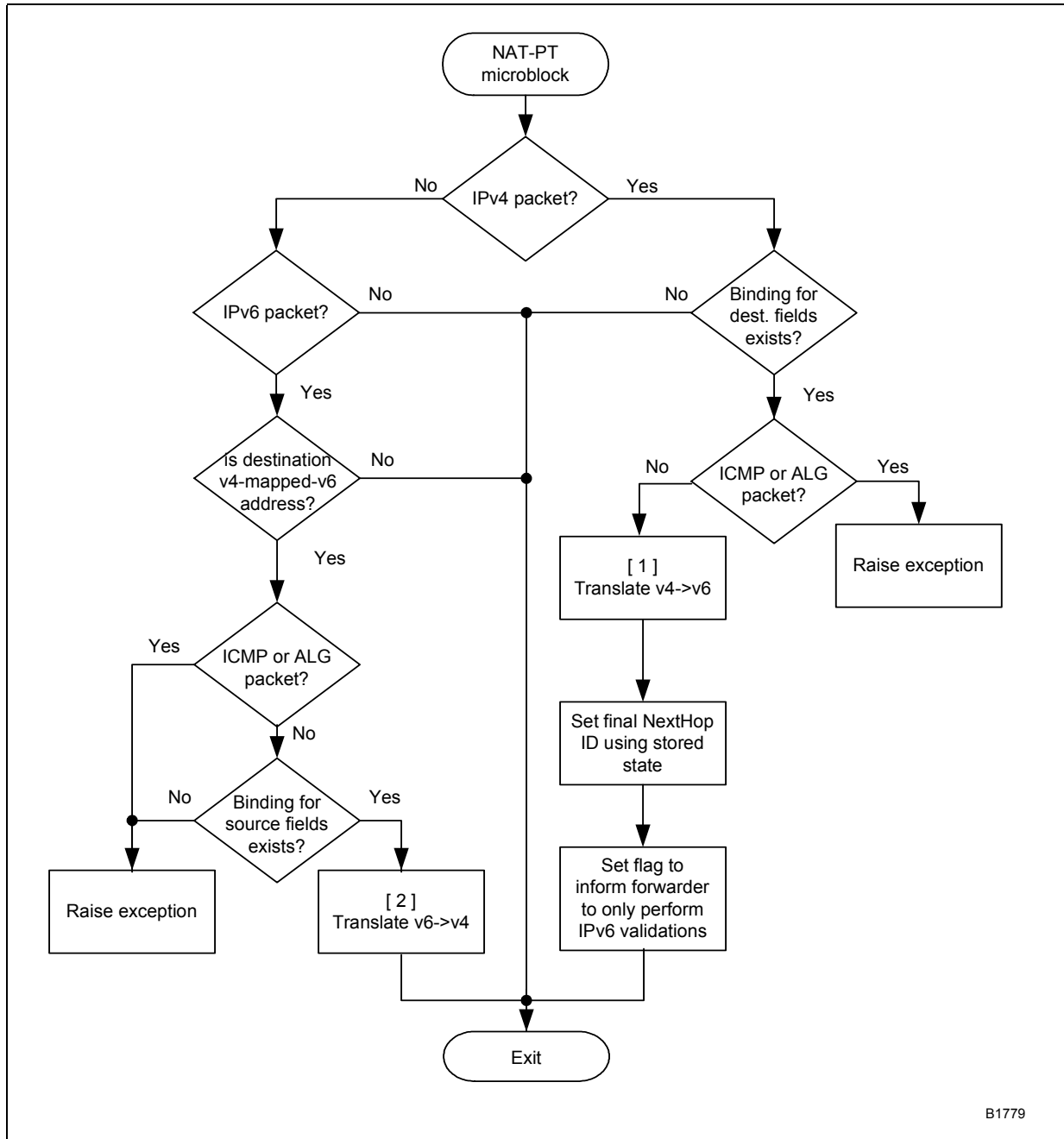
In order to check if a binding exists, the microblock first applies a hash on the appropriate fields based on the packet direction (see [Section 27.4.2.1.1, “NAT Table Indexing Mechanism” on page 485](#)). If no match is found, the microblock checks if NAT-PT mode is enabled. If it is, then the microblock compares the destination address with the V4 address set aside by the Translation core component for NAT-PT. If this also matches, it performs a second hash based on the appropriate fields for the NAT-PT index table. If no match is found, it means there is no prior binding and the flow proceeds as shown in the flowchart. If a match is found, data from the matched entry in the index table is used to index to the appropriate entry in the table where the actual state is maintained.

Note:

1. For an inbound session the final destination v6 address is known beforehand, at the time of creating the binding itself (unlike an outgoing session where the destination address is part of the v4-mapped-v6 address), which is used to speed up the translation process. The state created for an inbound session also includes the next-hop id that would be generated if a v6 lookup on the bound address were done. Thus, when an inbound packet is translated to a v6 packet, the next-hop id already exists and is set in the meta-data for the subsequent microblocks to use. This avoids the lookup on the v6 address, which would otherwise have to be done in order to route the translated v6 packet correctly. The NAT-PT microblock (via a flag in the meta-data) indicates to the forwarder that a lookup need not be done. In this case the forwarder microblock simply does the necessary validations on the packet before passing it on.
2. This optimization does add some burden on the Translation core component to ensure that the next-hop id stored in the NAT-PT state is in sync with the IPv6 forwarding table.
3. The microblock raises an exception to create a binding only for outbound packets. This is because a binding is created for only a v6 address and not the other way round. Even when a v4 host initiates a session, session initiation would have to be preceded by a DNS host lookup to obtain the address. This lookup is intercepted by a DNS ALG, which would take care of adding an appropriate binding.

Figure 27-4 shows that in the IPv4 path, the check for DNS packet is done only if a binding exists. A static binding for the DNS server in the IPv6 realm must be set up beforehand. The IPv4 address is used in this binding is provided to the IPv4 realm. The DNS servers in the IPv4 realm uses this address in the DNS query and hence, the DNS query packets can be intercepted by the DNS ALG.

Figure 27-4. Process Flow for NAT-PT Microblock



Translation of IPv4 packets to IPv6 begins by checking for unexpired source route options in the IPv4 packet. RFC 2765 mandates that no IPv4 options be translated while transiting from IPv4 to IPv6 realm. In addition, it requires that packets with unexpired source route options be dropped and

an error message returned to the sending application. Translation is processed for packets that pass this check. Only specific fields in the IPv4 header are translated as shown in [Figure 27-4](#). The notation STATE(destFields4) is used to indicate that the field is extracted using the state stored in the microblock by keying on the destination fields in the IPv4 header. The PREFIX value used in the outgoing IPv6 packet's source address is patched by the core. By default, it is set to the v4-mapped-v6 prefix, 0:0:0:0:FFFF/96.

The next task is to determine if the IPv6 packet generated should contain a fragment extension header. If the sender of the packet allows fragmentation (as determined by looking at the DF field) or if the original IPv4 packet is already fragmented, the microblock adds a fragment extension header. [Figure 27-4](#) shows the values to set in the various fields in the header.

The checksum fields in the headers of TCP and UDP packets incorporate a pseudo-header that includes the network addresses. Since translation involved changing the network addresses, the checksum fields in the TCP/UDP headers should be updated. The incremental checksum update algorithm specified in RFC 1631 is used. If the UDP checksum is 0, instead of an incremental update, the entire checksum needs to be generated. This can happen, as a UDP checksum, unlike in IPv6, is not mandatory in IPv4.

In case the packet is fragmented such that the entire transport header is not in one packet, the microblock looks at the offsets of the fields in the transport header and updates the checksum field in the fragment that contains it. This is possible because the microblock only needs to do an incremental update and the fields changed are already known. This check allows fragmented transport packets to be handled in the microblock itself instead of unnecessarily generating exceptions.

Note: [Figure 27-4](#) only shows the steps needed for NAT. For NAPT, the port fields in the transport header is updated, in addition to the steps needed for NAT. The updated checksum in the transport header must account for these port changes.

Wherever there is an ambiguity as to which header, IPv6 or IPv4, a particular field is extracted from, the name of the field is suffixed with the protocol number—for example, Frag_offset.

Figure 27-5. IPv4 to IPv6 Translation (Page 1 of 2)

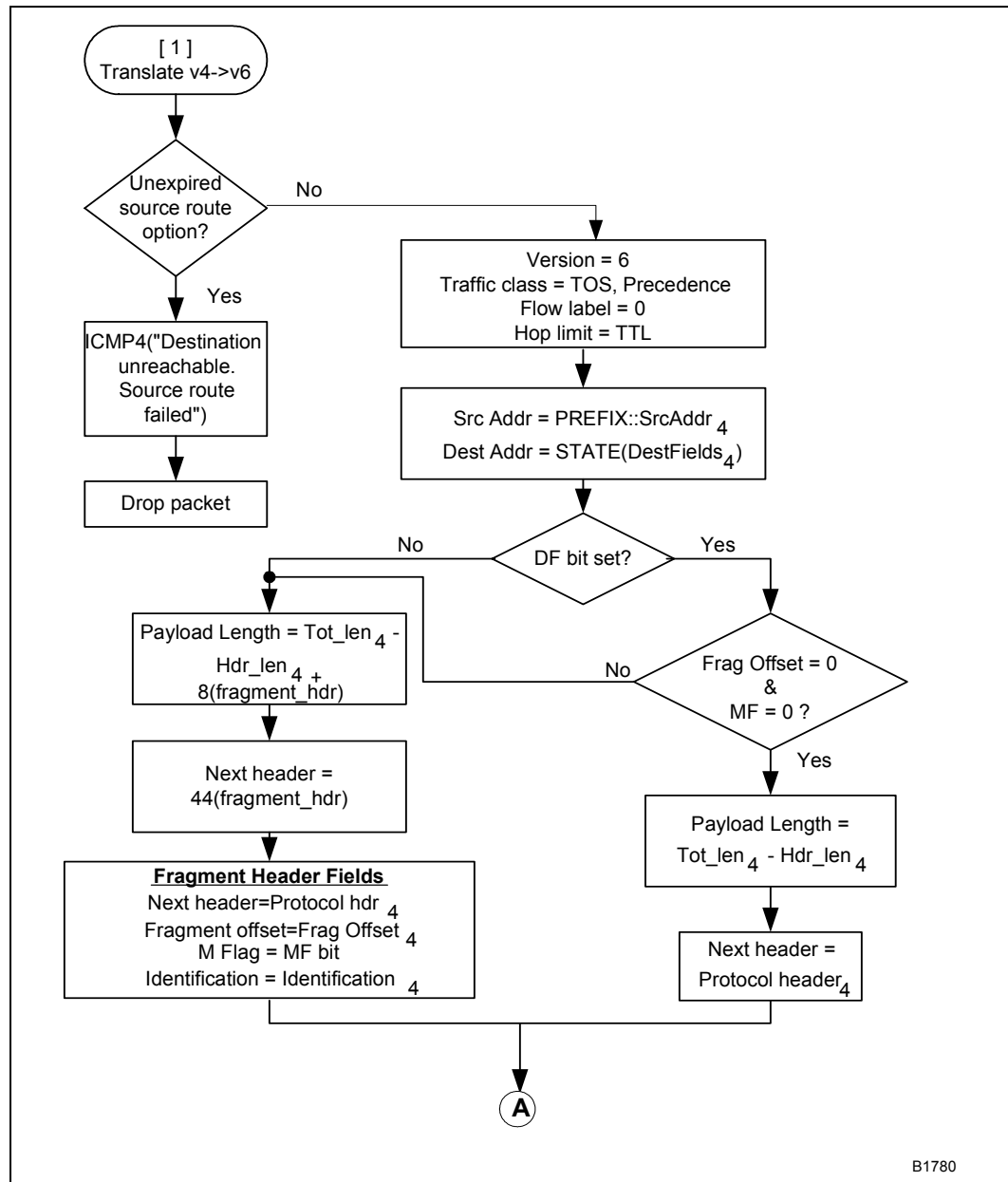
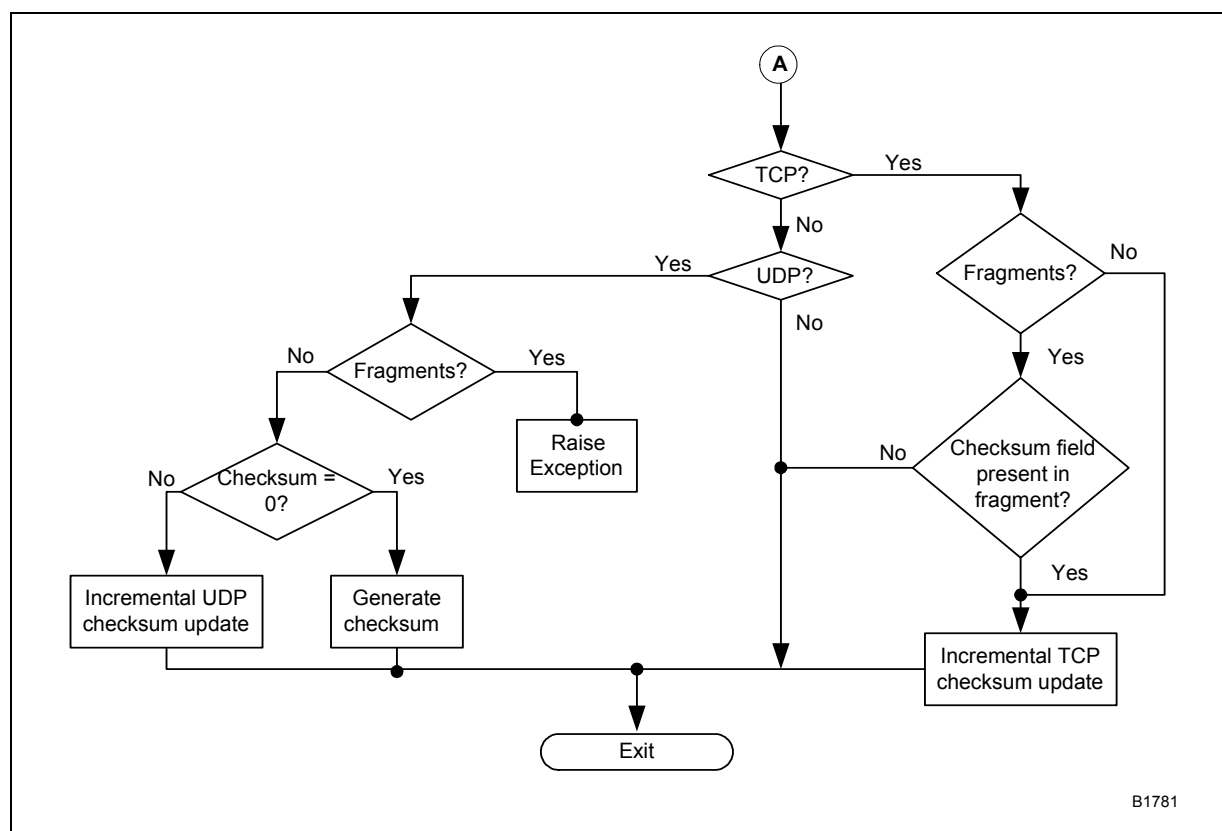


Figure 27-6. IPv4 to IPv6 Translation (Page 2 of 2)



Translation of IPv6 packets to IPv4 begins with a check for routing extension header. RFC 2765 mandates that IPv6 extension headers (except fragment) be ignored. In addition, it requires that packets with a routing extension header containing a non-zero segments left field be dropped and an error message returned to the sender. Packets that are not dropped due to the above checks are translated. Figure 27-5 and Figure 27-6 shows only the specific fields in the IPv6 header that are translated. Based on the presence of the fragment header in the IPv6 packet, the appropriate fields in the IPv4 are set (Figure 27-5 and Figure 27-6), allowing (or disallowing) fragmentation of the packet in the IPv4 realm.

The notation STATE (SrcFields6) is used to indicate that the field is extracted using the state stored in the microblock by keying on the relevant source fields in the IPv6 header. Once all the fields in the IPv4 header are set appropriately, the header checksum is generated and added in.

As in the case of the IPv4 to IPv6 translation, the transport header checksum field needs to be updated to account for the change in the addresses (and ports in case of NATP).

Figure 27-7. IPv6 to IPv4 Translation (Page 1 of 2)

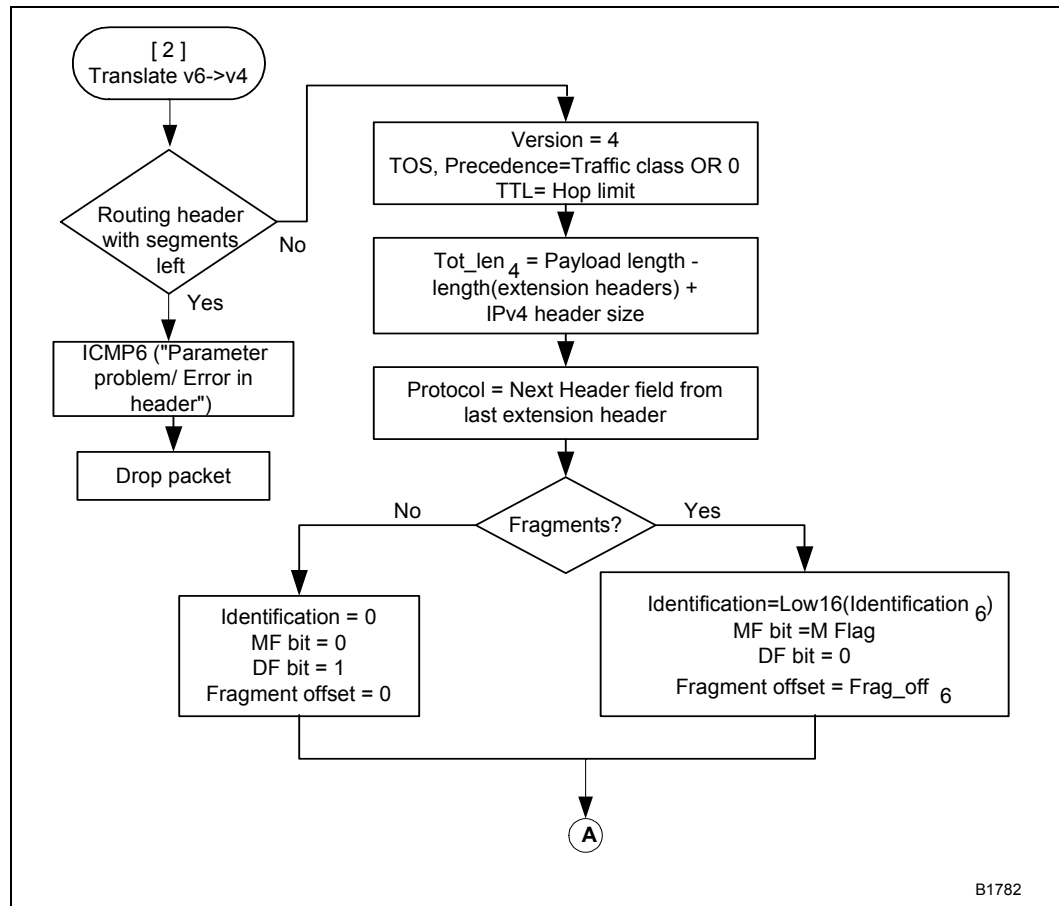
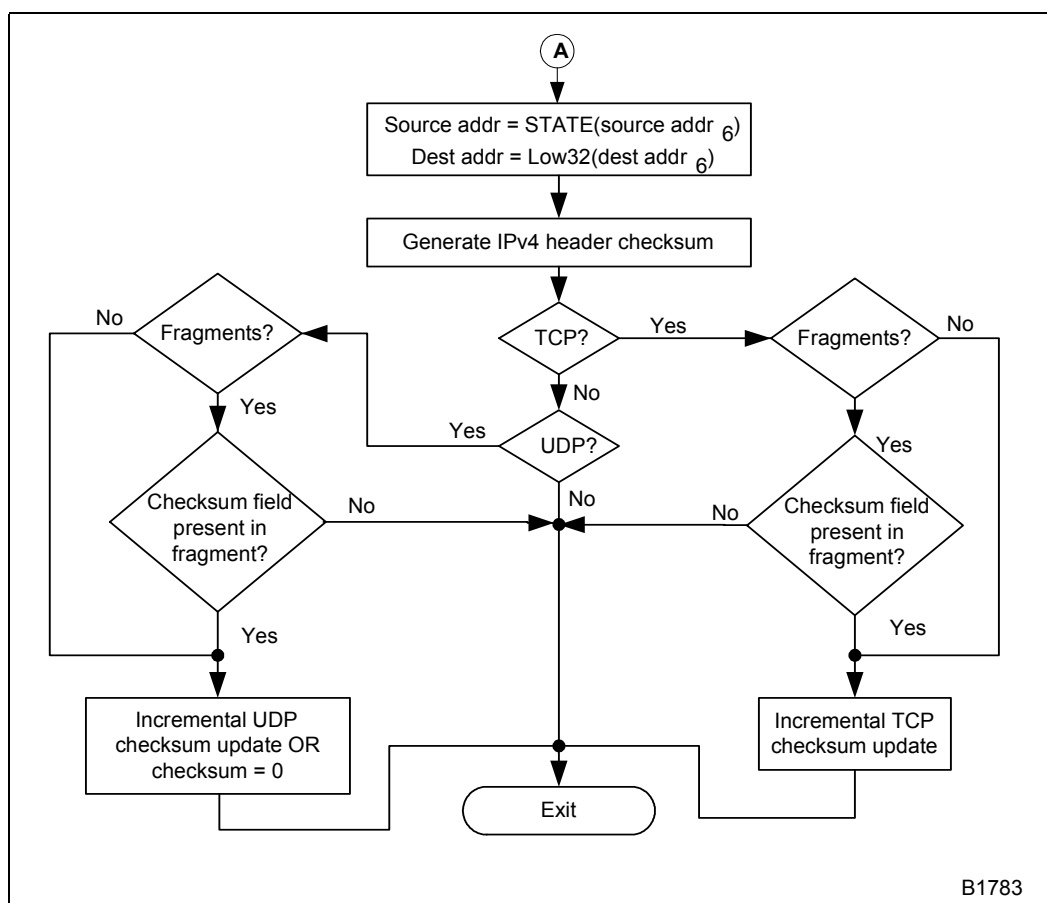


Figure 27-8. IPv6 to IPv4 Translation (Page 2 of 2)



27.5 XScale Core Interface

This section lists the symbols and exception codes shared between the NATPT microblock and the NAT-PT core component.

27.5.1 Symbols

In order for the NAT-PT microblock and the Translation core component to work together, various symbols need to be in sync between the two components. These include symbols related to:

- Translation data structures created by the Translation core component and accessed by the microblock (explained in [Section 27.4.2, “Translation Data Structures”](#) on page 484).
- Hashing operations. The Translation core component hashes on key fields and adds entries to the tables. Later, the microblock must hash on the same fields to access the state added by the core.
- Address Prefix that dictates which packets will be translated. The Prefix value is configurable from the Translation core component and used by the microblock.

- Debug counters. The Translation core component allocates space for the counters. The address of this allocated space is passed to the microblock.

Table 27-9. Symbols Required for NAT-PT Microblock and the Core component to Work

Symbol	Description
NATPT_NAT_TABLE_BASE	Base address of the NAT table
NATPT_NAT_V6V4_HASH_TABLE_BASE	Base address of the v6-v4 NAT Hash Index
NATPT_NAT_V4V6_HASH_TABLE_BASE	Base address of the v4-v6 NAT Hash Index
NATPT_NAT_COLL_HASH_TABLE_BASE	Base address of the space set aside for hash collisions
NATPT_HASH_MUL_48_0, NATPT_HASH_MUL_48_1	Multiplier to be used when v4 addresses are hashed
NATPT_HASH_MUL_128_0, NATPT_HASH_MUL_128_1, NATPT_HASH_MUL_128_2, NATPT_HASH_MUL_128_3	Multiplier to be used when v6 addresses are hashed
NATPT_PREFIX_0, NATPT_PREFIX_1, NATPT_PREFIX_2	Address prefix of packets that need to be translated
NATPT_SRAM_COUNTER_BASE	Base address of space set aside for counters

27.5.2 Exception Codes

Table 27-10 lists the exception codes generated by the NAT-PT microblock. These are passed along with the packet to the NAT-PT core component.

Table 27-10. Exception Codes Generated by the NAT-PT Microblock

Exception code	Value	Description
NATPT_EXCEPTION_NOENTRY_IPV6	0x80	No entry found in the NAT state for a v6 packet that needs to be translated
NATPT_EXCEPTION_ICMP_IPV4	0x82	Received an ICMP packet
NATPT_EXCEPTION_ICMP_IPV6	0x83	Received an ICMPv6 packet
NATPT_EXCEPTION_DNS	0x8A	Received a DNS packet
NATPT_EXCEPTION_FTP	0x8B	Received an FTP packet
NATPT_EXCEPTION_REASSEMBLE_UDP	0x8C	Received a fragmented UDP packet
NATPT_EXCEPTION_UDP_V4_CHECKSUM_ZERO	0x8D	Received a complete UDP packet with checksum equal to zero
NATPT_EXCEPTION_IPV6_SECURITY_HEADER	0x8E	Received an IPv6 packet with the Authentication and/or Encapsulation extension header

27.5.3 Performance Analysis

Table 27-11. Translation Microblock Cycle Count Analysis

Cycle Count Analysis	Values
Available cycle count for 98 byte min Ethernet packet	(117*4)cycles
Estimated cycle count for min v4 packet (i.e. v4 packet with TCP payload) based on flow chart	452 cycles
Estimated cycle count for min v6 packet (i.e. v6 packet with TCP payload) based on flow chart	541 cycles

Table 27-12. I/O Latency Analysis (IPv4-TCP)

I/O Latency Analysis (IPv4-TCP)	Values
Available I/O latency for min packet	(117*4*8)cycles
Read Hash entry	1 SRAM read (2LW)
Read NAT state entry	1 SRAM read (6 LW)
Set entry used flag in NAT state entry	1 SRAM write
Flush fragment header	1 DRAM write (1 QW)
Flush TCP checksum	1 DRAM write (1 QW)
Check binding for v4 destination address	1 hash operation (hash_48)

Table 27-13. I/O Latency Analysis (IPv6-TCP)

I/O Latency Analysis (IPv6-TCP)	Values
Available I/O latency for min packet	(117*4*8)cycles
Read Hash entry	1 SRAM read (2LW)
Read NAT state entry	1 SRAM read (5 LW)
Set entry used flag in NAT state entry	1 SRAM write
Flush TCP checksum	1 DRAM write (1 QW) + 1 DRAM read (1QW)
Check binding for v6 source address	1 hash operation (hash_128)

27.5.4 Characterization Data

Table 27-14 contains characterization data for the IPv6 to IPv4 tunnel decapsulation microblock.

Table 27-15 contains characterization data for the IPv6 to IPv4 tunnel encapsulation microblock.

Table 27-14. IPv6 to IPv4 Tunnel Decap Microblock Characterization Data

Data	Value
General:	
Microblock Name	tunnel decap
Microblock Version Number	1.0
Implementation Language	microcode
Configuration Options use to gather this set of data	1. V6V4_INGRESS_LIST_TRIE 2. V6V4_DECAP_VERIFY_IPV4_SOURCE 3. V6V4_DECAP_VERIFY_IPV6_SOURCE 4. V6V4_DECAP_COUNTERS 5. V6V4_UPDATE_OFFSET 6. IXP_MICROCODE
Measurement Environment (tool settings)	SDK 3.5
Performance: (cycle counts, latencies, bandwidths)	
Instruction cycle count per packet (common case packet / worst case cycle count)	auto tunnel path : 70 src valid path: 90 no src valid path: 70
Common-case packet/path assumptions to be documented here	
Scratch Memory	
# of longwords read (for bandwidth calculations)	3
# of longwords written (for bandwidth calculations)	3
# and type of each atomic operation performed (for bandwidth calculations)	0
SRAM	
# of longwords read	auto tunnel path : 1 src valid path: 17 no src valid path: 1
# of longwords written	0
# and type of each atomic operation performed (for bandwidth calculations)	0
DRAM	
# of quadwords read	8
# of quadwords written	6
Other I/O instructions (i.e. MSF, PCI, CAP, and Hash)	
List of dependent I/O accesses in the longest latency path	
Per-Microengine Resources:	
Control-Store Usage (# of instructions used)	314

Table 27-14. IPv6 to IPv4 Tunnel Decap Microblock Characterization Data (Continued)

Data	Value
Local Memory Footprint (# of long words used)	24*8=192
Local Memory Configuration (shared, or per-context pointer)	per-context
Local Memory - # of LM pointers used	2
GPR Usage – minimum, static usage (absolutes, static, globals)	
Transfer Reg. Usage – minimum, static usage	
Next Neighbor Reg. Usage – minimum, static usage	
Signal Usage – minimum, static usage	
CAM used? (yes or no)	No

Global Resources:	
Scratch footprint (# of longwords used) – constant or formula ...	None
SRAM footprint (# of longwords used) – constant or formula ...	TUNNEL_DECAP_NEXTHOP_SRAM_SIZE V6V4_NUM_INGRESS_LIST_BLOCKS*V6V4_INGRESS_LIST_LW
DRAM footprint (# of quadwords used) – constant or formula ...	None
Q-Array usage - # of queues used and if they need to be cached	None
CRC Unit used?	no
Hash Unit used? (yes or no)	no

MSF Usage Information:	
Media Bus Configuration	N/A
RBUF, TBUF usage	N/A
CBus signals	

Other Information:	
Critical Section Length (compute cycles + memory accesses)	
# of phases	1
Packet Metadata - fields read	
Packet Metadata - fields written	
Header - fields read	
Header - fields written	

Documentation:	
Thread Ordering Requirements	no

Table 27-14. IPv6 to IPv4 Tunnel Decap Microblock Characterization Data (Continued)

Data	Value
OS dependencies	VxWorks, Linux
Chip/hardware dependencies (i.e. Crypto Unit in 2850)	IXP2800
Tested on which SDK Release(s)	IXA SDK 3.5
Tested on hardware? Which hardware configuration?	Yes, POS, ETH
Tested in which applications (not an all inclusive list)	Ingress tunneling apps
Possible Configuration Options	
Major limitations (i.e. Rx/Tx microblocks written to use entire ME)	
Packet Sequencing Issues (esp. in POT applications)	
Core Component or Interface requirements or dependencies	

Table 27-15. IPv6 to IPv4 Tunnel Encap Microblock Characterization Data

Data	Value
General:	
Microblock Name	tunnel encap
Microblock Version Number	1.0
Implementation Language	microcode
Configuration Options use to gather this set of data	1. V6V4_ENCAP_VERIFY_DEST_AUTO 2. V6V4_UPDATE_OFFSET 3. V6V4_ENCAP_COUNTERS 4. IXP_MICROCODE
Measurement Environment (tool settings)	SDK 3.5
Performance: (cycle counts, latencies, bandwidths)	
Instruction cycle count per packet (common case packet / worst case cycle count)	100
Common-case packet/path assumptions to be documented here	
Scratch Memory	
# of longwords read (for bandwidth calculations)	3
# of longwords written (for bandwidth calculations)	3
# and type of each atomic operation performed (for bandwidth calculations)	0
SRAM	
# of longwords read	8
# of longwords written	0

Table 27-15. IPv6 to IPv4 Tunnel Encap Microblock Characterization Data (Continued)

Data	Value
# and type of each atomic operation performed (for bandwidth calculations)	0
DRAM	
# of quadwords read	8
# of quadwords written	8
Other I/O instructions (i.e. MSF, PCI, CAP, and Hash)	
List of dependent I/O accesses in the longest latency path	
Per-Microengine Resources:	
Control-Store Usage (# of instructions used)	177
Local Memory Footprint (# of long words used)	24*8=192
Local Memory Configuration (shared, or per-context pointer)	per-context
Local Memory - # of LM pointers used	2
GPR Usage – minimum, static usage (absolutes, static, globals)	
Transfer Reg. Usage – minimum, static usage	
Next Neighbor Reg. Usage – minimum, static usage	
Signal Usage – minimum, static usage	
CAM used? (yes or no)	No
Global Resources:	
Scratch footprint (# of longwords used) – constant or formula ...	None
SRAM footprint (# of longwords used) – constant or formula ...	TUNNEL_ENCAP_NEXTHOP_SRAM_SIZE
DRAM footprint (# of quadwords used) – constant or formula ...	None
Q-Array usage - # of queues used and if they need to be cached	None
CRC Unit used?	no
Hash Unit used? (yes or no)	no
MSF Usage Information:	
Media Bus Configuration	N/A
RBUF, TBUF usage	N/A
CBus signals	
Other Information:	

Table 27-15. IPv6 to IPv4 Tunnel Encap Microblock Characterization Data (Continued)

Data	Value
Critical Section Length (compute cycles + memory accesses)	1
# of phases	
Packet Metadata - fields read	
Packet Metadata - fields written	
Header - fields read	
Header - fields written	
Documentation:	
Thread Ordering Requirements	no
OS dependencies	VxWorks, Linux
Chip/hardware dependencies (i.e. Crypto Unit in 2850)	IXP2800
Tested on which SDK Release(s)	IXA SDK 3.5
Tested on hardware? Which hardware configuration?	Yes, POS, ETH
Tested in which applications (not an all inclusive list)	Ingress tunneling apps
Possible Configuration Options	
Major limitations (i.e. Rx/Tx microblocks written to use entire ME)	
Packet Sequencing Issues (esp. in POT applications)	
Core Component or Interface requirements or dependencies	



Layer 2

The Layer 2 microblocks include the following:

- [Chapter 28, “Layer-2 Decapsulation and Classify”](#)
- [Chapter 29, “Layer-2 Encapsulation”](#)

Layer-2 Decapsulation and Classify 28

The layer-2 decapsulation/classify microblocks refer to a set of media dependent blocks that remove the layer-2 header and classify the packet for further processing down the pipeline. These include the PPP decap/classify, Ethernet decap/classify and LLC SNAP decap/classify microblocks.

28.1 PPP Decapsulation and Classify

This microblock is used in the POS application. It uses the type field in the PPP header to classify the packet into IPv4, IPv6, LCP, IPCP etc. It sets the header type field in the packet metadata and the `dl_next_block` dispatch loop variable based on this classification. PPP signaling packets, where the type is LCP, IPCP, and so on, are sent to the Intel XScale® core—that is, `dl_next_block` is set to `IX_EXCEPTION`. If the packet type is not recognized, the header type field in the metadata is set to unknown and the `dl_next_block` variable is set to `IX_NULL`.

To remove the PPP header, the microblock increments the offset and decrements the packet and SOP buffer size fields in the metadata via the appropriate dispatch loop macros.

28.2 Ethernet Decapsulation, Classify and Filter

This microblock is used in the Ethernet application. It uses the Ether Type field in the Ethernet header to classify the packet into IPv4, IPv6, MPLS, ARP, and so on. It sets the header type field in the packet metadata and the `dl_next_block` dispatch loop variable based on this classification. ARP packets are sent to the Intel XScale® core—`dl_next_block` is set to `IX_EXCEPTION`. If the packet type is not recognized, the header type field in the metadata is set to unknown and the `dl_next_block` variable is set to `IX_NULL`.

To remove the Ethernet header, the microblock increments the offset and decrements the packet and SOP buffer size fields in the meta data via the appropriate dispatch loop macros.

This microblock also implements MAC filtering based on the destination MAC address in the Ethernet header. A hash table is used to store the MAC addresses, as shown in [Figure 28-1](#).

Figure 28-1. Hash Table Entry for MAC Filtering

DMAC0	DMAC1	DMAC2	DMAC3
DMAC4	DMAC5	PORT	Next Index

B0941-01

The pseudo-code for the MAC filtering is provided below:

```

_ether_filter_mac()
{
  if(dmac is broadcast)
  {
    filter pass
    go to end
  }
  else
  {
    set filter state is FAIL ; this is default

    read port type from SRAM port bit map storage

    compute the hash value of the dmac--- hash table index

    transfer the index to SRAM address for the table entry

    while(index not zero)
    {
      transfer the index to SRAM address for the table entry

      read hash table entry using the address

      if (read index = current index)
      {
        if (read port == current port)
          if(read MAC == current MAC)
            set filter state is PASS
        go to check promisc
      }
      else
      {
        if (read port == current port)
          if(read MAC == current MAC)
          {
            set filter state is PASS
            go to check promisc
          }
        else
          read index => current index
      }
    }
  }
}

// end of the macro

check_promisc:

  wait SRAM bit map read done

  if (filter state is PASS)

```



```
    go to end
  else
    if(port type is promisc)
      MAC filter is PASS
    else
      MAC filter FAIL
  end:
end:
```

28.3 LLC SNAP Decapsulation/Classify

It uses the header type field of packet metadata, set by AAL5 RX to identify whether the packet is encapsulated or not. If the packet is encapsulated—that is, the packet has an LLC-SNAP header, it uses the SNAP header protocol id to classify the packet into IPv4, IPv6, etc. It sets the header type field in the packet metadata and the `dl_next_block` dispatch loop variable and performs packet decapsulation based on this classification. ARP packets are sent to the XScale (`dl_next_block` is set to `IX_EXCEPTION`). If the frame is not encapsulated—that is, it is directly IP datagram inside the AAL-5 frame, it just classifies the packet into IPv4, IPv6, etc. The packet is then sent down the microengine pipeline for further processing. If the packet type is not recognized, the header type field in the metadata is set to unknown and the `dl_next_block` variable is set to `BID_DROP`.

To remove the LLC SNAP header—that is, packet decapsulation, the microblock increments the offset and decrements the packet and SOP buffer size fields in the metadata via the appropriate dispatch loop macros.

Figure 28-2. Header Type Field of Packet Metadata

```

If (dl_next_block == BID_LLC_SNAP_DECAP) {
    Get packet header - type from metadata
    If (header - type == LLC_SNAP) {
        // Here the frame is encapsulated
        IP_hdr_offset = 8B (here LLC_SNAP header)
        Get SNAP header protocol id
        Do packet decap based on encap (here LLC_SNAP header)
        If (protocol id = ETHER_IPv4_PID) {
            dl_next_block = BID_IPv4
            Set header - type in pkt meta data = ETHER_IPv4_PID
        } Else
            dl_next_block = BID_DROP
    } Else {
        // Here the frame is not encapsulated.
        If (header - type == IPv4) {
            IP_hdr_offset = 0B
            dl_next_block = BID_IPv4
        } Else
            dl_next_block = BID_DROP
    }
} Else
    by_pass

```

28.4 Performance Analysis

Table 28-1 shows the worst case instruction count for the different decap blocks:

Table 28-1. Layer-2 Decapsulation Worst Case Instruction Count

Microblock	Instruction Count
PPP Decap/Classify	16
Ethernet Decap/Classify/Filter	73
LLCSNAP Decap/Classify	16

28.4.1 Characterization Data

This section contains characterization data for the following microblocks:

- [Table 28-2, “Ethernet Decap Microblock Characterization Data” on page 511](#)
- [Table 28-3, “PPP Decap Microblock Characterization Data” on page 513](#)

Table 28-2. Ethernet Decap Microblock Characterization Data

Data	Value
General:	
Microblock Name	Ethernet_Decap
Microblock Version Number	1
Implementation Language	microcode
Configuration Options use to gather this set of data	N/A
Measurement Environment (tool settings)	N/A
Performance: (cycle counts, latencies, bandwidths)	
Instruction cycle count per packet (common case packet / worst case cycle count)	73
Common-case packet/path assumptions to be documented here	The packet IP header is already read into ME registers.
Scratch Memory	N/A
# of longwords read (for bandwidth calculations)	
# of longwords written (for bandwidth calculations)	
# and type of each atomic operation performed (for bandwidth calculations)	
SRAM	
# of longwords read	3
# of longwords written	
# and type of each atomic operation performed (for bandwidth calculations)	
Co-processors	
bytes read (per channel)	
bytes written (per channel)	
DRAM	
# of quadwords read	N/A
# of quadwords written	12
Other I/O instructions (i.e. MSF, PCI, CAP, and Hash)	N/A
List of dependent I/O accesses in the longest latency path	Read port_type bit map (sram rd 1) Read Ether MAC filtering hash table (sram rd 2)

Table 28-2. Ethernet Decap Microblock Characterization Data (Continued)

Data	Value
Per-Microengine Resources:	
Control-Store Usage (# of instructions used)	200
Local Memory Footprint (# of long words used)	0
Local Memory Configuration (shared, or per-context pointer)	
Local Memory - # of LM pointers used	
GPR Usage – minimum, static usage (absolutes, static, globals)	
Transfer Reg. Usage – minimum, static usage	
Next Neighbor Reg. Usage – minimum, static usage	N/A
Signal Usage – minimum, static usage	0
CAM used? (yes or no)	N/A
Global Resources:	
Scratch footprint (# of longwords used) – constant or formula ...	N/A
SRAM footprint (# of longwords used) – constant or formula ...	2048
DRAM footprint (# of quadwords used) – constant or formula ...	
Q-Array usage - # of queues used and if they need to be cached	N/A
CRC Unit used?	N/A
Hash Unit used? (yes or no)	N/A
MSF Usage Information:	
Media Bus Configuration	N/A
RBUF, TBUF usage	N/A
CBus signals	
Other Information:	
Critical Section Length (compute cycles + memory accesses)	N/A
# of phases	0
Packet Metadata - fields read	input_port, check_sum, sop_buf_size, sop_offset, packet_size
Packet Metadata - fields written	sop_buf_size, sop_offset, packet_size, rx_stat, check_sum, header_type
Header - fields read	Dest MAC, Ether Fr Type, 14B of IP header to calculate checksum
Header - fields written	N/A

Table 28-2. Ethernet Decap Microblock Characterization Data (Continued)

Data	Value
Documentation:	
Thread Ordering Requirements	N/A
OS dependencies	N/A
Chip/hardware dependencies (i.e. Crypto Unit in 2850)	
Tested on which SDK Release(s)	IXA SDK 3.5
Tested on hardware? Which hardware configuration?	YES, 4 Gb Ethernet, 10 Gb Ethernet
Tested in which applications (not an all inclusive list)	4 Gb Ethernet, 10 Gb Ethernet
Possible Configuration Options	None
Major limitations (i.e. Rx/Tx microblocks written to use entire ME)	N/A
Packet Sequencing Issues (esp. in POT applications)	N/A
Core Component or Interface requirements or dependencies	N/A

Table 28-3. PPP Decap Microblock Characterization Data

Data	Value
General:	
Microblock Name	PPP_decap
Microblock Version Number	1
Implementation Language	microcode
Configuration Options use to gather this set of data	N/A
Measurement Environment (tool settings)	N/A
Performance: (cycle counts, latencies, bandwidths)	
Instruction cycle count per packet (common case packet / worst case cycle count)	16
Common-case packet/path assumptions to be documented here	The packet IP header is already read into ME registers.
Scratch Memory	N/A
# of longwords read (for bandwidth calculations)	
# of longwords written (for bandwidth calculations)	
# and type of each atomic operation performed (for bandwidth calculations)	
SRAM	N/A
# of longwords read	
# of longwords written	

Table 28-3. PPP Decap Microblock Characterization Data (Continued)

Data	Value
# and type of each atomic operation performed (for bandwidth calculations)	
DRAM	N/A
# of quadwords read	
# of quadwords written	
Other I/O instructions (i.e. MSF, PCI, CAP, and Hash)	N/A
List of dependent I/O accesses in the longest latency path	N/A
Per-Microengine Resources:	
Control-Store Usage (# of instructions used)	75
Local Memory Footprint (# of long words used)	N/A
Local Memory Configuration (shared, or per-context pointer)	N/A
Local Memory - # of LM pointers used	N/A
GPR Usage – minimum, static usage (absolute, static, globals)	
Transfer Reg. Usage – minimum, static usage	
Next Neighbor Reg. Usage – minimum, static usage	N/A
Signal Usage – minimum, static usage	
CAM used? (yes or no)	N/A
Global Resources:	
Scratch footprint (# of longwords used) – constant or formula ...	N/A
SRAM footprint (# of longwords used) – constant or formula ...	N/A
DRAM footprint (# of quadwords used) – constant or formula ...	N/A
Q-Array usage - # of queues used and if they need to be cached	N/A
CRC Unit used?	N/A
Hash Unit used? (yes or no)	N/A
MSF Usage Information:	
Media Bus Configuration	N/A
RBUF, TBUF usage	N/A
CBus signals	N/A
Other Information:	

Table 28-3. PPP Decap Microblock Characterization Data (Continued)

Data		Value
Critical Section Length (compute cycles + memory accesses)		0
# of phases		0
Packet Metadata - fields read		sop_offset, sop_buffer_size, packet_size
Packet Metadata - fields written		sop_offset, sop_buffer_size, packet_size, hdr_type
Header - fields read		PPP frame type
Header - fields written		N/A
Documentation:		
Thread Ordering Requirements		N/A
OS dependencies		N/A
Chip/hardware dependencies (i.e. Crypto Unit in 2850)		
Tested on which SDK Release(s)		IXA SDK 3.5
Tested on hardware? Which hardware configuration?		YES, OC-48 POS, OC-192 POS
Tested in which applications (not an all inclusive list)		OC-48 POS, OC-192 POS
Possible Configuration Options		No variation available
Major limitations (i.e. Rx/Tx microblocks written to use entire ME)		N/A
Packet Sequencing Issues (esp. in POT applications)		N/A
Core Component or Interface requirements or dependencies		N/A

The Layer-2 Encapsulation microblocks refer to a set of media dependent blocks that add the layer-2 header to the packet. These include the PPP, Ethernet and LLC SNAP encaps blocks.

29.1 PPP Encapsulation

This microblock adds the PPP header to the packet based on the header type field in the meta data. For example if the header type field is IPv4, then the PPP header is set to 0x0021.

If the next hop ID is invalid—that is, if the value is -1—then the microblock assumes that the PPP header has already been added to the packet by a previous stage of the pipeline, in which case it does not add another header to the packet.

29.2 Ethernet Encapsulation

This microblock adds the Ethernet header to the packet. It uses the next hop ID as an index into a table containing layer-2 header and ARP information. If layer-2 information for this packet is not in the table, the packet is queued to send to the Intel XScale[®] core. Code running on the Intel XScale[®] core initiates an ARP request for this packet and populates the table entry when a reply is received.

If the next hop id is invalid—that is, if the value is -1—then the microblock assumes that the layer-2 header has already been added to the packet by a previous stage of the pipeline, in which case it does not add another header to the packet.

29.3 LLC SNAP Encapsulation

This microblock adds the LLC SNAP header to the packet. It uses the next hop ID as an index into a table containing layer-2 header information including the LLC SNAP header, VPI/VCI and VCQ number.

If the next hop id is invalid—that is, if the value is -1—then the microblock assumes that the layer-2 header has already been added to the packet by a previous stage of the pipeline, in which case it does not add another header to the packet.

Since a header has been added to the packet, the offset, packet size, SOP buffer size, SOP cell count fields in the metadata need to be updated. Also the block computes the cell count on the entire packet to send to the Queue Manager microengine. It also checks if an extra ATM cell needs to be added to accommodate the eight-byte trailer. If so it adds one byte to the EOP buffer size. This ensures that the EOP buffer is freed by the TX microblock only after the additional cell is transmitted. The TX microblock however does not transmit this extra one byte since it uses the packet size to keep track of the bytes sent. As a compile time option, the microblock may get outgoing VPI/VCI from the layer-2 table and store it in the `flow_id` of the meta data.

Table 29-1 describes the format of an entry in the layer-2 table for LLC SNAP:

Table 29-1. Format of an Entry in the Layer-2 Table for LLC SNAP

LW	Bits	Size	Field	Description
0	23:0	24	Out VPI/VCI	Outgoing VPI/VCI for the packet
1	15:0	16	VCQ#	VCQ# for the packet
2	31:0	32	LLC/SNAP header	Bytes 0-3 of LLC/SNAP header
3	31:0	32	LLC/SNAP header	Bytes 4-7 of LLC/SNAP header

29.4 Performance Analysis

29.4.1 Ethernet Encap

For the Ethernet Encap block, the worst case instruction and I/O are detailed below. Not that this is for the case, where the header is unaligned. If we assume that the header is at a known compile time offset, this microblock can be substantially simplified.

Table 29-2. Layer-2 Encapsulation Worst Case Cycle Count

Component	Worst Case Cycle Count
Phase 1	12
Phase 2	17
Phase 3	47
Phase 4	14
Total	90

Table 29-3. Layer-2 Encapsulation I/O Operations

I/O operations	Phase
SRAM read of 5 long words of SOP meta data	1
SRAM read of 4 long words of L2 table entry	2
DRAM read of 2 long words around SOP payload offset	2
DRAM write of 6 long words to put Ethernet header before payload	3
SRAM write of 2 long words to update SOP meta data	3
Scratch read of 3 long words of l2_encap source message	4
Scratch write of 3 long words of l2_encap sink message	4

29.4.2 Ethernet Encap Characterization Data

Table 29-4. Ethernet Encap Microblock Characterization Data

Data	Value
General:	
Microblock Name	ETHERNET_ENCAP
Microblock Version Number	1.0
Implementation Language	microcode
Configuration Options use to gather this set of data	
Measurement Environment (tool settings)	
Performance: (cycle counts, latencies, bandwidths)	
Instruction cycle count per packet (common case packet / worst case cycle count)	90
Common-case packet/path assumptions to be documented here	Assumptions: all numbers reported are worst-case for for 4x1 Gbs Ethernet normal path
Scratch Memory	
# of longwords read (for bandwidth calculations)	3
# of longwords written (for bandwidth calculations)	3
# and type of each atomic operation performed (for bandwidth calculations)	0
SRAM	
# of longwords read	MetaData: 5 L2Table: 4
# of longwords written	2
# and type of each atomic operation performed (for bandwidth calculations)	0
Co-processors	
bytes read (per channel)	
bytes written (per channel)	
DRAM	
# of quadwords read	N/A
# of quadwords written	1
# of quadwords written	3
Other I/O instructions (i.e. MSF, PCI, CAP, and Hash)	0
List of dependent I/O accesses in the longest latency path	
Per-Microengine Resources:	
Control-Store Usage (# of instructions used)	235
Local Memory Footprint (# of long words used)	None

Table 29-4. Ethernet Encap Microblock Characterization Data (Continued)

Data		Value
Local Memory Configuration (shared, or per-context pointer)	None	
Local Memory - # of LM pointers used	None	
GPR Usage – minimum, static usage (absolutes, static, globals)		
Transfer Reg. Usage – minimum, static usage		
Next Neighbor Reg. Usage – minimum, static usage	0	
Signal Usage – minimum, static usage	0	
CAM used? (yes or no)	No	

Global Resources:		
Scratch footprint (# of longwords used) – constant or formula ...	0	
SRAM footprint (# of longwords used) – constant or formula ...	0	
DRAM footprint (# of quadwords used) – constant or formula ...	0	
Q-Array usage - # of queues used and if they need to be cached	0	
CRC Unit used?		
Hash Unit used? (yes or no)	No	

MSF Usage Information:		
Media Bus Configuration	No	
RBUF, TBUF usage	-	
CBus signals	-	

Other Information:		
Critical Section Length (compute cycles + memory accesses)	0	
# of phases	4	
Packet Metadata - fields read	bufferHandle packetSize bufferSize payloadOffset nextHopId	
Packet Metadata - fields written		
Header - fields read		
Header - fields written		

Documentation:		
-----------------------	--	--

Table 29-4. Ethernet Encap Microblock Characterization Data (Continued)

Data	Value
Thread Ordering Requirements	
OS dependencies	VxWorks, Linux
Chip/hardware dependencies (i.e. Crypto Unit in 2850)	2400
Tested on which SDK Release(s)	
Tested on hardware? Which hardware configuration?	None
Tested in which applications (not an all inclusive list)	None
Possible Configuration Options	
Major limitations (i.e. Rx/Tx microblocks written to use entire ME)	None
Packet Sequencing Issues (esp. in POT applications)	None
Core Component or Interface requirements or dependencies	ATM/POS TX and Ethernet TX core components

29.4.3 PPP Encap

For the PPP Encap block, the worst case instruction and I/O are detailed below. Not that this is for the case, where the header is unaligned. If we assume that the header is at a known compile time offset, this microblock can be substantially simplified.

Table 29-5. PPP Encap Worst Case Cycle Count

Component	Worst Case Cycle Count
Phase 1	12
Phase 2	15
Phase 3	42
Phase 4	14
Total	83

Table 29-6. PPP Encap I/O Operations

I/O Operations	Phase
SRAM read of five long words of SOP meta data	1
DRAM read of two long words around SOP payload offset	2
DRAM write of four long words to put PPP header before payload in DRAM	3
SRAM write of two long words to update SOP meta data	3
Scratch read of three long words of l2_encap source message	4
Scratch write of three long words of l2_encap sink message	4

Table 29-7. PPP Encap Microblock Characterization Data

Data	Value
General:	
Microblock Name	PPP_ENCAP
Microblock Version Number	1.0
Implementation Language	microcode
Configuration Options use to gather this set of data	
Measurement Environment (tool settings)	
Performance: (cycle counts, latencies, bandwidths)	
Instruction cycle count per packet (common case packet / worst case cycle count)	83 (available budget: 97)
Common-case packet/path assumptions to be documented here	Assumptions <ul style="list-style-type: none"> All numbers reported are worst-case for oc48_POS normal path
Scratch Memory	
# of longwords read (for bandwidth calculations)	3
# of longwords written (for bandwidth calculations)	3
# and type of each atomic operation performed (for bandwidth calculations)	0
SRAM	
# of longwords read	MetaData: 5
# of longwords written	2
# and type of each atomic operation performed (for bandwidth calculations)	0
DRAM	
# of quadwords read	1
# of quadwords written	2
Other I/O instructions (i.e. MSF, PCI, CAP, and Hash)	0
List of dependent I/O accesses in the longest latency path	
Per-Microengine Resources:	
Control-Store Usage (# of instructions used)	195
Local Memory Footprint (# of long words used)	None
Local Memory Configuration (shared, or per-context pointer)	None
Local Memory - # of LM pointers used	None
GPR Usage – minimum, static usage (absolute, static, globals)	

Table 29-7. PPP Encap Microblock Characterization Data (Continued)

Data		Value
Transfer Reg. Usage – minimum, static usage		
Next Neighbor Reg. Usage – minimum, static usage		0
Signal Usage – minimum, static usage		0
CAM used? (yes or no)		No
Global Resources:		
Scratch footprint (# of longwords used) – constant or formula ...		0
SRAM footprint (# of longwords used) – constant or formula ...		0
DRAM footprint (# of quadwords used) – constant or formula ...		0
Q-Array usage - # of queues used and if they need to be cached		0
CRC Unit used?		
Hash Unit used? (yes or no)		No
MSF Usage Information:		
Media Bus Configuration		No
RBUF, TBUF usage		-
CBus signals		-
Other Information:		
Critical Section Length (compute cycles + memory accesses)		0
# of phases		4
Packet Metadata - fields read		bufferHandle packetSize bufferSize payloadOffset headerType
Packet Metadata - fields written		
Header - fields read		
Header - fields written		
Documentation:		
Thread Ordering Requirements		
OS dependencies		VxWorks, Linux
Chip/hardware dependencies (i.e. Crypto Unit in 2850)		2400
Tested on which SDK Release(s)		

Table 29-7. PPP Encap Microblock Characterization Data (Continued)

Data	Value
Tested on hardware? Which hardware configuration?	None
Tested in which applications (not an all inclusive list)	None
Possible Configuration Options	
Major limitations (i.e. Rx/Tx microblocks written to use entire ME)	None
Packet Sequencing Issues (esp. in POT applications)	None
Core Component or Interface requirements or dependencies	ATM/POS TX and Ethernet TX core components

29.4.4 LLC SNAP Encap

For the LLC SNAP Encap block, the worst case instruction count is 144 cycles.

29.4.4.1 LLC SNAP Encap Characterization Data

Table 29-8. LLC SNAP Encap Microblock Characterization Data

Data	Value
General:	
Microblock Name	LLC_SNAP Decap
Microblock Version Number	1.0
Implementation Language	microcode
Configuration Options use to gather this set of data	N/A
Measurement Environment (tool settings)	N/A
Performance: (cycle counts, latencies, bandwidths)	
Instruction cycle count per packet (common case packet / worst case cycle count)	25
Common-case packet/path assumptions to be documented here	The packet IP header is already read into ME registers.
Scratch Memory	N/A
# of longwords read (for bandwidth calculations)	
# of longwords written (for bandwidth calculations)	
# and type of each atomic operation performed (for bandwidth calculations)	
SRAM	N/A
# of longwords read	
# of longwords written	
# and type of each atomic operation performed (for bandwidth calculations)	

Table 29-8. LLC SNAP Encap Microblock Characterization Data (Continued)

Data	Value
Co-processors bytes read (per channel) bytes written (per channel)	N/A
DRAM # of quadwords read # of quadwords written	N/A
Other I/O instructions (i.e. MSF, PCI, CAP, and Hash)	N/A
List of dependent I/O accesses in the longest latency path	N/A
Per-Microengine Resources:	
Control-Store Usage (# of instructions used)	100
Local Memory Footprint (# of long words used)	0
Local Memory Configuration (shared, or per-context pointer)	
Local Memory - # of LM pointers used	
GPR Usage – minimum, static usage (absolutes, static, globals)	
Transfer Reg. Usage – minimum, static usage	
Next Neighbor Reg. Usage – minimum, static usage	N/A
Signal Usage – minimum, static usage	0
CAM used? (yes or no)	No
Global Resources:	
Scratch footprint (# of longwords used) – constant or formula ...	
SRAM footprint (# of longwords used) – constant or formula ...	
DRAM footprint (# of quadwords used) – constant or formula ...	
Q-Array usage - # of queues used and if they need to be cached	N/A
CRC Unit used?	N/A
Hash Unit used? (yes or no)	N/A
MSF Usage Information:	
Media Bus Configuration	N/A
RBUF, TBUF usage	N/A
CBus signals	

Table 29-8. LLC SNAP Encap Microblock Characterization Data (Continued)

Data	Value
Other Information:	
Critical Section Length (compute cycles + memory accesses)	N/A
# of phases	0
Packet Metadata - fields read	sop_offset, sop_buffer_size, packet_size, header_type
Packet Metadata - fields written	header_type
Header - fields read	SNAP hdr Protocol Id
Header - fields written	N/A
Documentation:	
Thread Ordering Requirements	N/A
OS dependencies	N/A
Chip/hardware dependencies (i.e. Crypto Unit in 2850)	
Tested on which SDK Release(s)	IXA SDK 3.5
Tested on hardware? Which hardware configuration?	4 OC-12 AAL5 Ingress
Tested in which applications (not an all inclusive list)	4OC-12AAL5 Ingress(simulation&hardware), OC-48 AAL5 Ingress (simulation)
Possible Configuration Options	No variation available
Major limitations (i.e. Rx/Tx microblocks written to use entire ME)	N/A
Packet Sequencing Issues (esp. in POT applications)	N/A
Core Component or Interface requirements or dependencies	N/A

DiffServ

The DiffServ microblocks include the following:

- [Chapter 30, “6-tuple Exact Match Classifier Microblock”](#)
- [Chapter 31, “Three Color Meter Microblock”](#)
- [Chapter 32, “DSCP Marker Microblock”](#)
- [Chapter 33, “DSCP Classifier Microblock”](#)
- [Chapter 34, “Weighted Random Early Detection \(WRED\) Microblock”](#)

Microblocks described in this section provide Classification, Metering, Marking, Shaping and Policing—collectively called Diffserv functionality. Together they avoid congestion in the network.

Classifiers select packets in a stream based on the content of some portion of the packet header. Traffic meters measure the temporal properties of the stream of packets selected by the classifier. Then Packet Markers set the DS field of a packet to a particular code point. Based on marking, some or all packets in a traffic stream are delayed or dropped in order to bring the stream in to compliance thus avoiding congestion.

The following summarizes the Diffserv microblocks supported in the SDK:

Microblock	Description	Usage	Cycle Budget
6 Tuple Classifier	Multi-field range matching classifier.	Runs on multiple microengines in parallel. Number of microengines used depends on required data rate (OC-12 to OC-192)	97 for OC48 POS57 for OC192 POS94 10 GBE
DSCP Classifier	Classifier based on 6-bit DSCP value in IP header. IPv4 and IPv6 packets are handled.	Runs on multiple microengines in parallel. Number of microengines used depends on required data rate (OC-12 to OC-192)	97 for OC48 POS57 for OC192 POS94 10 GBE
TCM	Three Color Meter based on RFC 2697 and RFC 2698.	Runs on multiple microengines in parallel. Number of microengines used depends on required data rate (OC-12 to OC-192)	97 for OC48 POS57 for OC192 POS94 10 GBE
DSCP Marker	Updates DSCP field in IP header based on Flow ID. Handles IPv4 and IPv6 packets.	Runs on multiple microengines in parallel. Number of microengines used depends on required data rate (OC-12 to OC-192)	97 for OC48 POS57 for OC192 POS94 10 GBE
WRED	Weighted Random Early Detection microblock provides active queue congestion avoidance.	Runs on multiple microengines in parallel. Number of microengines used depends on required data rate (OC-12 to OC-192)	97 for OC48 POS57 for OC192 POS94 10 GBE

6-tuple Exact Match Classifier Microblock

30

30.1 Overview

The DiffServ MIB (Management Information Base for the Differentiated Services Architecture, www.ietf.org/rfc/rfc3289.txt) defines a multi-field, range matching classifier, intended for IP edge nodes. The generic classifier comprises of filtering rules in a form of:

- IP source address, including host, CIDR Prefix, and “any source address”
- IP destination address, including host, CIDR Prefix, and “any destination address”
- IPv6 Flow ID
- IP protocol or “any”
- TCP/UDP/SCTP source port range, including “any”
- TCP/UDP/SCTP destination port range, including “any”
- Differentiated Services Code Point or “any”

This chapter describes a building block that supports multi-field classification with the restriction that only exact-match values on IPv4 packets are supported.

A traditional range-matching classification is based on a flow-cache model. This approach exploits an elementary observation that packets make up flows (a flow comprises packets with same values on classification fields). If a router receives a packet belonging to a new flow (for example, a TCP session), then it can expect more packets on this flow in a near future.

In a flow-cache model, the first packet of a flow goes through a slow path on Intel XScale® core. A core component runs a complex search algorithm to find a matching rule. Then, it calculates a hash value on the packet header and caches a lookup result in microblock data structures. When next packets arrive in the same flow, the microblock can quickly retrieve the lookup result from a hash table (fast processing path).

Unfortunately, the flow-cache model does not scale well to high-speed nodes.

- An OC-48 access router can handle a large number of flows simultaneously.

This brings in two issues: memory consumption and slow-path performance. Firstly, the size of a hash table should be proportionally bigger than the expected number of concurrent flows to minimize hash collisions. However, a large hash table increases maintenance overhead related to aging and evicting LRU entries.

In addition, a core component can become congested when a significant number of new flows arrive at the same time. At OC-48 speed, queuing points backlog quickly. Thus, packets can be dropped because of slow-path overload and not because of link congestion.

- The model is vulnerable to denial-of-service attacks on high-speed access links.

A malicious user can degrade performance of an access router by generating lots of short-lived flows. On such conditions, the slow-path component becomes overloaded. Again, it can lead to dropping—or significantly delaying—packets from other sources.

For the above reasons, the DiffServ design follows a flow-cache model only partially. As in the model, the fast path microblock supports exact-match lookups. However, no range matching occurs in a slow path core component. Instead, a core component updates a hash table directly after receiving a management requests. This is opposed to a flow-cache model where a hash table is updated dynamically as packets appear on new flows. The selected approach avoids costly processing on a slow-path. However, it limits the supported functionality to exact-match rules only—or “narrow” range-matching rules that can be expressed as several exact-match rules.

Nevertheless, because of flow-cache model deficiencies, it is expected that wire-speed range matching should be performed in the fast path, basing on a TCAM support. However, until TCAM chips are available, the microengines provide an exact-match classification.

Following “Management Information Base for the Differentiated Services Architecture” (www.ietf.org/rfc/rfc3289.txt), a classifier—set of rules—is bound to a particular interface, and filtering rules can vary between interfaces. There are two alternatives to support this model:

- Maintain several classifier instances, selected on a per-interface basis.
- Use a single classifier but extend the filtering rules with the interface number.

The presented design follows the second option because it reduces the number of required data structures and thus saves memory accesses.

30.2 Assumptions and Dependencies

The following design assumptions are made:

- The microblock does not access a packet header stored in DRAM. It assumes that the header is cached locally inside a microengine—for example, in GRP or transfer registers. The header caching should be assured in the dispatch loop source microblock.
- The microblock checks an IP header version but does not verify if the header is valid. IP header sanity checks should be performed in the IPv4 forwarder microblock.
- The microblock runs in one dispatch loop along with the IPv4 forwarder microblock.
- The microblock utilizes a hardware hash unit to speed up the classification process.
- Because of possible hash collisions, packets can get reordered within the microblock. It is the responsibility of a dispatch loop to impose the proper packet order.
- The default rule is available and configurable by a means of imported variables.
- The classification algorithm permits rules with “any” interface number.

30.2.1 Configuration Options

30.2.1.1 Build Switches

Table 6 1 identifies compile time build switches, which can be turned on or off as per the requirements of a specific application.

Table 30-1. Build Switches for IPv4 6-tuple Classifier Microblock

Symbol	Description
CLASSIFIER_6T_PACKET_COUNTER_FEATURE	Enables per rule statistics
CLASSIFIER_6T_HASH_BY_CRC	Enables using CRC32 instead of hash unit
CLASSIFIER_6T_DRAM_HASH_TABLE	Keep hash table in DRAM
CLASSIFIER_6T_SRAM_HASH_TABLE	Keep hash table in SRAM. If both SRAM_HASH_TABLE and DRAM_HASH_TABLE switches are defined or none of them, compile-time error is raised.
CLASSIFIER_6T_DEFAULT_STATS_ENABLE	Enables statistics for the default rule. The switch is only taken into account when PACKET_COUNTER_FEATURE is included.

30.2.1.2 Default Configuration

The build switches with which the microblock will be released are as follows:

- CLASSIFIER_6T_SRAM_HASH_TABLE

30.3 Microblock Design

This section describes the high level design for the IPv4 6-tuple classifier microblock.

30.3.1 Functionality

The microblock supports the following features:

- A microblock implements exact-match classification on IPv4 packets.
- A microblock supports up to 64k of exact-match rules. Each rule comprises:
 - Incoming interface number
 - IPv4 host source address
 - IPv4 host destination address
 - IP protocol
 - TCP/UDP/SCTP source port
 - TCP/UDP/SCTP destination port
 - Differentiated Services Code Point (6 bits of TOS field)
- A microblock supports one default rule. All packets that do not match any exact-match pattern are processed according to this rule.

- A rule (either exact-match or default) is associated with the following output:
 - QoS information (flow_id, class_id, color_id)
 - Forwarding information (next_hop_id, nexthop_id_type)
 - Identifier of a next block in a processing path (dl_next_block)
- A microblock does not classify packets with IP header options. Packets with IP header length other than 20 bytes are thrown to Intel XScale[®] core as exceptions.

30.3.2 Microblock Interfaces

30.3.2.1 Input Microblock Variables

Table 30-2 shows the input microblock variables.

Table 30-2. Input Variables Consumed

Variable	Size	Type ¹	Description
dl_input_port	16 bits	D	Logical number of an incoming interface, on which a packet was received.
dl_next_block	8 bits	D	Either hard-coded in a former block, or computed in run-time. It should be equal to BID_CLASSIFIER_6T. Otherwise, the classifier block executes an empty bypass path.
ip_header_in	24 bytes	X	An input xbuffer array with a cached IP header (20 bytes) and TCP/UDP ports (4 bytes).

1. D = dispatch loop variable, X = xbuf array

30.3.2.2 Output Microblock Variables

Table 30-3 shows the input microblock variables.

Table 30-3. Output Variables Modified

Variable	Size	Type ¹	Description
dl_next_block	8 bits	D	Identifier of a microblock that should continue with packet processing. The binding should be hard-coded in system.h as: <ul style="list-style-type: none"> • CLASSIFIER_6T_METER • CLASSIFIER_6T_MARK • CLASSIFIER_6T_FORWARD • CLASSIFIER_6T_INV_IP - next block for packets with IP header options (throw to Xscale or drop) The output block is configurable on a per-rule basis. In particular, this applies to a default rule (i.e. when a hash table lookup fails).
dl_flow_id	32 bits	D	Identifier of a traffic flow to which the packet belongs. Subsequent blocks can use this value to select a policy instance.
dl_class_id	16 bits	D	Identifier of a QoS aggregate that share the same ordering constraint (i.e. should be placed in the same queue). It is a relative queue number within an output port.
dl_color_id	2 bits	D	Packet drop precedence level: green, yellow or red color.
dl_nexthop_id_type	4 bits	D	Type of the next hop identifier (MPLS tunnel, IPv6 table, Layer-2 table index, etc.)

Table 30-3. Output Variables Modified (Continued)

Variable	Size	Type ¹	Description
dl_next_hop_id	16 bits	D	Identifier of forwarding information associated with a given packet flow. Subsequent blocks can use this value to forward selected packets to manually configured tunnels (e.g. MPLS TE).
exception_id	8 bits	D	The microblock sets the variable to its own ID (BID_CLASSIFIER_6T), when it sends an exception packet to Xscale (it is a packet with IP header options).
exception_code	8 bits	D	The microblock set the exception code only when it sends an exception packet to Xscale. The possible exception codes are:

1. D = dispatch loop variable

30.3.2.3 Imported Variables

This section lists variables that should be patched by a core component.

Table 30-4. Variables Imported by this Block

Variable	Default	Description
CLASSIFIER_6T_HASH_DRAM_BASECLASSIFIER_6T_HASH_SRAM_BASE	-	Base address of the hash table maintained in DRAM or SRAM (depending on compile-time option)
CLASSIFIER_6T_STATS_SRAM_BASE	-	Base address of the statistics table maintained in SRAM
CLASSIFIER_6T_HASH_KEY_MASK	16	Number of MSB bits from a hash key used as a hash table index.
CLASSIFIER_6T_DEFAULT_FLOW_ID	0	Default packet flow in case when no matching rule is found.
CLASSIFIER_6T_DEFAULT_CLASS_ID	8	Default relative output queue identifier in case when no matching rule is found.
CLASSIFIER_6T_DEFAULT_COLOR_ID	0 (green)	Default packet drop precedence level when no matching rule is found.
CLASSIFIER_6T_DEFAULT_NEXTHOP_ID	-1 (ID not known)	Default forwarding identifier in case when no matching rule is found.
CLASSIFIER_6T_DEFAULT_NEXTHOP_ID_TYPE	0 (IPv4 routing table)	Default next hop lookup table in case when no matching rule is found.
CLASSIFIER_6T_DEFAULT_NEXT_BLOCK	CLASSIFIER_6T_NEXT4 (in system.h)	Default output to send packets for which no matching rule is found. This can be bound to any of 4 classifier outputs. Depending on configuration packets that fail on lookup can be redirected to Xscale, or processed in ME.
CLASSIFIER_6T_DEFAULT_STATS_ADDRESS	-	Address of the statistics entry for default traffic.

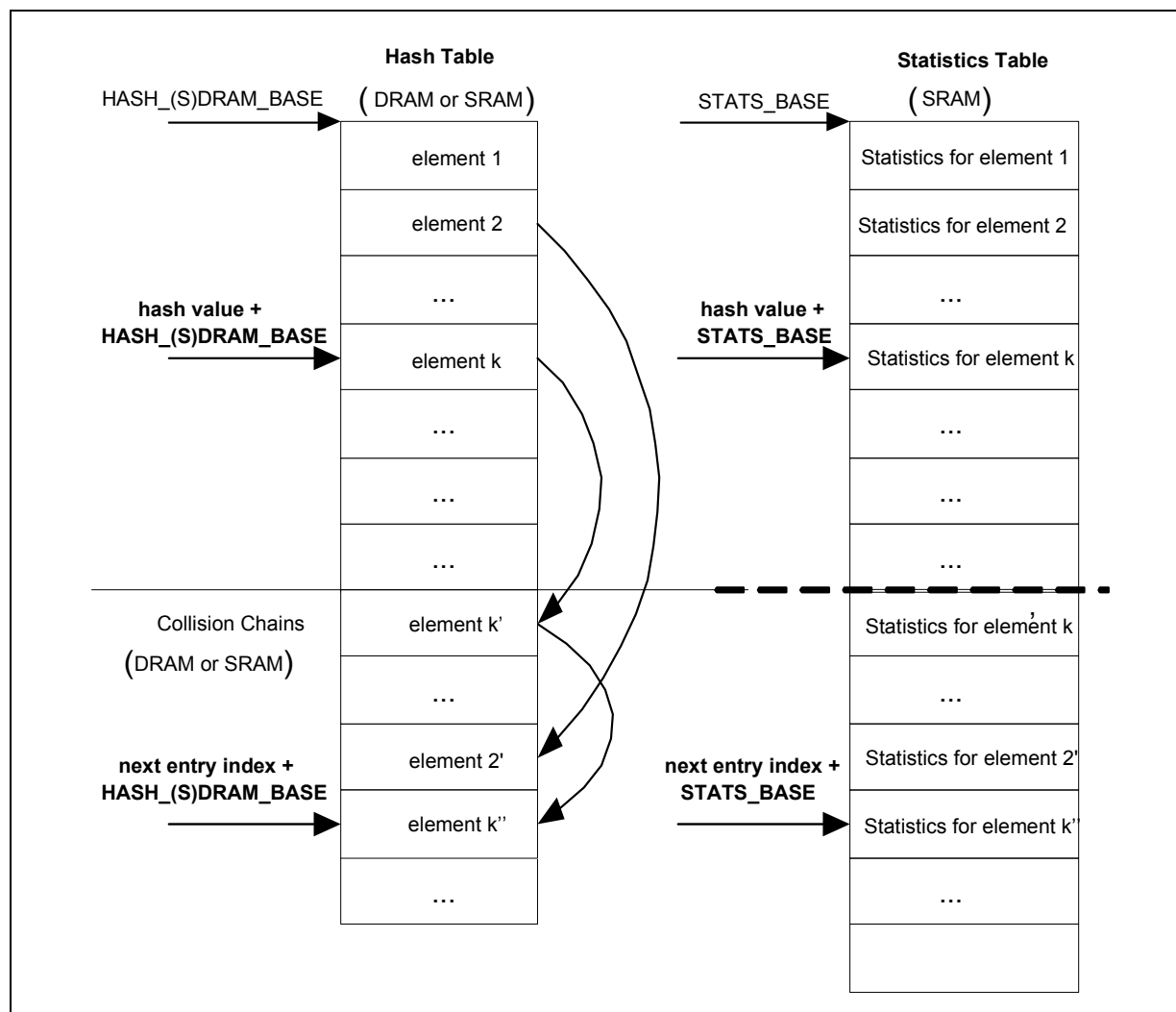
30.3.3 Data Structures

The 6-tuple classifier stores all rules in a hash table. The collisions are resolved by chaining ambiguous records. Due to possibly large size of a hash table, it is recommended to keep both the primary table and collision chains in DRAM memory. However, OC-48 performance goals may require using SRAM memory. To allow balancing between performance and costs, the memory storage type is provided as a compile-time option. The auxiliary collision table is located just after the hash table, so one base address is used.

Statistics counters are located in a separate table for the reasons:

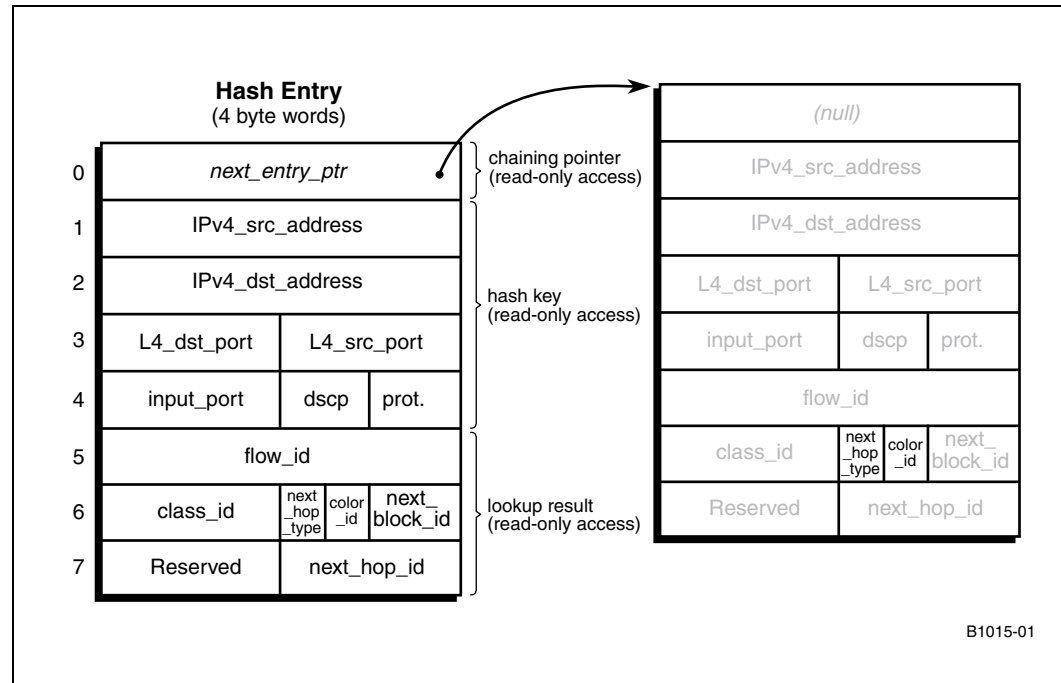
- Statistics gathering is implemented using SRAM atomic operations, so the table must be in SRAM,
- The hash table and the statistics table can be split among two SRAM channels.

Figure 30-1. Hash Table and Statistics Table Organization



Hash table elements and collision chain records have the same layout. A single record occupies eight 32-bit long words, as illustrated in Figure 30-2.

Figure 30-2. Hash Entry Layout



The microblock only reads hash entries, while the core component updates them. There is no special flag indicating that a hash entry is valid. Basically, a microblock checks if an entry is matching or not. Invalid (unused) entries should be initialized with all zeros, so that they never match with the processed packet.

Table 30-5. Hash Entry Definition

Field name	Size	Description
<code>next_entry_index</code>	32 bits	Index in collision table of the next entry in a chain. If this field contains a 0xFFFFFFFF value (all ones), no chained entry exists. The address of next entry is calculated in the same way as address of an entry in the hash table, but <code>next_entry_index</code> is used instead of hash index.
<code>IPv4_src_address</code>	32 bits	IPv4 source address of the exact-match rule.
<code>IPv4_dst_address</code>	32 bits	IPv4 destination address of the exact-match rule.
<code>L4_src_port</code>	16 bits	Source port number of a layer 4 protocol (e.g. UDP or TCP).
<code>L4_dst_port</code>	16 bits	Destination port number of a layer 4 protocol (e.g. UDP or TCP).
<code>input_port</code>	16 bits	Logical number of an interface on which the packet has been received.
<code>dscp</code>	8 bits	DiffServ codepoint stored in Type of Service field of an IPv4 header. Only 6 left-most bits are significant. Two remaining bits must be 0.
<code>protocol</code>	8 bits	Layer 4 protocol number (e.g. TCP or UDP)
<code>flow_id</code>	32 bits	Identifier of a traffic flow to which the packet belongs. Subsequent blocks can use this value to select a policy instance. 0xFFFFFFFF value of the field means that the entry does not contain valid QoS info.
<code>class_id</code>	16 bits	A relative queue number within an output port.

Table 30-5. Hash Entry Definition (Continued)

Field name	Size	Description
statistics_flag (S)	1 bit	Indicates whether the microblock should gather statistics for the rule. This parameter is used only when the microblock is compiled with statistics handling.
reserved	1 bit	Not used now.
nexthop_type	4 bits	Type of the next hop identifier (MPLS tunnel, IPv6 table, Layer-2 table index, etc.)
color_id	2 bits	Packet drop precedence level: green, yellow or red color.
next_block_id	8 bits	Identifier of a next microblock in a processing pipeline.
next_hop_id	16 bits	Identifier of a next hop information used to forward packets. 0xFFFF value of the field means that the entry does not contain valid forwarding info.

Statistic entries in the statistics table consist of two 64-bit counters. The first counter stores a number of packets matching the rule. The second counter keeps a number of bytes matching the rule. The statistics entries have the same indexes as the corresponding entries in the hash table. The statistics counters for default rule are placed in SRAM, at the end of statistics table. It is the statistics table has one entry more than hash table.

30.3.4 Flow Chart

30.3.4.1 Synchronization

The classifier microblock does not operate on shared data structures. Thus, there is no need for inter-thread synchronization. The microblock is organized as a functional pipe-stage. All threads in a microengine execute the same algorithm, but they process different packets.

30.3.4.2 Classification algorithm

As illustrated in [Figure 30-3](#), an algorithm first checks if the `dl_next_block` variable points to the classifier. If not, a bypass path is executed. If next block refers to this microblock, it checks if the processed packet is IPv4 with a header length of 20 bytes. The microblock does not process IPv4 packets with header options. Such packets, if detected, are thrown to Xscale core component as exceptions.

Next the microblock checks if the processed packet is fragmented. Packets that are not fragmented have both More Fragments header flag and Fragment Offset header field equal to zero. If the packet is fragmented, it is classified to the default class without performing 6-tuple exact match lookup.

For non-fragmented packets, the microblock builds a hash input from the 6-tuple value retrieved from the header cache. If the microblock is compiled with `CLASSIFIER_6T_HASH_BY_CRC` option, it calculates the hash result using CRC 32 instruction. Otherwise, it issues 128-bit hash calculation request to the hash hardware unit.

When the hash result is ready, a classifier takes only `HASH_KEY_MASK` most significant bits (default is 16) of the first returned long word. These bits are shifted right so they form a DRAM/SRAM offset to a hash table entry. A thread loads the pointed entry and swaps out waiting for a read completion. After the hash entry is retrieved, it compares IP header fields with values stored in

the hash entry. On successful verification, a thread checks if a packet was received from the allowed input port. According to DiffServ MIB assumptions, the input port can be specified as either exact number or a wildcard value (0xFFFF).

If a matching entry is found, the classifier writes selected dispatch loop variables with data stored in a hash entry. Otherwise, if verification fails the algorithm loads the next hash entry in a chain, as indicated by `next_entry_ptr`, and repeats the entry verification procedure. If a classifier reaches end of chain (`next_entry_ptr` set to null) without finding a matching entry, a default rule is applied.

The algorithm supports exact-match classification on an IP header, as well as wildcard classification on an input port. Such combination accommodates usage scenarios where only a packet header is known, but not the input port (e.g. IP tunnel endpoints). It also works well with DiffServ, unless a given 6-tuple rule is configured on selected subset of external ports. However, such configurations are expected to be very rare: a rule belongs either to a specific input port or to all interfaces.

Figure 30-3. 6-tuple Classifier Algorithm (Page 1 of 3)

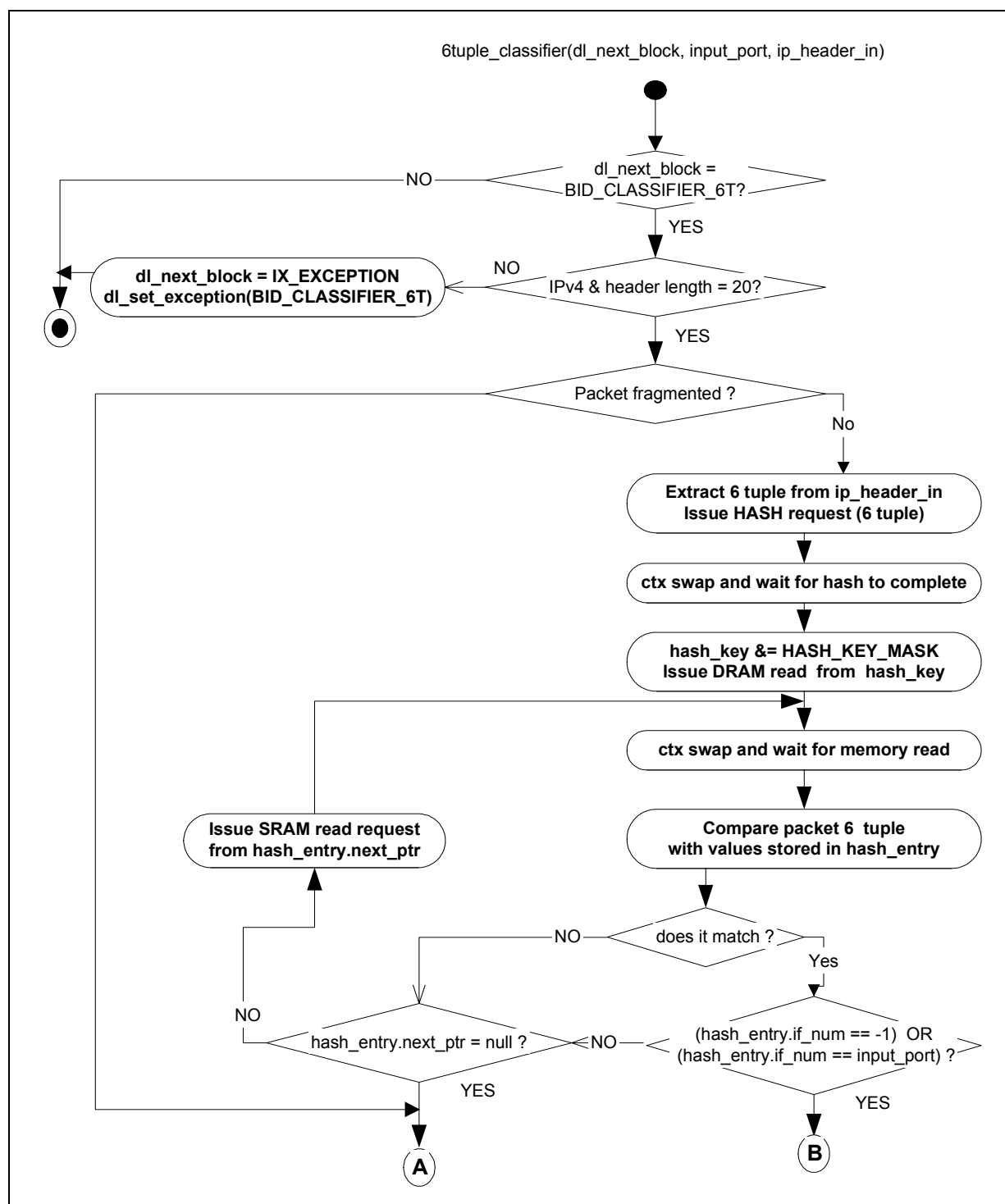


Figure 30-4. 6-tuple Classifier Algorithm (Page 1 of 3)

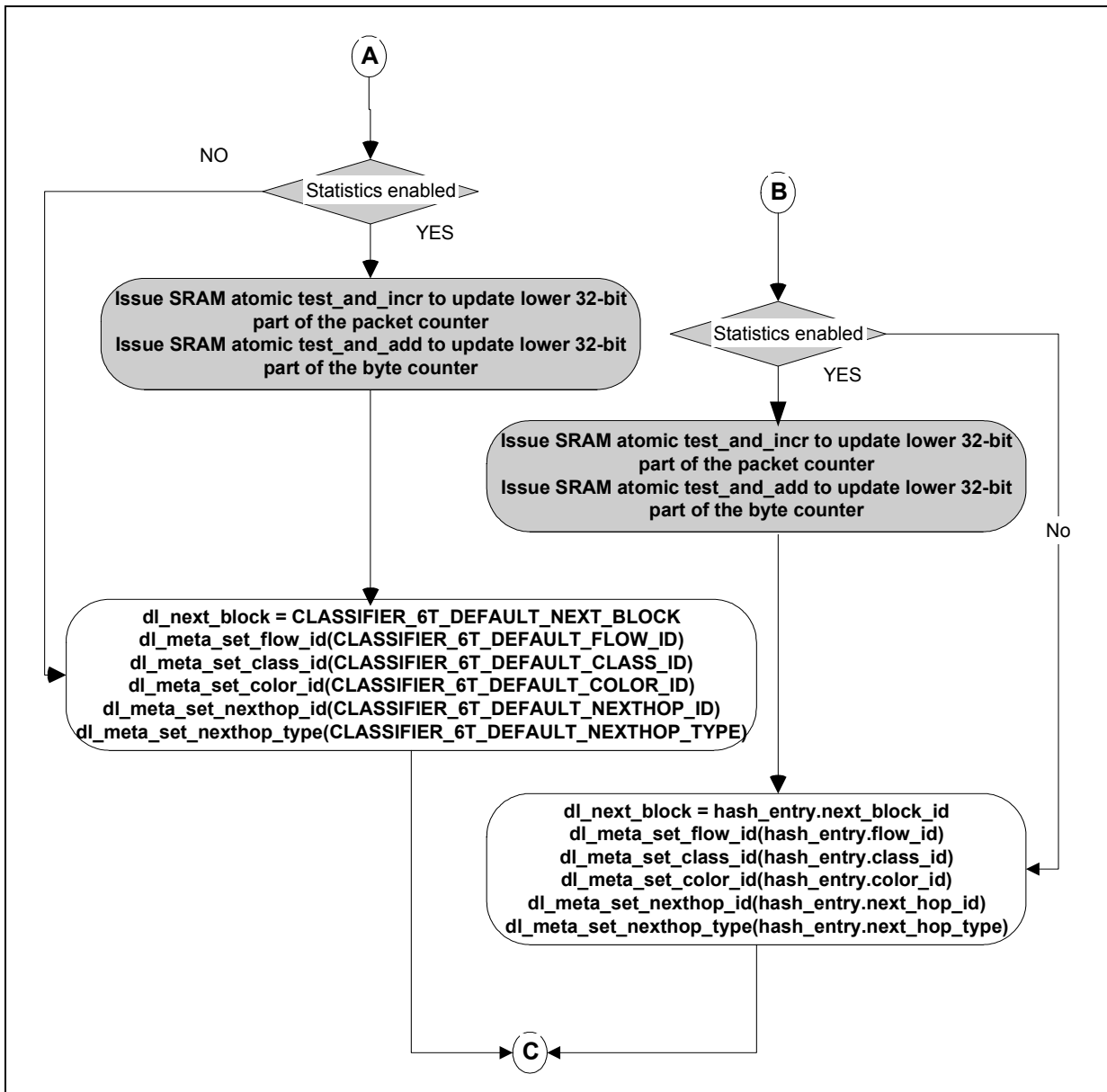
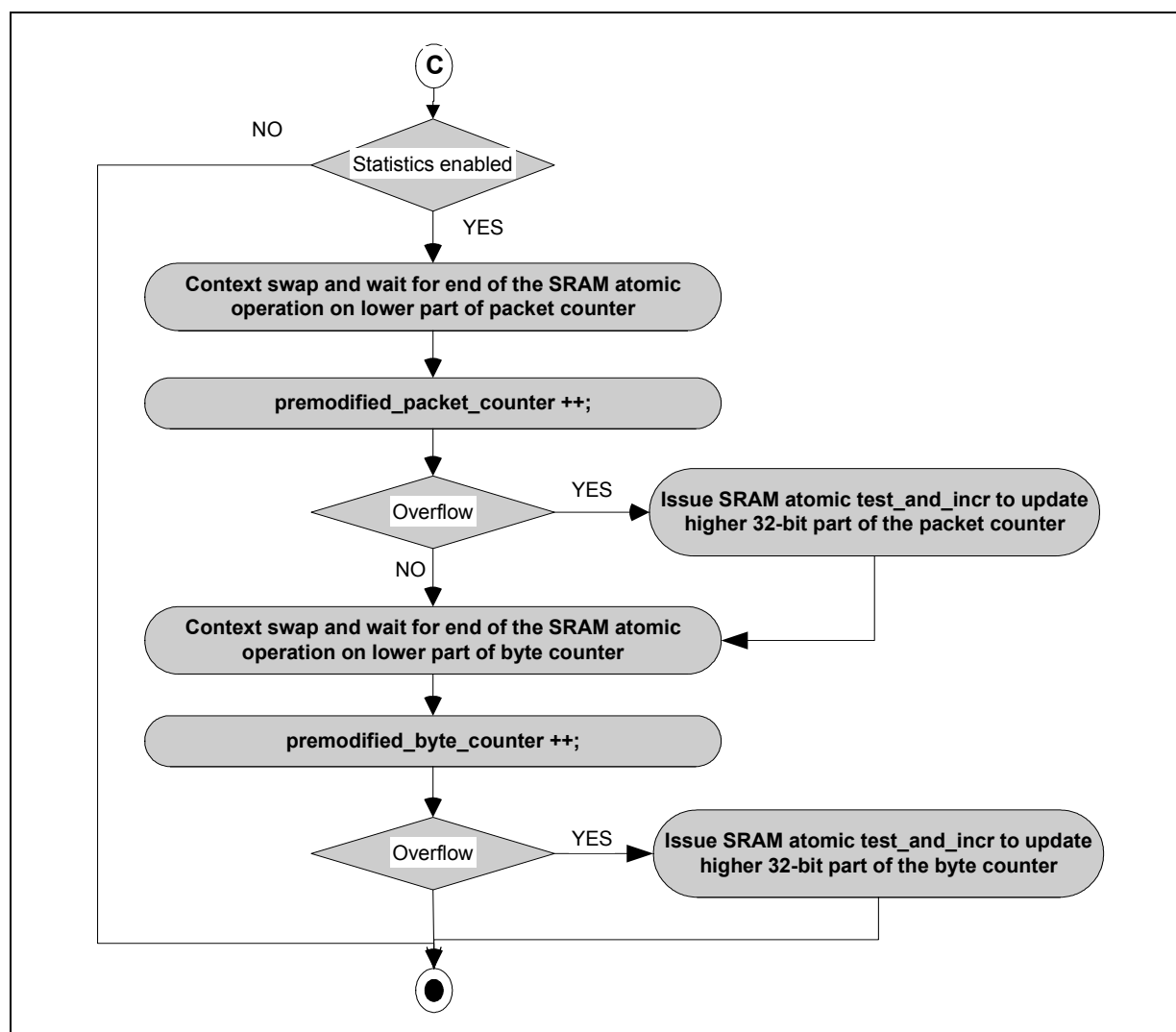


Figure 30-5. 6-tuple Classifier Algorithm (Page 3 of 3)



30.3.4.3 Statistics Gathering

The part of algorithm responsible for gathering statistics is shown in Figure 6 3 in gray. The code is present only if the microblock is compiled with statistics handling feature. The microblock does not use the the Xscale-based statistics gathering scheme described in [IXA SDK]. This scheme assumes that the 64-bit counter is kept by the core component while a microblock uses 32-bit auxiliary variable.

The scheme does not scale well for large number of counters since the core component issues too many SRAM atomic operations. Another method for implementing 64-bit counters is presented in SRTCM design (see [Section 31.6.2, “SRTCM Algorithm” on page 553](#)).

Therefore the 6-tuple classifier microblock updates the whole 64-bit counter. It updates the lower 32-bit counter part using SRAM atomic operations that return the premodified value. This makes it possible to detect overflow of the lower part. If such overflow occurs, the microblock issues atomic increment operation for the higher counter part.

This scheme ensures that the SRAM operations are issued only when update of counters is needed.

Note that the microblock should request atomic operation on the lower part of the counters just after it finds the configuration entry. This method shortens the idle time of waiting for the operation to complete.

30.4 Microcode Budget

30.4.1 Performance Analysis

Tables 30-6, 30-7, and 30-8 reflect the current version of code prototyping.

Tables 30-6 and 30-7 does not take into account the SRAM atomic increment operation on the most significant 32-bit part of the counters. The most significant part of the byte counter is updated at most every 466034 packets and the most significant part of the packet counter is updated every 4294967296 packets.

Table 30-6. Cycle Count Table (including unfilled defers)

Phase	Best case (don't match)	Worst case (match)
Check header & extract 6-tuple	14	14
Issue hash request	14	14
Parse DRAM/SRAM ¹ entry (one pass)	10	15
Write results	8	10
Total	46	53
Statistics overhead	19	24

1. Depends on compile-time option that determines memory storage type

Table 30-7. I/O Latency Analysis Table

Operation	I/O type	Accesses	Transfer size (in 32-bit long words)
Hash entry read from DRAM/SRAM ¹	DRAM/SRAM read	1-? ²	8
Hash request	hash	0 or 1 ³	4
Byte counter update	SRAM atomic test_and_add	0 or 1 ⁴	1
Packet counter update	SRAM atomic test_and_incr	0 or 1 ⁴	1

1. depends on compile-time option that determines memory storage type

2. maximum value depends on the collision number

3. depends on compile-time option that determines method of hash index calculation (hash unit or CRC32)

4. depends on compile-time option that determines whether the microblock supports statistics

Table 30-8. Memory Accesses Summary Table

I/O type	Accesses
SRAM/DRAM ¹	1-?
Hash unit	0 or 1

1. depends on compile-time options that determine memory storage type

30.4.2 Memory Footprint Analysis

Table 30-9. DRAM/SRAM Footprint

DRAM Data structures	Size (bytes)
Hash table (entry size is 8 long words)	2 MB (64k entries, each entry has 32 bytes)
Collision chains (entry size is 8 long words)	1 MB (32k entries, each entry has 32 bytes)
SRAM statistics table (entry size is 4 long words)	0 ¹ or 1 MB (64k + 32k entries, each entry has 16 bytes)
Total	3 MB

1. depends on compile-time option that determines whether the microblock supports statistics

Table 30-10. Code Store Footprint

Code Store	Size (instruction)
Total	69 (97 with statistics) + 19 initialization

31.1 Overview

The SRTCM functionality is standardized by “A Single Rate Three Color Marker” (www.ietf.org/rfc/rfc2697.txt), and the following citation, taken from it, describes the metering algorithm:

The behavior of the SRTC meter is specified in terms of its mode and two token buckets, C and E, which both share the common rate CIR (bytes/sec.). The maximum size of the token bucket C is CBS (bytes) and the maximum size of the token bucket E is EBS (bytes).

The token buckets C and E are initially (at time 0) full, i.e., the token count TC = CBS and the token count TE = EBS. Thereafter, the token counts TC and TE are updated CIR times per second as follows:

- If TC is less than CBS, TC is incremented by one, else
- If TE is less than EBS, TE is incremented by one, else
- Neither TC nor TE is incremented.

When a packet of size B bytes arrives at time t, the following happens if the srTCM is configured to operate in the Color-Blind mode:

- If $TC - B \geq 0$, the packet is green and TC is decremented by B down to the minimum value of 0, else
- If $TE - B \geq 0$, the packet is yellow and TE is decremented by B down to the minimum value of 0, else
- The packet is red and neither TC nor TE is decremented.

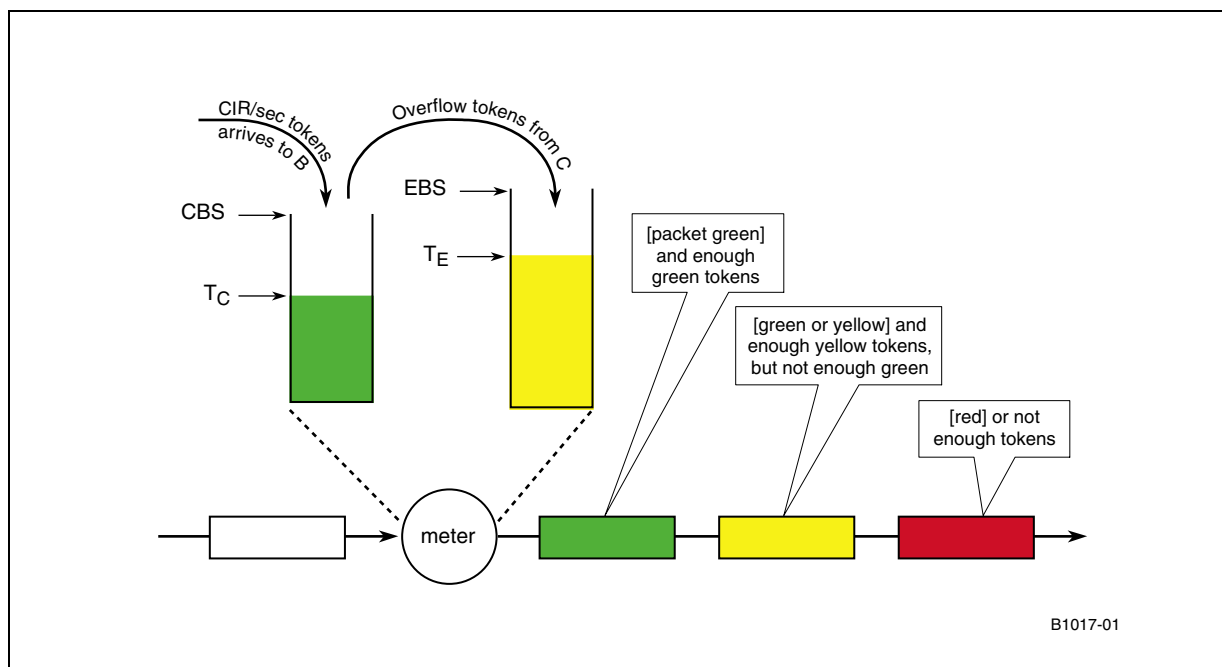
When a packet of size B bytes arrives at time t, the following happens if the srTCM is configured to operate in the Color-Aware mode:

- If the packet has been precolored as green and $TC - B \geq 0$, the packet is green and TC is decremented by B down to the minimum value of 0, else
- If the packet has been precolored as green or yellow and if $TE - B \geq 0$, the packet is yellow and TE is decremented by B down to the minimum value of 0, else
- The packet is red and neither TC nor TE is decremented.

Note that according to the above rules, marking of a packet with a given color requires that there be enough tokens of that color to accommodate the entire packet. Other marking policies are clearly possible. The above policy was chosen in order to guarantee a deterministic behavior where the volume of green packets is never smaller than what has been determined by the CIR and CBS, i.e., tokens of a given color are always spent on packets of that color.

Figure 31-1 illustrates the color-aware mode of the SRCTM algorithm. The conditions embedded in “[]” brackets refer to the color-aware part of the algorithm. Without these sections, the algorithm operates in the color-blind mode.

Figure 31-1. Color-aware and Color-blind Modes of SRTCM



Functionality of TRTCM is standardized in [RFC 2698]. The following citation taken from [RFC 2698] describes the metering algorithm.

The behavior of the Meter is specified in terms of its mode and two token buckets, P and C, with rates PIR and CIR, respectively. The maximum size of the token bucket P is PBS and the maximum size of the token bucket C is CBS.

The token buckets P and C are initially (at time 0) full, i.e., the token count $T_p(0) = PBS$ and the token count $T_c(0) = CBS$. Thereafter, the token count T_p is incremented by one PIR times per second up to PBS and the token count T_c is incremented by one CIR times per second up to CBS.

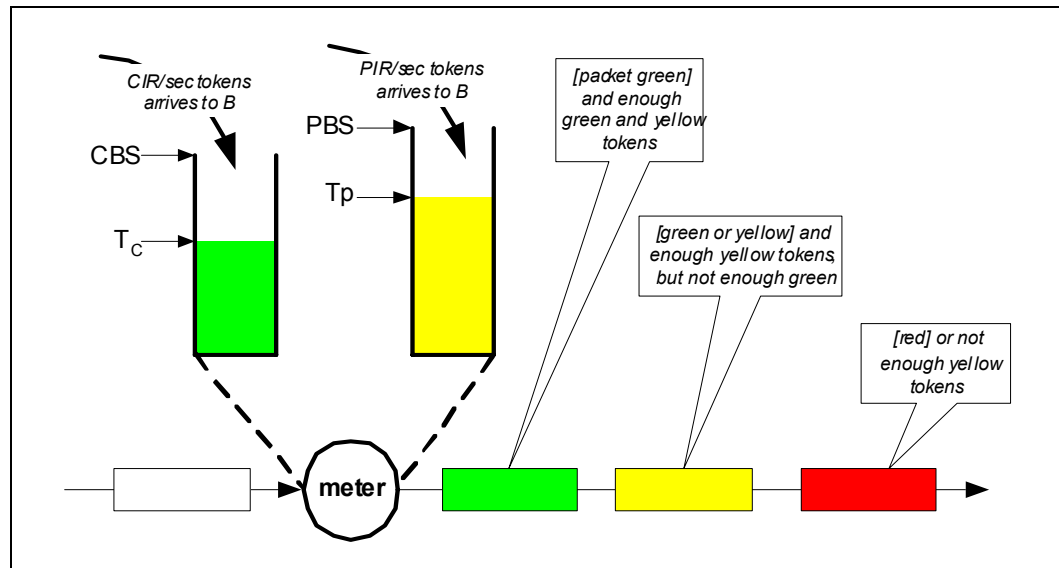
When a packet of size B bytes arrives at time t, the following happens if the trTCM is configured to operate in the Color-Blind mode:

- If $T_p(t) - B < 0$, the packet is red, else
- if $T_c(t) - B < 0$, the packet is yellow and T_p is decremented by B, else
- The packet is green and both T_p and T_c are decremented by B.

When a packet of size B bytes arrives at time t, the following happens if the trTCM is configured to operate in the Color-Aware mode:

- If the packet has been precolored as red or if $T_p(t) - B < 0$, the packet is red, else
- If the packet has been precolored as yellow or if $T_c(t) - B < 0$, the packet is yellow and T_p is decremented by B, else
- The packet is green and both T_p and T_c are decremented by B.

Table 31-2 illustrates the color-aware mode of the TRCTM algorithm. The conditions embedded in “[]” brackets refer to the color-aware part of the algorithm. Without these sections, the algorithm operates in the color-blind mode.

Figure 31-2. Color-aware and Color-blind Modes of TRTCM


31.2 Functionality

The microblock supports the following features:

- Color-blind version of A Single Rate Three Color Marker (www.ietf.org/rfc/rfc2697.txt).
- Color-aware version of A Single Rate Three Color Marker (www.ietf.org/rfc/rfc2697.txt).
- Color-blind version of the A Two Rate Three Color Marker (www.ietf.org/rfc/rfc2698.txt).
- Color-aware version of the A Two Rate Three Color Marker (www.ietf.org/rfc/rfc2698.txt).
- 64-bit packet and byte counters, gathered on a per-flow and per-color basis.

31.3 Assumptions and Dependencies

31.3.1 Assumptions

- The same microblock implements the two algorithms, because in normal condition traffic flows are metered using either SRTCM or TRTCM algorithm. Since implementation of both algorithms requires critical section, their implementation in separate microblocks would introduce unnecessary delays caused by running void synchronization in one of the blocks.
- The type of metering algorithm run by the microblock can be chosen by a compile time option, or can be chosen as a configuration parameter of a meter instance. This means that the block can be compiled in SRTCM version, TRTCM version or mixed version. In the latter case, a configuration parameter decides which algorithm is run.
- The color-aware mode is available as a compile-time option. When disabled, the microcode runs in a color-blind mode.

- The microblock uses `flow_id` to select a meter instance—that is, TCM parameters.
- The microblock can process packets of any type—for example, IP or MPLS. It does not access a packet header. For IP packets, a standalone DSCP marker updates a TOS field. For MPLS packets, the MPLS marker updates EXP bits.
- The microblock does not generate exception packets. It provides three outputs: two outputs for packets to be passed (for example, packets marked by DSCP marker and packets marked with MPLS marker) and one output for dropped packets. The outputs are configured on a per-flow basis.
- The statistics can be disabled by a compile-time option (all statistics) or dynamically in run-time (selected flows).
- Statistics updates are executed within a critical section, as described in [Section 8.7, “Sending Packets from Core Components to Microblocks”](#) on page 112. The algorithm does not utilize SRAM atomic operations.
- The microblock has three outputs: TCM_NEXT1, TCM_NEXT2 and TCM_NEXT3. The output through the traffic exits the microblock is configurable. The outputs differ in time the packets exiting through the output are processed. The fastest output is TCM_NEXT1. The slowest output is TCM_NEXT3. In MPLS/DiffServ pipeline, the SRTCM_NEXT1 output is bound to the DSCP marker block; the TCM_NEXT2 output is bound to the IPv4 forwarder block; and the TCM_NEXT3 block is bound to the IX_DROP constant.

31.3.2 Configuration Options

31.3.2.1 Build Switches

Table 8 1 identifies compile time build switches, which can be turned on or off as per the requirements of a specific application.

Table 31-1. Build Switches for TCM Microblock

Symbol	Description
TCM_PACKET_COUNTER_FEATURE	Enables statistics gathering
TCM_COLOR_AWARE_FEATURE	Enables color-aware metering
TCM_SRTCM_FEATURE	Enables SRTCM algorithm as defined in [RFC2697]
TCM_TRTCM_FEATURE	Enables TRTCM algorithm as defined in [RFC2698]
TCM_DISABLE_FIN_SYNC	Disables final thread synchronization. The feature should be turn on, if the next microblock starts with thread synchronization.

31.3.2.2 Default Configuration

The build switches with which the microblock will be released are as follows:

- TCM_SRTCM_FEATURE
- TCM_TRTCM_FEATURE

31.4 Microblock Interfaces

31.4.1 Input Microblock Variables

Table 31-2. Input Variables Consumed by SRTCM

Variable	Size in bits	Type ¹	Description
dl_next_block	8	D	Either hardcoded in the previous block, or obtained during classification stage. If not equal to BID_TC_METER, the TCM block executes an empty bypass path.
dl_flow_id	16	D	Identifier of a meter instance (entry index in SRAM table). The value must be other than 0, if dl_next_block points to TCM. Otherwise, meter behavior is unpredictable (CAM lookup attempt with value 0).
dl_packet_size	16	D	Number of bytes in the currently processed packet, used to consume tokens from buckets and to determine a packet color.
dl_color_id	2	D	Incoming packet color. This value is relevant for color-aware mode only and should be in range: 0 = green, 1 = yellow or 2 = red. Only two least significant bits of dl_color_id are important; other bits are masked out. If dl_color_id equals 3, a red color is applied.

1. D = dispatch loop variable

31.4.2 Output Microblock Variables

Table 31-3. Output Variables Modified by SRTCM

Variable	Size in bits	Type ¹	Description
dl_next_block	8	D	Identifier of a next block that performs selected action on a packet. The binding should be hard-coded in <code>system.h</code> as TCM_NEXT1 (e.g. DSCP mark packet), TCM_NEXT2 (e.g. MPLS mark packet) or TCM_NEXT3 (e.g. drop packet).
dl_flow_id	16	D	The outgoing PHB to be marked in a packet header (i.e. DiffServ DSCP field).
dl_color_id	2	D	Outgoing packet color (green = 0, yellow = 1, red = 2).

1. D = dispatch loop variable

31.4.3 Imported Variables

Table 31-4. Variables Imported by this Block

Variable	Default	Description
TCM_TABLE_SRAM_BASE	—	Base address of the meter instance table maintained in SRAM.
TCM_64BIT_STAT_SRAM_BASE	—	Base address of the statistics table storing most significant parts of 64-bit long counters.

31.5 Data Structures

The TCM block maintains a meter table in SRAM memory. The table is a simple array indexed by a `flow_id`. A table entry occupies 16 long words and contains SRTCM or TRTCM parameters configured for a single meter instance. The parameters include both externally configurable attributes (as described in “A Single Rate Three Color Marker”—www.ietf.org/rfc/rfc2697.txt and “A Two rate Three Color Meter”—www.ietf.org/rfc/rfc2698.txt) as well as internal variables. Figure 31-3 illustrates layout of SRAM data structures when the microblock is compiled with statistics and Figure 31-4 illustrates layout of SRAM data structures when the microblock is compiled without statistics.

Figure 31-3. TCM Data Structures with Statistics

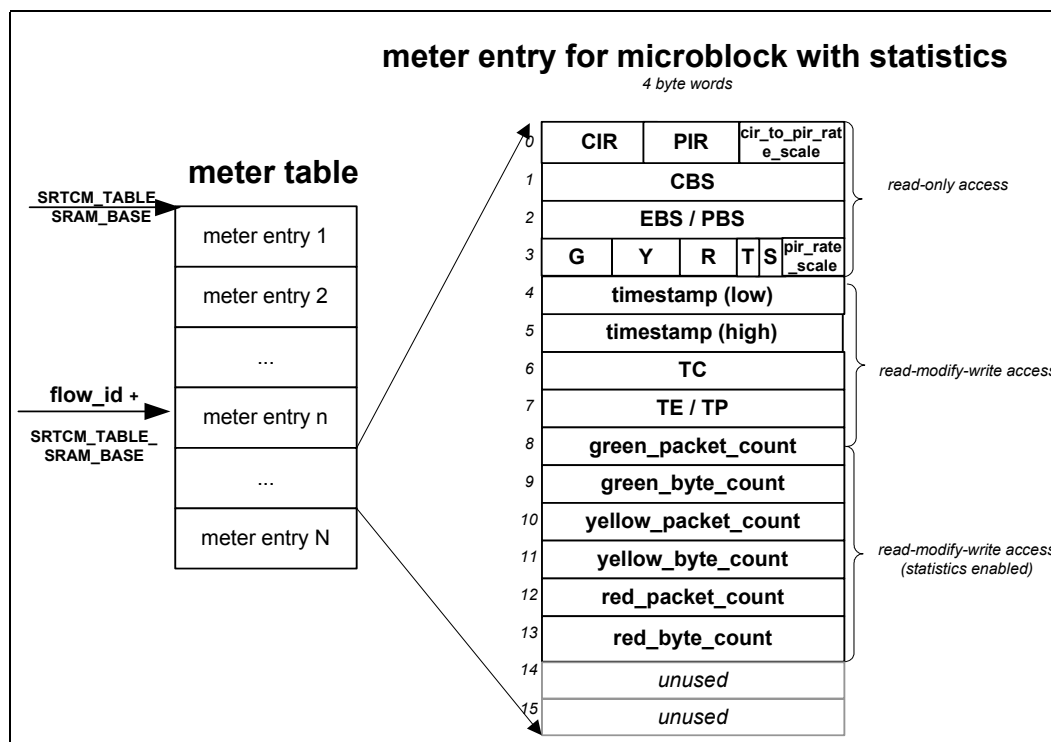
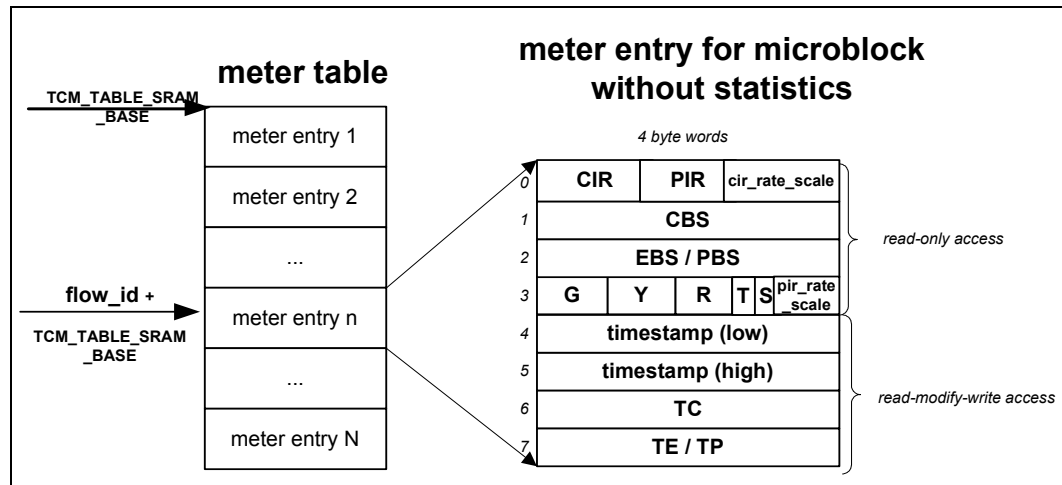
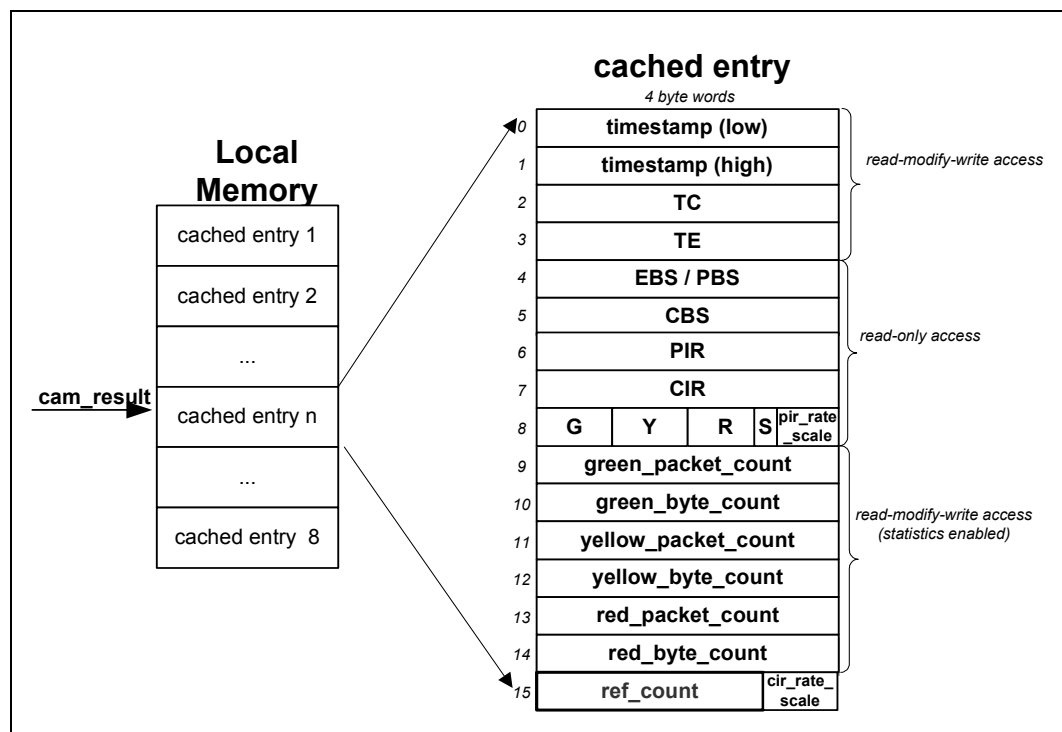


Figure 31-4. TCM Data Structures without Statistics



The microblock caches a meter entry in Local Memory. The cached data contains the same information as the corresponding SRAM entry. The LM layout is slightly rearranged so that fields not aligned to a 4-byte boundary are stored in lower 8 long words. This enables efficient usage of MEv2 instructions that operate on particular bytes of a 32-bit word.

Figure 31-5. SRTC Meter Data Structures in Local Memory



In addition, the last long word in the Local Memory entry contains a reference counter. It counts the number of threads that access a meter instance. When a counter drops to 0, the cached entry is flushed from Local Memory to SRAM.

Table 31-5. SRTCM Table Entry Definition

Field name	Size	Description
timestamp	64 bits	The MEv2 timestamp of the last arrived packet from the same traffic flow shifted left <code>pir_rate_scale</code> bits. Needed to update the TC and TE parameters.
TC	32 bits	Available token count in the C bucket (in bytes).
TE / TP	32 bits	Available token count in the E bucket, for SRTCM, or P bucket, for TRTCM (in bytes).
CIR	13 bits	Committed Information Rate at which a traffic source is signed up to send packets to the meter instance. It is measured in bytes/timestamp, shifted left by <code>pir_rate_scale + cir_to_pir_rate_scale</code> bits to accommodate slow flows. Zero value of the field means that the entry is empty.
PIR	14 bits	Peak Information Rate at which a traffic source is signed up to send packets to the meter instance. It is measured in bytes/timestamp, shifted left by <code>rate_scale</code> bits to accommodate slow flows.
CBS	32 bits	Committed Burst Size; a maximum value of TC (in bytes).
EBS / PBS	32 bits	Excess Burst Size; a maximum value of TE (in bytes), when SRTCM is used, or Peak Burst Size; a maximum value of TP (in bytes), when TRTCM is used
green_dscp (G)	8 bits	Action and packet mark value for green packets. Bits 7..2 are written to outgoing <code>flow_id</code> . Bits 1..0 determine an action to be taken; they select a meter output: TCM_NEXT1 (01b), TCM_NEXT2 (10b) or TCM_NEXT3(11b).
yellow_dscp (Y)	8 bits	Action and packet mark value for yellow packets. Bits 7..2 are written to outgoing <code>flow_id</code> . Bits 1..0 determine an action to be taken; they select a meter output: TCM_NEXT1 (01b), TCM_NEXT2 (10b) or TCM_NEXT3(11b).
red_dscp (R)	8 bits	Action and packet mark value for red packets. Bits 7..2 are written to outgoing <code>flow_id</code> . Bits 1..0 determine an action to be taken; they select a meter output: TCM_NEXT1 (01b), TCM_NEXT2 (10b) or TCM_NEXT3(11b).
meter_type_flag (T)	1 bit	Flag indicating if the meter instance is SRTCM (T=0) or TRTCM (T=1). The flag is only used when the microblock is compiled in the mixed version.
statistics_flag (S)	2 bits	Flag indicating if statistics are enabled (S=1) or not (S=0). Only the most significant bit is used.
pir_rate_scale	5 bits	<ul style="list-style-type: none"> the microblock is compiled in SRTCM version or the microblock is compiled in mixed version and the <code>meter_type_flag</code> indicates SRTCM algorithm.
cir_rate_scale	5 bits	A factor used to pre-scale the packet interarrival time so that it matches the shifted CIR value. It must be in range from 0 to 31 bits.
green_pkt_count	32 bits	Less significant 32 bits of Green Packets Counter (in packets).
green_byte_count	32 bits	Less significant 32 bits of Green Bytes Counter (in bytes).
yellow_pkt_count	32 bits	Less significant 32 bits of Yellow Packets Counter (in packets).
yellow_byte_count	32 bits	Less significant 32 bits of Yellow Bytes Counter (in bytes).
red_pkt_count	32 bits	Less significant 32 bits of Red Packets Counter (in packets).
red_byte_count	32 bits	Less significant 32 bits of Red Bytes Counter (in bytes).

31.6 Flow Chart

31.6.1 Synchronization

All microengine threads execute the SRTCM algorithm in parallel. They use the folding technique—refer to the *Intel® Internet Exchange Architecture Portability Framework Developer's Manual*—combined with CAM lookup to implement a critical section. In addition, the following assumptions is taken:

- Only the first thread—thread 0—loads the current timestamp into global registers; all other threads use the same value.
- Only on CAM miss, a thread updates tokens in buckets. With this approach the packet inter-arrival time between two consecutive CAM hits is treated as 0.

Figure 31-6. TCM Inter-thread Synchronization (Page 1 of 2)

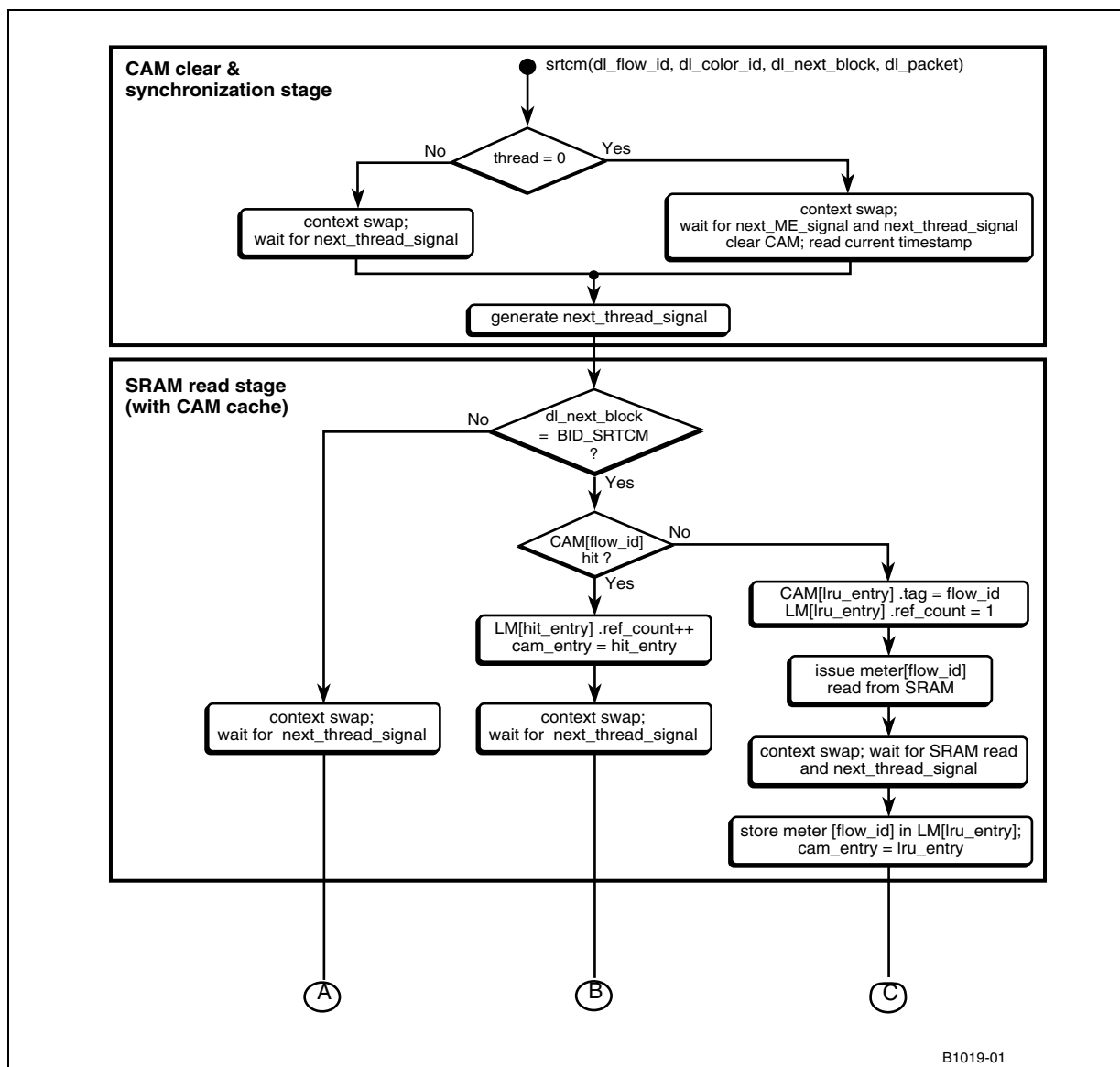
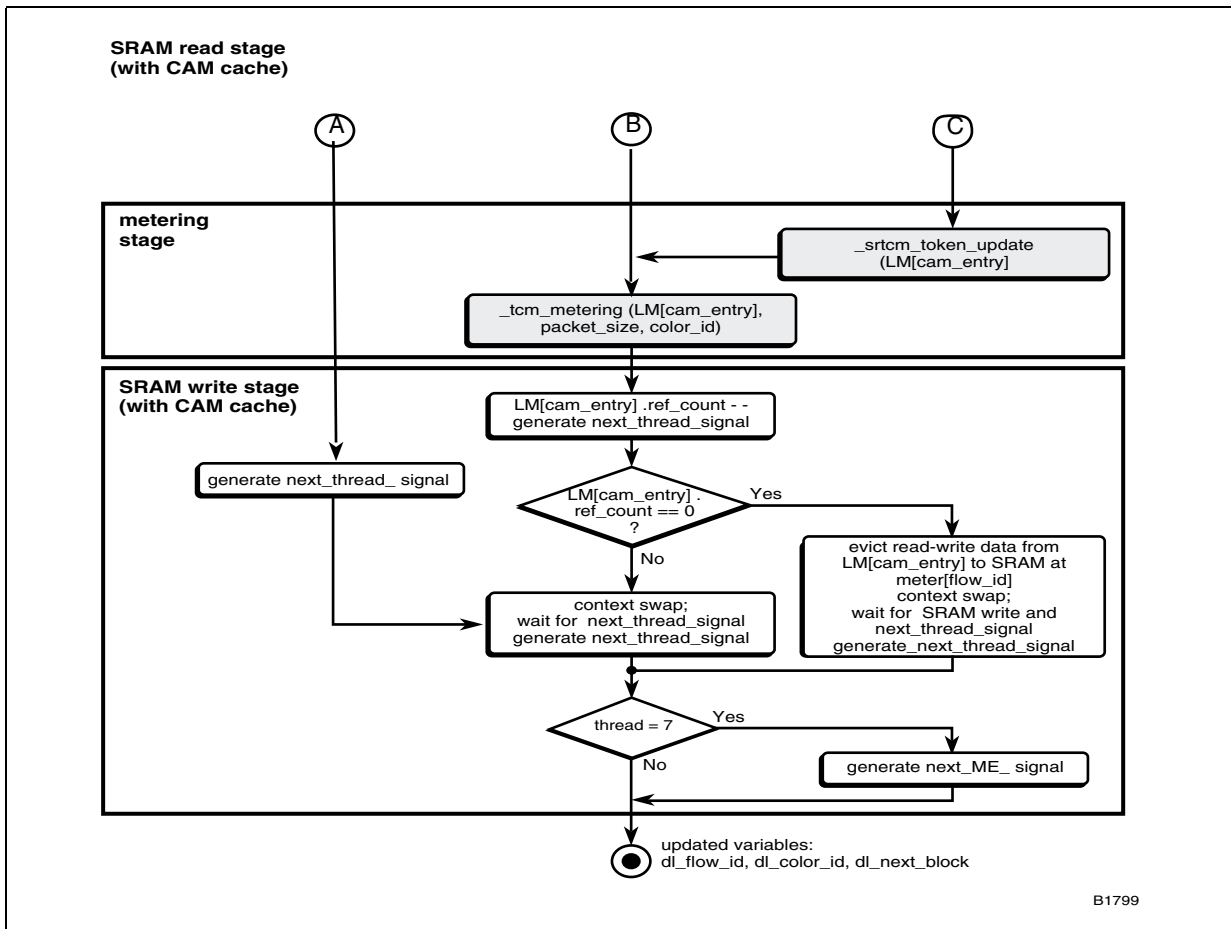


Figure 31-7. TCM Inter-thread Synchronization (Page 2 of 2)



The final context swap handshake (grayed code) assures that the critical section is kept as short as possible. This code can be compiled-out if the threads are to be synchronized on entry to the subsequent block anyway.

31.6.2 SRTCM Algorithm

Generally, the SRTCM implementation adheres to “A Single Rate Three Color Marker” (www.ietf.org/rfc/rfc2697.txt). In addition, the following design-level assumptions are taken:

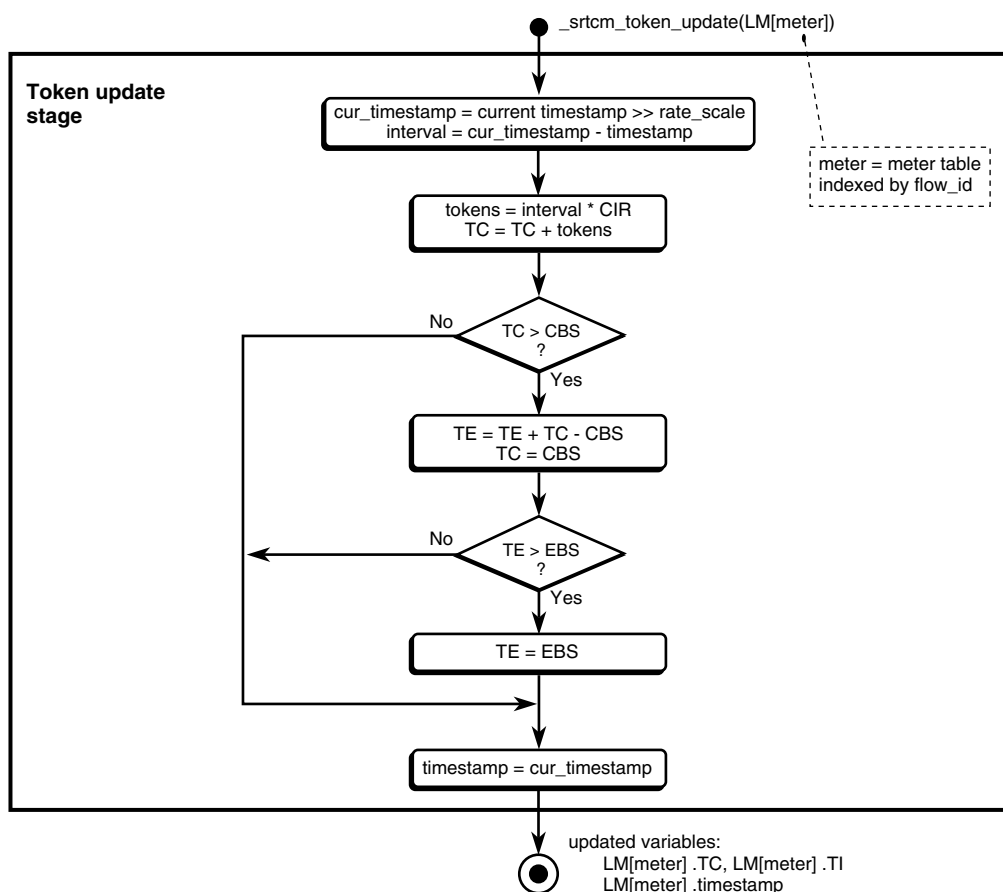
- Tokens are updated on each packet arrival, not periodically.
- Color-aware mode is implemented (with a compile-out possibility).
- Statistics are gathered (with a compile-out possibility).

31.6.2.1 Per-packet Token Updates

“A Single Rate Three Color Marker” (www.ietf.org/rfc/rfc2697.txt) requires the token counts TC and TE to be updated CIR times per second. Periodic token updates bring in performance concerns in IXP2400 and IXP2800 Network Processors. Basically, such solution would require either a dedicated microengine thread that continuously updates the metering table entries, or a background task running on Intel XScale® core. The first solution is not applicable since the SRTCM block is a part of a functional pipe stage. All threads constituting a functional pipeline have to perform the same operations. On the contrary, a dedicated task on Intel XScale® core would have to inspect meter entries quite frequently due to possibly high flow rates. As a result, a significant amount of SRAM memory accesses would occur, which is critical to the fast path performance.

Alternatively, token counters can be updated whenever a new packet arrives to a given flow. The per-packet token update algorithm works as follows:

Figure 31-8. SRTCM Token Update Macro



B1020-01

The rate scaling factor allows computing TC/TE updates for underflows. The essential step in updating these parameters is to compute the packet inter-arrival time (current timestamp – previous timestamp). The basic time unit in Mev2 timestamp equals to 16 Mev2 clock cycles. For underflows, the CIR value expressed in bytes/timestamp could be a fractional value—for example, 64 kbit/s \approx 0.0002 bytes/timestamp. Conversely, the mean packet interval for slow flows would be a large value if expressed in timestamp units. To avoid multiplication of very small and very big values, CIR is pre-scaled up by $2^{\text{rate_scale}}$ during configuration, and the packet interval is scaled down accordingly in run-time. The `rate_scale` value must be set so that the error of rounding the scaled CIR value to integer is less than 1%—that is, the following formula must be satisfied:

$$\text{scaled_CIR} - \text{round}(\text{scaled_CIR}) < 0,01 * \text{scaled_CIR}$$

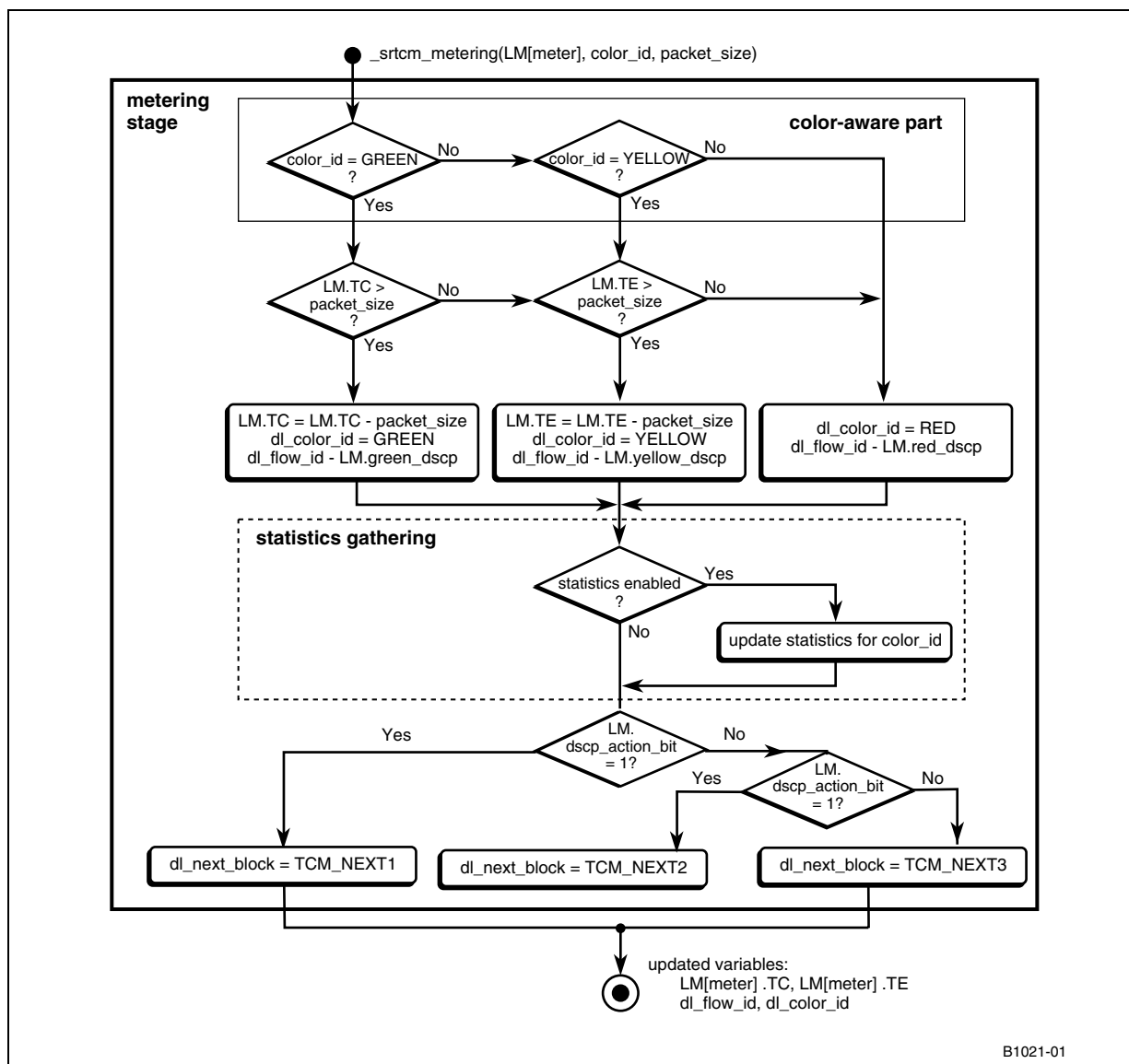
where `scaled_CIR` is equal to CIR in bytes per timestamp shifted left by `rate_scale`.

The MEv2 timestamp is 64 bits long, and it consists of two 32-bit long words. A question is whether both long words should be taken into account or is it enough to use the less significant 32 bits only. The lower 32 bits roll out every: $2^{32} * 16 * 600 \text{ MHz} \approx 114,5$ seconds. Packet arriving at intervals close or higher than the rollover time would be unfairly metered. Such a scenario applies to very slow flows, or to flows that arrive in bursts. To avoid unfairness in both cases, all 64 bits of a timestamp are used.

31.6.2.2 Color-aware Metering

As illustrated in [Figure 31-9](#), the color-aware mode reduces to a color-blind mode, when an incoming packet is marked green. Since the main share of code is similar in both modes, the design decision was to implement the color-aware extensions. To use the color-blind mode only, one can simply initialize `color_id` with green (value zero). For further optimization, the color-aware parts of the metering algorithm are provided as a conditional code which can be disabled by a compile-time option.

Figure 31-9. SRTC Metering Macro



31.6.2.3 Statistics Gathering

Statistics can be disabled by a compile-time option. If not compiled out, the SRTC block checks for each flow_id if statistics should be gathered. If true, two counters are increased on a packet arrival: byte counter and packet counter.

According to DS MIB [RFC3289], the statistics counters should be 64-bit long. To keep the basic SRTC table entry below 16 long words, the entry contains only less significant 32 bits of these counters. The upper 32 bits are kept in a separate memory location. There are two alternative methods to update the upper 32 bits of a counter.

- An XScale component updates the 64-bit counter in background using an SRAM atomic swap operation, as described in Intel® XScale™ Core Support of the *IXA Portability Framework Reference Manual*.
- A SRTCM meter microblock issues an SRAM atomic increment on the upper part whenever rollover occurs on a lower 32-bit counter.

The first method seems appealing, as it does not impact directly the fast-path. However, there are several deficiencies. Firstly, an XScale™ component has to perform atomic SRAM updates more frequently than SRTCM microblock, because it does not know when a rollover occurs. Thus effectively, the first approach results in more SRAM accesses. Secondly, there is a danger of data corruption. An XScale component could perform an SRAM atomic swap while a counter is cached in Local Memory.

For these reasons, the second method is used. It adds a slight overhead to the SRTCM microblock, because it needs to handle rollovers. The overhead is negligible when compared to the amount of processed packets (~ 6 million packets/sec). With one flow of 2.5 Gbit/s, the packet counter rolls over every 588 seconds, and the byte counter every 12 seconds.

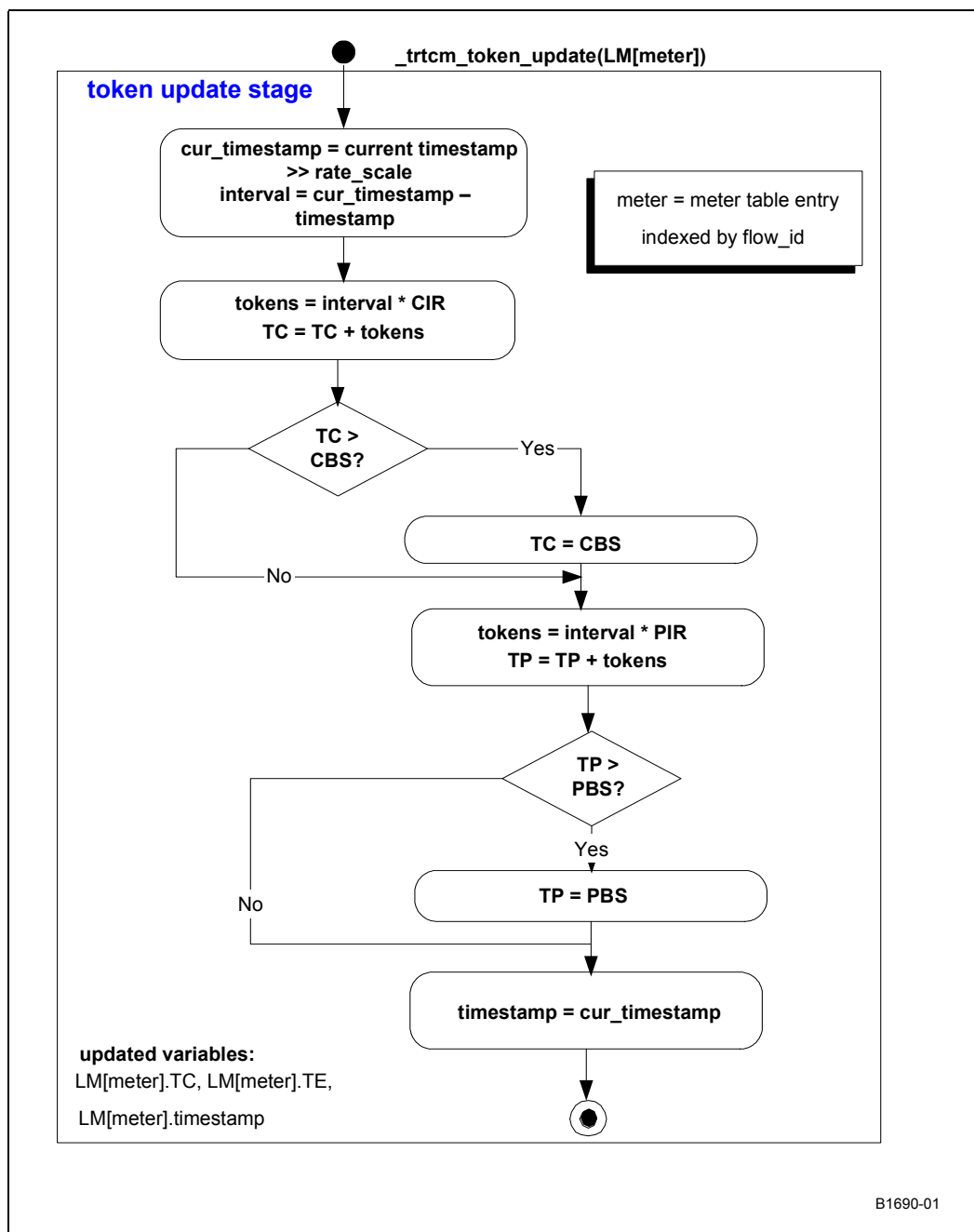
31.6.3 TRTCM Algorithm

Generally, the TRTCM implementation adheres to “A Two Rate Three Color Marker” (www.ietf.org/rfc/rfc2698.txt).

31.6.3.1 Per-packet Token Updates

Token counters are updated whenever a new packet arrives to a given flow. The reasons are the same as for SRTCM algorithm (see [Section 31.6.2, “SRTCM Algorithm” on page 553](#)). [Figure 31-10](#) illustrates the per-packet token update algorithm for TRTCM.

Figure 31-10. TRTCM Token Update Macro

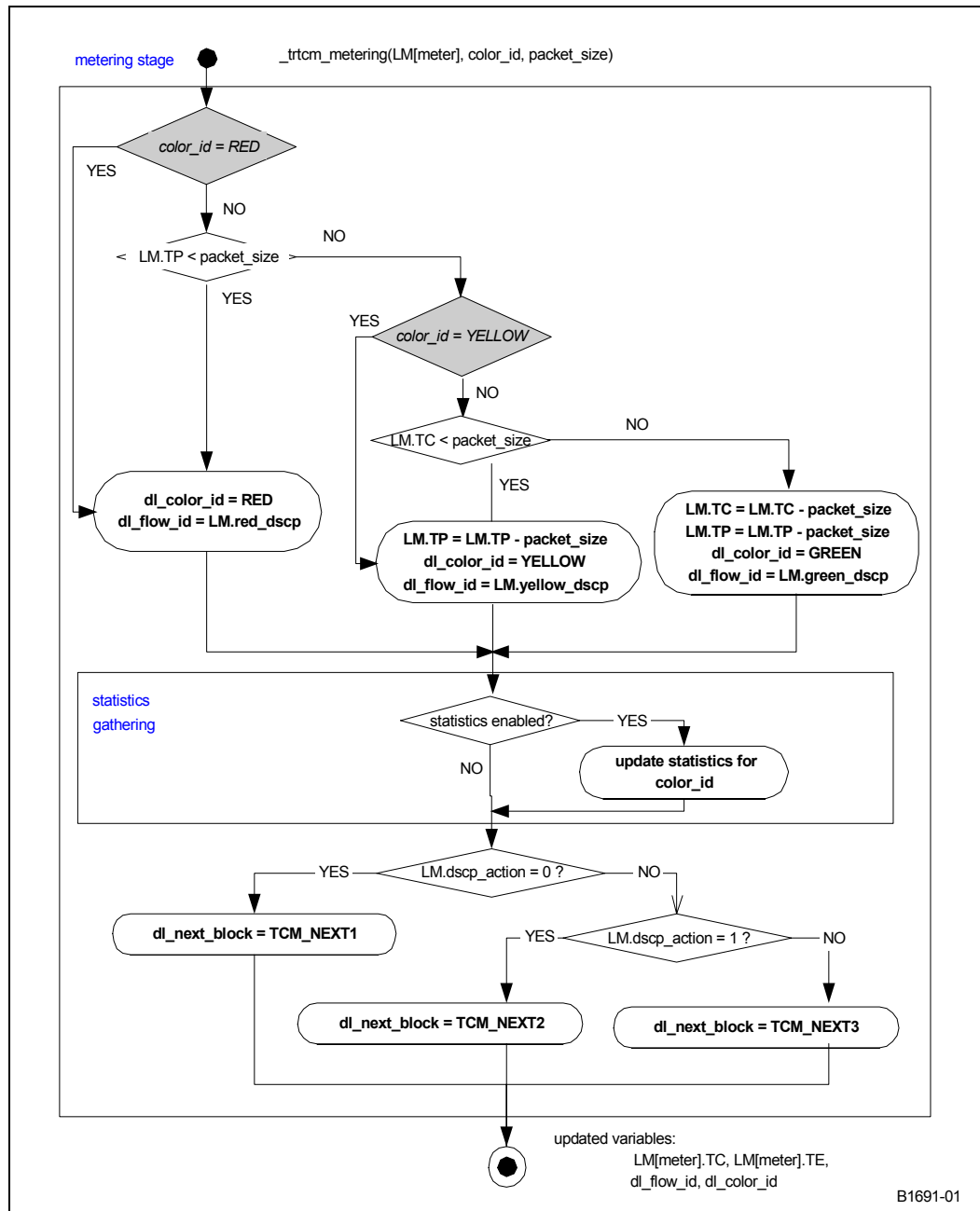


The rate scaling factor plays the same role as in SRTCM algorithm. It allows computing TC and TP updates for underflows. Refer to section [Section 31.6.2.1, “Per-packet Token Updates”](#) on [page 554](#) for detailed discussion of the parameter.

31.6.3.2 Color-aware Metering

Figure 31-11 shows color-aware version of TRTCM algorithm. The decision boxes specific for color-aware version are shown in gray. The color-aware version can be easily reduced to color-blind version when incoming packet is pre-colored green. Since the main share of code is similar in both modes, the design decision was to implement the color-aware extensions. Similarly to SRTCM algorithm, to use the color-blind mode only, one can simply initialize `color_id` with green (value zero). For further optimization, the color-aware parts of the metering algorithm are provided as a conditional code, which can be disabled by a compile-time option.

Figure 31-11. TRTC Metering Macro



31.6.3.3 Statistics Gathering

The TRTCM algorithm implements statistics gathering feature in the same way as SRTCM algorithm (see [Section 31.6.2.3, “Statistics Gathering” on page 556](#)).

31.7 Micro-code Budget

This section presents micro-code budget for three versions of the microblock:

- code version supporting SRTCM algorithm
- code version supporting TRTCM algorithm
- code version supporting both algorithms

31.7.1 Performance Analysis

Tables 31-6 and 31-7 shows the cycle count for SRTCM and TRTCM versions.

Table 31-6. Cycle Count Table for SRTCM Version (including unfilled defers)

Phase	Best case (all threads) ¹	Worst case (all threads)	Best Case (one thread) ²	Worst case (one thread) ³
Synchronization & CAM clear	6 5/8	6 5/8	6 (CTX>0)	11 (CTX = 0)
SRAM read	12 7/8	12	13 (CAM hit)	12 (CAM miss)
Token update	3 7/8	31	0 (CAM hit)	31 (CAM miss)
Packet metering	12 (all green packets)	16 (all yellow packets)	12 (green)	16 (yellow)
SRAM write	11 7/8	14 1/8	11 (no writeout)	15 (CTX = 7) ¹⁴ (writeout)
Total (color-blind, no stat.)	47 1/8	79 6/8	42	84
Color-aware overhead	4	4	4	4
Statistics overhead	7	18	7	18

1. best case for all threads corresponds to 1 miss and 7 hits; worst case - 8 misses
2. best case for one thread - CTX != 0 ; CAM hit ; green packet ; no write
3. worst case for one thread - CTX = 0 ; CAM miss ; yellow packet ; writeout

Table 31-7. Cycle Count Table for TRTCM Version (including unfilled defers)

Phase	Best case (all threads) ¹	Worst case (all threads)	Best Case (one thread) ²	Worst case (one thread) ³
Synchronization & CAM clear	6 5/8	6 5/8	6 (CTX != 0)	11 (CTX = 0)
SRAM read	12 7/8	12	13 (CAM hit)	12 (CAM miss)
Token update	6 3/8	51	0 (CAM hit)	51 (CAM miss)
Packet metering	11	19	11 (red)	19 (green)
SRAM write	11 7/8	14 1/8	11 (no write)	14 (writeout) ¹⁵ (CTX = 7)
Total (color-blind, no stat.)	48 6/8	102 6/8	41	107
Color-aware overhead	1	9	1	9
Statistics overhead	7	21	7	21

1. best case for all threads corresponds to 1 miss and 7 hits; worst case - 8 misses
2. best case for one thread - CTX != 0 ; CAM hit ; red packet ; no write
3. worst case for one thread - CTX = 0 ; CAM miss ; green packet ; writeout

Table 31-8. Cycle Count table for SRTCM and TRTCM version (including unfilled defers)

Phase	Best case (all threads) ¹	Worst case (all threads)	Best Case (one thread) ²	Worst case (one thread) ³
Synchronization & CAM clear	6 5/8	6 5/8	6 (CTX != 0)	11 (CTX = 0)
SRAM read	13 1/8	14	13 (CAM hit)	14 (CAM miss)
Token update	6 3/8	51	0 (CAM hit)	51 (CAM miss)
Packet metering	16	24	16 (red)	24 (green)
SRAM write	11 7/8	14 1/8	11 (no write)	14 (writeout) ¹⁵ (CTX = 7)
Total (color-blind, no stat.)	54	109 6/8	46	114
Color-aware overhead	0	4	0	4
Statistics overhead	7	23	7	23

1. best case for all threads corresponds to 1 miss and 7 hits; worst case - 8 misses
2. best case for one thread - TRTCM ; CTX != 0 ; CAM hit ; red packet ; no write
3. worst case for one thread - TRTCM ; CTX = 0 ; CAM miss ; green packet ; writeout

Table 31-9 does not include SRAM atomic increments to update 64-bit statistics. These operations are very rare (~ once per 1 million packets).

Table 31-9. I/O Latency Analysis Table (for all versions)

Operation	I/O type	Accesses	Transfer size (in 32-bit long words)
Load meter entry on CAM miss	SRAM read	1	14 or 8 (statistics compiled in or not)
Flush meter entry on exit	SRAM write	1	10 or 4 (statistics compiled in or not)

Table 31-10. Memory Access Summary Table

I/O type	Accesses
SRAM	2

31.7.2 Memory Footprint Analysis

Table 31-11. SRAM Footprint on Ingress IXP 2400

SRAM Data structures	Size (bytes)
Meter table (entry size is 16 long words)	64 kB (1024 flows x 16 long words)
Statistics table (entry size is 8 long words)	32 kB (1024 flows x 8 long words)
Total	96 kB

Table 31-12. SRAM Footprint on Egress IXP 2400

SRAM Data structures	Size (bytes)
Meter table (entry size is 16 long words)	16 kB (256 queues x 16 long words)
Statistics table (entry size is 8 long words)	8 kB (256 queues x 8 long words)
Total	24 kB

Table 31-13. Code Store Footprint

Code Store	Size (instruction) Color-blind mode	Size (instruction) Color-aware mode
SRTCM	113 (170 with statistics)	117 (174 with statistics)
TRTCM	135 (192 with statistics)	140 (197 with statistics)
SRTCM & TRTCM	172 (229 with statistics)	181 (238 with statistics)
Initialization	10	

32.1 Overview

The DCSP marker block writes a `flow_id` value to the DSCP field in an IP header.

The microblock handles both IPv4 and IPv6 packets. In case of IPv4 packet, the microblock updates also IP header checksum.

32.2 Microblock Interfaces

32.2.1 Input Microblock Variables

Tables 32-1 and 32-2 show input and out microblock variable.

Table 32-1. Variables Consumed by DSCP Marker

Variable	Size	Type ¹	Description
<code>dl_flow_id</code>	16 bits	D	A value (6 less significant bits) to be marked in a DS field of an IP header.
<code>dl_next_block</code>	8 bits	D	It should be equal to <code>BID_DSCP_MARKER</code> . Otherwise, this block executes an empty bypass path.
<code>dl_header_type</code>	4 bits	D	The type of packet stored at “offset” bytes into the data.
<code>in_ip_header</code>	20 bytes (IPv4)	X	Input IP header cached locally in a micro-engine

1. D = dispatch loop variable, X = xfer array

32.2.2 Output Microblock Variables

Table 32-2. Variables Modified by DSCP marker

Variable	Size	Type ¹	Description
<code>dl_next_block</code>	8 bits	D	Should be hard-coded in <code>system.h</code> as <code>DSCP_MARKER_NEXT1</code> .
<code>out_ip_header</code>	20 bytes (IPv4)	X	Output IP header with updated DS field, cached locally in a microengine

1. D = dispatch loop variable, X = xfer array

32.3 Flow Chart

32.3.1 Synchronization

This block does not implement a critical region, thus no synchronization is needed.

32.3.2 Marking Algorithm

The DSCP maker writes 6 less significant bits of a `dl_flow_id` variable into a DS field of an IP header. The DS field occupies 6 left-most bits of the Type of Service field (IPv4) or Traffic Class field (IPv6):

- Type of Service (IPv4) field occupies bits 8-15 of an IP header
- Traffic Class (IPv6) field occupies bits 4-11 of an IP header

In case of IPv4 packets, the marker updates also the IP header checksum to reflect the DS field modification. Since only one header field is affected, the marker uses an incremental checksum update algorithm, as described in equation 4 of [RFC 1624]:

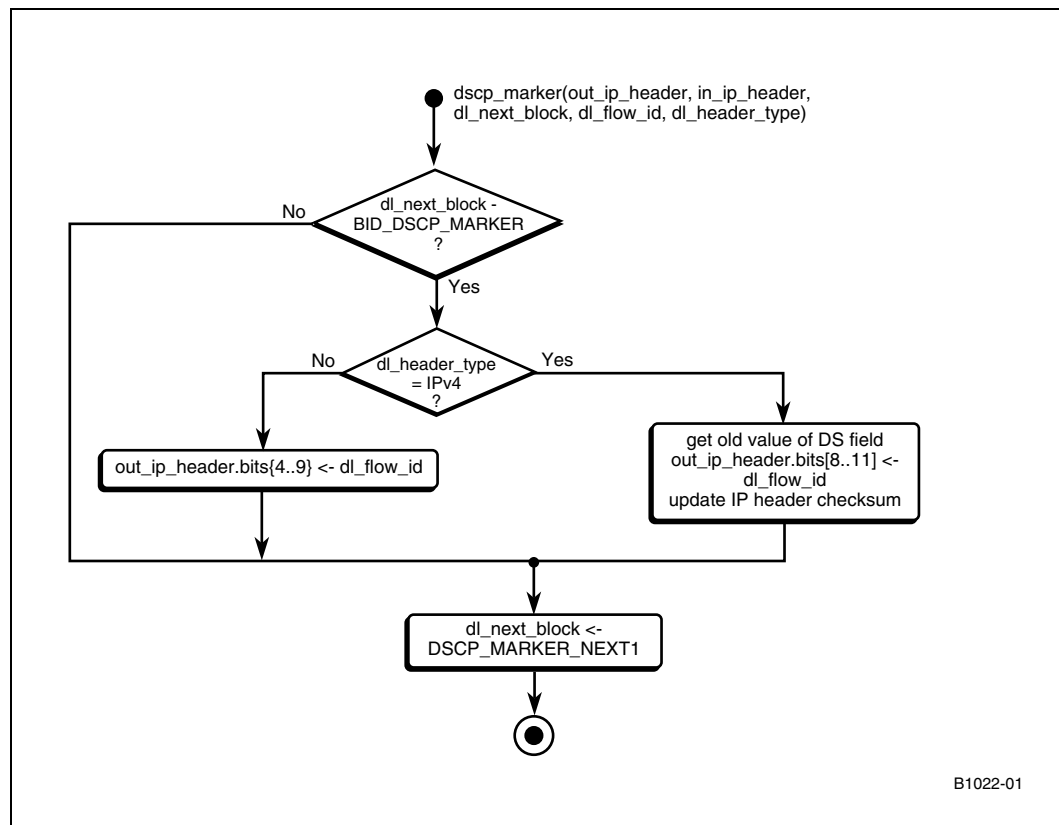
$$HC' = HC - \sim m - m'$$

where:

- `HC` is the old checksum in an IP header
- `HC'` is the new checksum in an IP header
- `m` is the old value of a 16-bit field that includes old DSCP bits
- `m'` is the new value of a 16-bit field that includes new DSCP bits

Bearing in mind differences between IPv4 and IPv6, the DSCP marker works as follows:

Figure 32-1. DSCP Marking Algorithm



The DSCP marker assumes that it receives an IP header in the xbuf array `in_ip_header`. The block writes updated fields to `out_ip_header` using the `xbuf_insert()` macro. The remaining fields of the IP header are copied into outgoing header using `xbuf_copy()`. If two xbuf arrays are the same, the latter macro resolves to a null code at compile time.

32.4 Micro-code Budget

32.4.1 Performance Analysis

Table 32-3. Cycle count table (including unfilled defers)¹

Operation	Worst Case
IPv4 total	20
IPv6 total	16

1. Assumption: input IP header cache is the same as output cache (i.e. `xbuf_copy()` expands to an empty macro)

32.4.2 Memory Footprint Analysis

Table 32-4. Code Store footprint

Code Store	Size (instruction)
Total	34

33.1 Overview

The nodes inside a DiffServ domain usually does not require multi-field classification functionality. They classify packets using 6-bit DSCP value set by the edge nodes—[RFC2474] describes coding of DSCP in IP packet header. In IPv4 packets, the value is encoded in 6 MSBs of TOS (Type of Service) field. In IPv6 packets, the value is encoded in 6 MSBs of IPv6 Traffic Class field.

This section describes a building block that supports DSCP classification for IPv4 and IPv6 packets. The block applies the same rules for both IP versions. The classification results correspond to QoS treatment to be applied to the packets. The microblock does not set any forwarding parameters.

As per [RFC3289] a classifier (set of rules) is bound to a particular interface, and filtering rules can vary between interfaces. The microblock implements this feature by extending the filtering rules with the interface number.

33.2 Microblock Design

This section describes the high level design for the DSCP classifier microblock.

33.2.1 Functionality

The microblock supports the following features:

- Implements DSCP classification on IPv6 and IPv4 packets. [RFC3289] uses Behavior Aggregate filtering term for this functionality.
- Supports a number of input ports, which is a compile time option (by default it is equal to 256).
- Does not support any default rule. It is assumed that the classification rules are complete. This means that classification results are defined for each <interface, DSCP value> pair. However, the core component supports default interface configuration applied to interfaces for which interface-specific rules have not been defined.
- Associates a rule with following information:
 - QoS information (`flow_id`, `class_id`, `color_id`).
 - Identifier of a next block in a processing path (`dl_next_block`)
 - The packet does not throw any exception packets.
 - A microblock implements 64-bit per-rule statistics. The statistics contain a number of packets and bytes matching a rule. This feature is implemented as a compile time option. Statistics gathering can also be switched off in run-time separately for each rule.

33.2.2 Assumptions and Dependencies

The following design assumptions are made for the microblock:

- Does not access a packet header stored in DRAM. It assumes that the header is cached locally inside a microengine—for example, in GPR or transfer registers. The header caching should be assured in the dispatch loop source microblock.
- Implements statistics feature as a compile time option. Statistics gathering can also be switched off in run-time separately for each rule.
- Implements statistics using SRAM atomic operations. The microblock updates both 32-bit parts of the 64-bit counters.
- Stores classification results and statistics counters in SRAM. They are kept in separate tables (memory regions). This makes it possible to locate them in two different memory channels to load-balance utilization of SRAM controllers and I/O busses.
- Provides five outputs:
 - METER output for packets subjected to traffic conditioning action (usually bound to TCM block),
 - MARKER output for packets requiring remarking on DiffServ domain boundary (usually bound to DSCP marker block),
 - IP4_FWD for IPv4 packets that do not require any further QoS processing on an ingress blade (usually bound to IPv4 forwarder),
 - IP6_FWD for IPv6 packets that do not require any further QoS processing on an ingress blade (usually bound to IPv4 forwarder),
 - INV_IP for non-IP packets (usually bound to IX_DROP).

33.2.3 Configuration Options

33.2.3.1 Build Switches

Table 33-1 identifies compile time build switches, which can be turned on or off as per the requirements of a specific application..

Table 33-1. Build Switches for DSCP Classifier Microblock

Symbol	Description
CLASSIFIER_DSCP_PACKET_COUNTER_FEATURE	Enables per rule statistics

33.2.3.2 Default Configuration

The microblock is released without any of the above switches defined.

33.2.4 Microblock Interfaces

33.2.4.1 Input Microblock Variables

Table 33-2 lists the variables input to the microblock.

Table 33-2. Input Variables Consumed

Variable	Size	Type ¹	Description
dl_input_port	16 bits	D	Logical number of an incoming interface, on which a packet was received.
dl_next_block	8 bits	D	Either hard-coded in a former block, or computed in run-time. It should be equal to BID_CLASSIFIER_DSCP. Otherwise, the classifier block executes an empty bypass path.
ip_header_in	2 bytes	X	An input xbuffer array with a cached IPv6 or IPv4 header. Only the first two bytes are used.

1. D = dispatch loop variable, X = xbuf array

33.2.4.2 Output Microblock Variables

Table 33-3 lists the output variable.

Table 33-3. Output Variables Modified

Variable	Size	Type ¹	Description
dl_next_block	8 bits	D	Identifier of a microblock that should continue with packet processing. The binding should be hard-coded in system.h as: <ul style="list-style-type: none"> CLASSIFIER_DSCP_METER CLASSIFIER_DSCP_MARK CLASSIFIER_DSCP_IP4_FWD CLASSIFIER_DSCP_IP6_FWD CLASSIFIER_DSCP_INV_IP The output block is configurable on a per-rule basis.
dl_flow_id	32 bits	D	Identifier of a traffic flow to which the packet belongs. Subsequent blocks can use this value to select a policy instance.
dl_class_id	16 bits	D	Identifier of a QoS aggregate that share the same ordering constraint—that is, should be placed in the same queue. It is a relative queue number within an output port.
dl_color_id	2 bits	D	Packet drop precedence level: green, yellow or red color.

1. D = dispatch loop variable

33.2.4.3 Imported Variables

Table 33-4 lists the variables that should be patched by a core component.

Table 33-4. Variables Imported by this Block

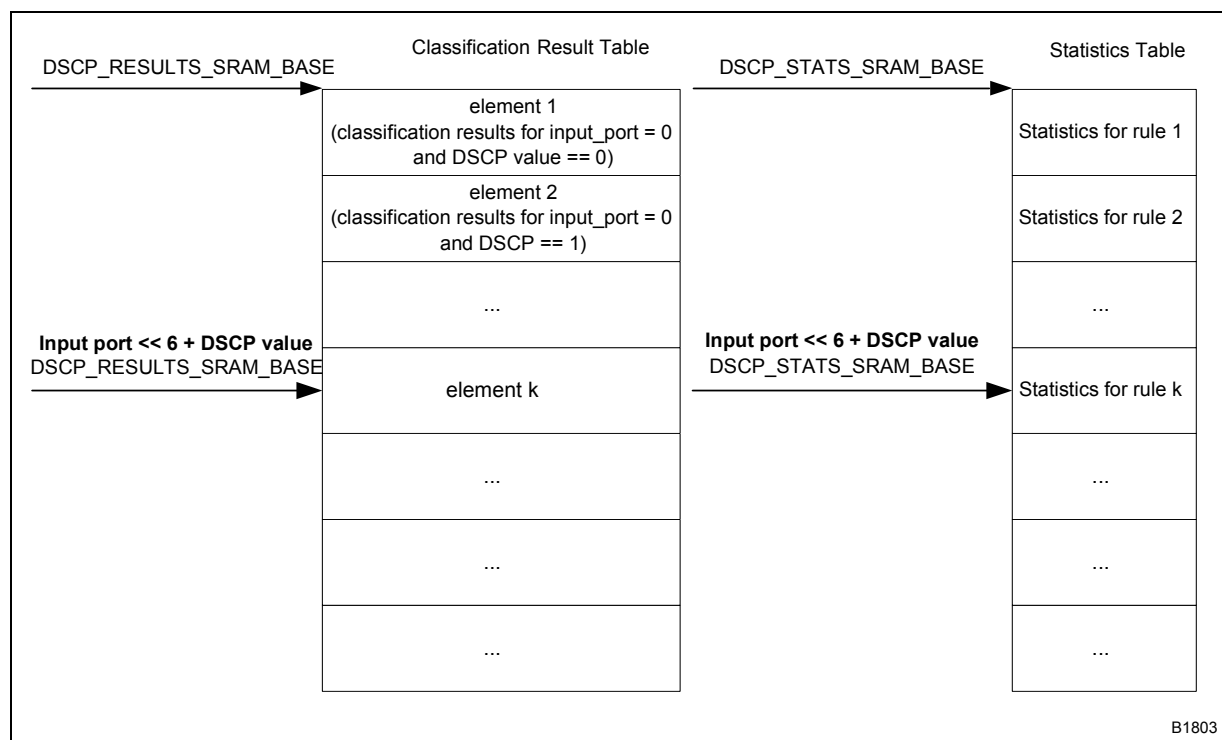
Variable	Default	Description
CLASSIFIER_DSCP__CONFIG_SRAM_BASE	-	Base address of the hash table maintained in DRAM or SRAM (depending on compile-time option)
CLASSIFIER_DSCP_STATS_SRAM_BASE	-	Base address of the statistics table maintained in SRAM

33.2.5 Data Structures

Figure 33-1 illustrates the SRAM table which stores all the DSCP classifier classification results. The table is indexed with <input port, DSCP> pair. A single entry contains a complete classification result. This ensures that for each packet only one lookup in the table is required.

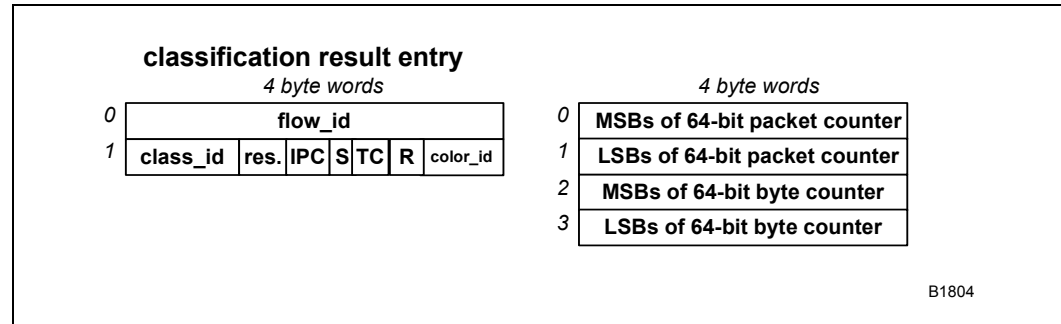
Statistics counters are located in a separate table. The table is indexed with the same index.

Figure 33-1. DSCP Rule Table Organization



A single element in the classification result table occupies two 32-bit long words. Statistic entries in the statistics table consist of two 64-bit counters. The first counter stores a number of packets matching the rule. The second counter keeps a number of bytes matching the rule. [Figure 33-2](#) illustrates both entries.

Figure 33-2. Classification Result and Statistics Entry Layout



The microblock only reads configuration entries, while the core component updates them.

Table 33-5. Classification Result Entry Definition

Field name	Bits	Size	Description
flow_id	31..0	32 bits	Identifier of a traffic flow to which the packet belongs. Subsequent blocks can use this value to select a policy instance.
class_id	31..16	16 bits	A relative queue number within an output port.
reserved	15..6	10 bits	Not used now
Input Port Configured flag (IPC)	5..5	1 bit	Input Port Configured flag indicating whether the input port corresponding to the entry has its own configuration. When the flag is asserted, the microblock sets dl_next_block to CLASSIFIER_DSCP_IP4_DEFAULT_OUTPUT for IPv4 packets or CLASSIFIER_DSCP_IP6_DEFAULT_OUTPUT for IPv6 packets.
Statistics flag (S)	4..4	1 bit	Indicates whether the microblock should gather statistics for the rule. This parameter is used only when the microblock is compiled with statistics handling.
TC flag	3..3	1 bit	Traffic Conditioning flag indicating whether the packets must be metered. The flag is meaningful only if the Input Port Configured flag is set. When the flag is asserted, the microblock sets the next_block_id variable to CLASSIFIER_DSCP_METER_OUTPUT.
remark flag	2..2	1 bit	Indicates whether DSCP value in the packets must be remarked. The flag is only significant when the Valid flag is set and TC flag is not set. When the flag is asserted, the microblock sets the next_block_id variable to CLASSIFIER_DSCP_MARKER_OUTPUT.
color_id	1..0	2 bits	Packet drop precedence level: green, yellow or red color.

Statistic entries in the statistics table consist of two 64-bit counters. The first counter stores a number of packets matching the rule. The second counter keeps a number of bytes matching the rule.

33.2.6 Flow Chart

33.2.6.1 Synchronization

The classifier microblock does not operate on shared data structures. Thus, there is no need for inter-thread synchronization. The microblock is organized as a functional pipe-stage. All threads in a microengine execute the same algorithm, but they process different packets.

33.2.6.2 Classification Algorithm

As illustrated in Figure 7 3, an algorithm first checks if the `dl_next_block` variable points to the classifier. If not, a bypass path is executed. Otherwise, the classifier verifies if the processed packet is IPv6 or IPv4. If the packet is neither IPv4 nor IPv6, the microblock sets `dl_next_block` variable to `CLASSIFIER_DSCP_INV_IP_OUTPUT` and exits without changing the QoS related variables. Otherwise, the microblock retrieves DSCP value from the packet.

Next the microblock calculates index in the configuration table using DSCP value and `dl_input_port` variable. Knowing the entry index, the microblock issues a request to read the entry from SRAM and swaps out waiting for the I/O operation.

When the entry is in transfer registers, the microblock copies QoS parameters from the entry to dispatch loop variables. After that the microblock sets the `dl_next_block` variable.

First it checks if the Input Port Configured flag is set. If the flag is not set, the microblock sets next block to `CLASSIFIER_DSCP_IP4_DEFAULT_OUTPUT` or `CLASSIFIER_DSCP_IP6_DEFAULT_OUTPUT` depending on the IP version. If the port configured flag is asserted, the microblock checks TC flag. If the TC flag is set, the algorithm sets the variable to `CLASSIFIER_DSCP_METER_OUTPUT`. If the remark flag is set, the algorithm sets the variable to `CLASSIFIER_DSCP_MARKER_OUTPUT`. Otherwise, for IPv4 packets the variable is set to `CLASSIFIER_DSCP_IP4_FWD_OUTPUT`, and for IPv6 packets the variable is set to `CLASSIFIER_DSCP_IP6_FWD_OUTPUT`.

Figure 33-3. DSCP Classifier Algorithm (Page 1 of 3)

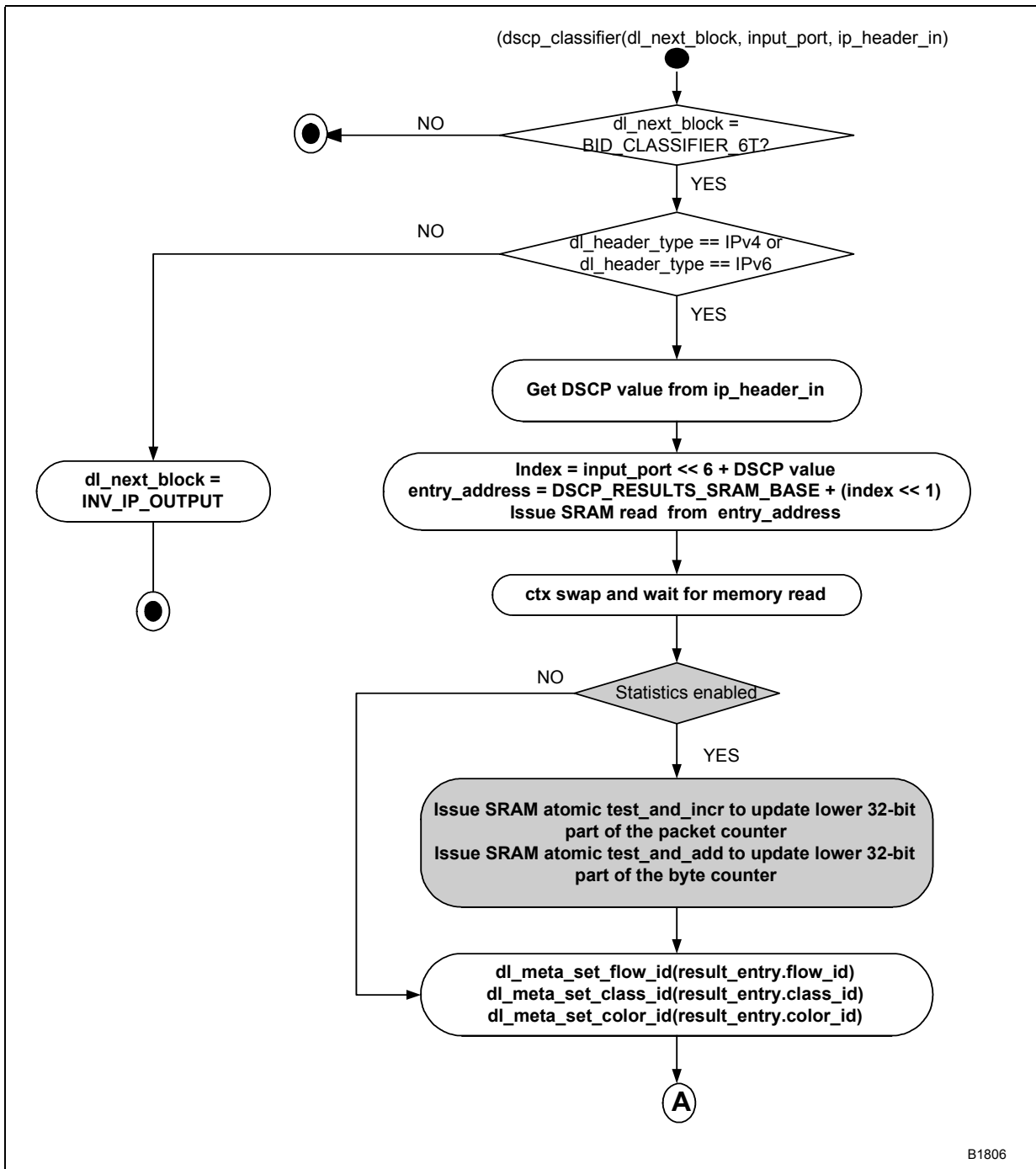


Figure 33-4. DSCP Classifier Algorithm (Page 2 of 3)

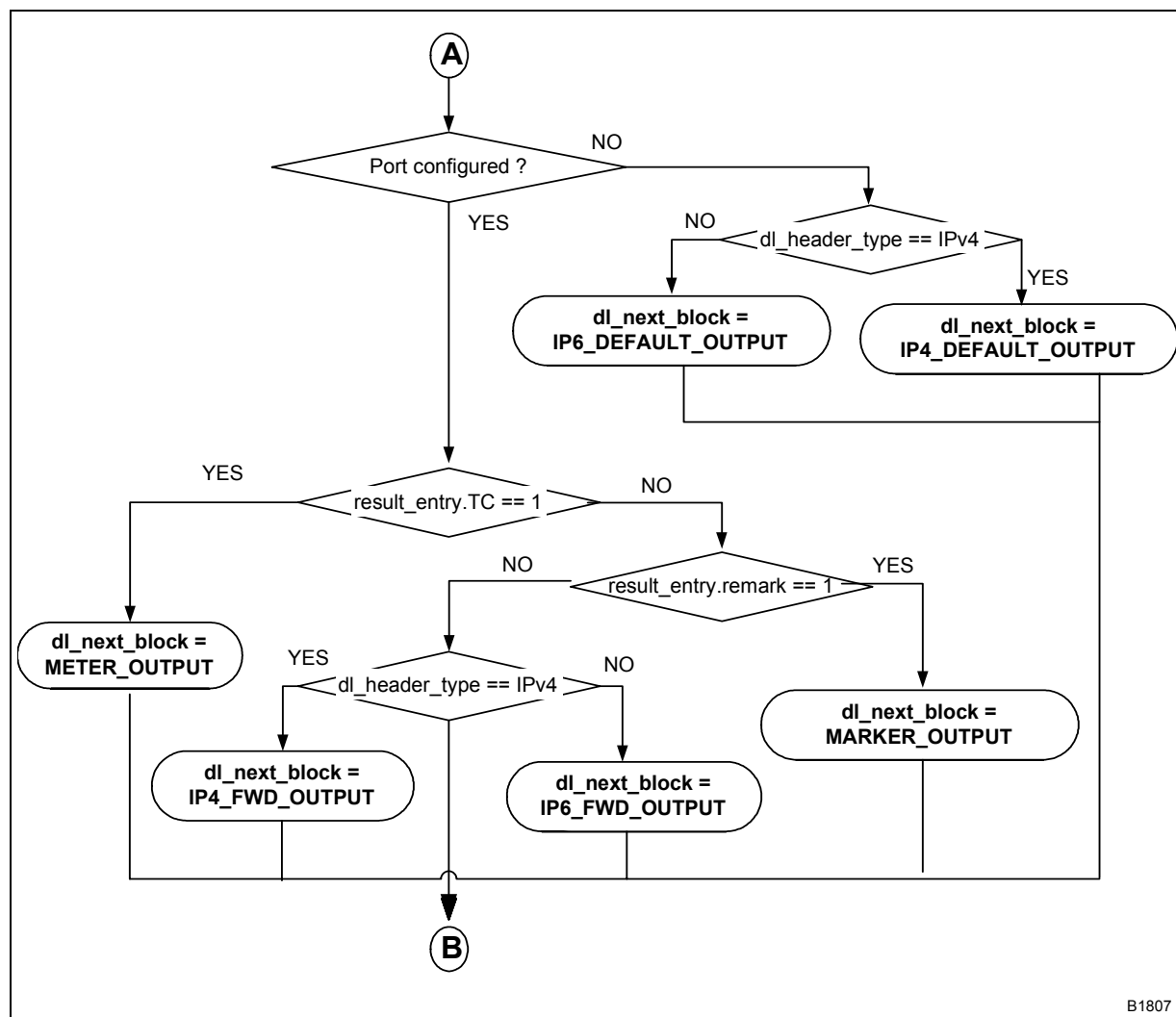
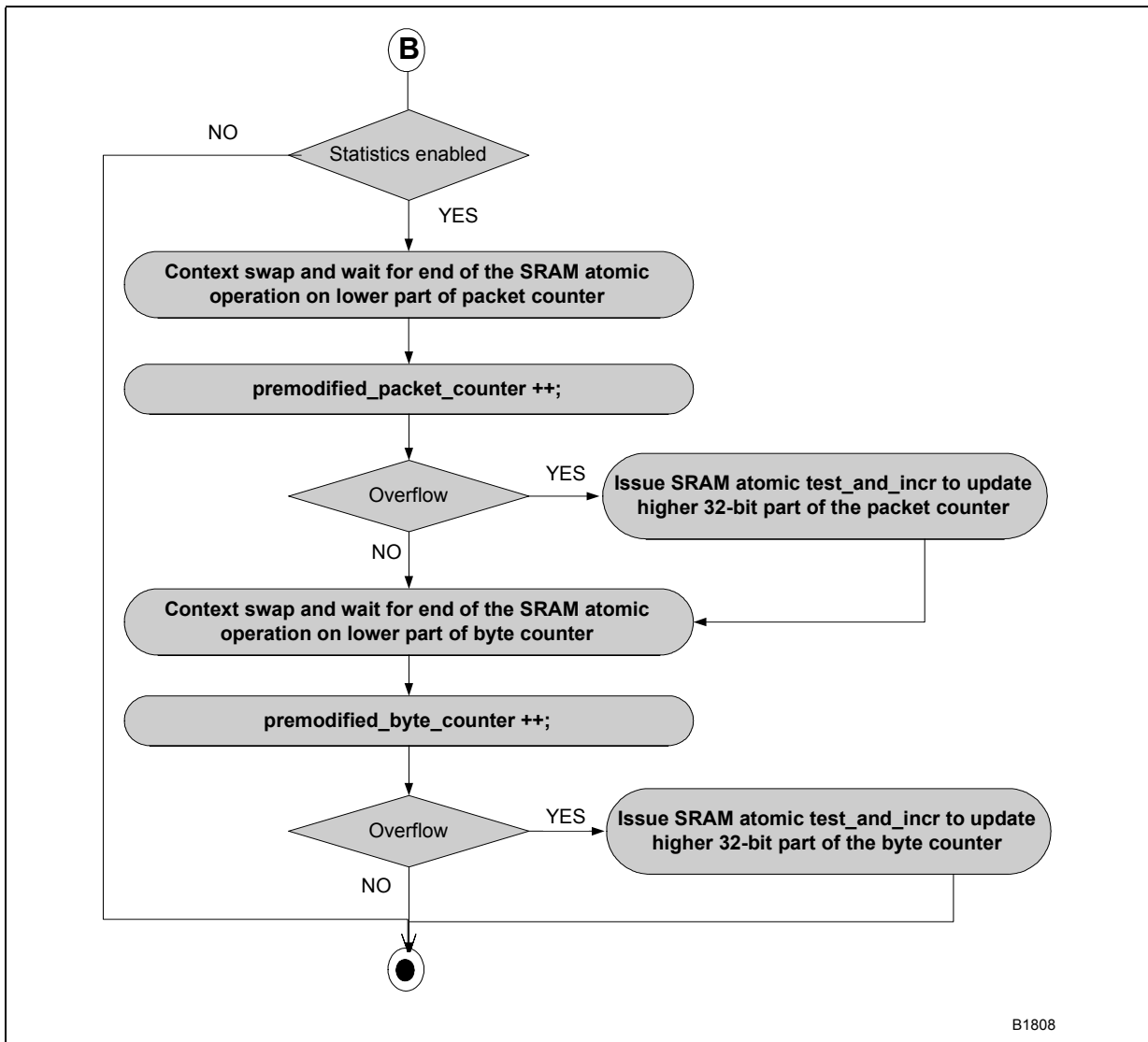


Figure 33-5. DSCP Classifier Algorithm (Page 3 of 3)



33.2.6.3 Statitics Gathering

The part of algorithm responsible for gathering statistics is shown in Figure 33-5 in gray. The code is present only if the microblock is compiled with statistics handling feature. The microblock does not use the the XScale-based statistics gathering scheme described in *Intel® Internet Exchange Architecture Portability Framework Developer's Manual*. This scheme assumes that the 64-bit counter is kept by the core component while a microblock uses 32-bit auxiliary variable.

Unfortunately the scheme does not scale well for large number of counters, because the core component issues too many unnecessary SRAM atomic operations. Another method for implementing 64-bit counters is presented in SRTCM design (see Section 31.6.2, “SRTCM Algorithm” on page 553). However this solution requires critical section with thread sequencing.

Therefore the DSCP classifier microblock updates the whole 64-bit counter. It updates the lower 32-bit counter part using SRAM atomic operations that return the premodified value. This makes it possible to detect overflow of the lower part. If such overflow occurs, the microblock issues atomic increment operation for the higher counter part.

This scheme ensures that the SRAM operations are issued only when update of counters is needed.

Note: The microblock should request atomic operation on the lower part of the counters just after it finds the configuration entry. This method shortens the time of waiting for the operation to complete and minimizes the probability of ME going to the idle state.

33.2.7 Micro-code Budget

33.2.7.1 Performance Analysis

Tables 33-6 and 33-7 reflect the data in current version of code prototyping.

Table 33-6. Cycle Count Table (including unfilled defers)

Phase	Best case ¹	Worst case
IPV4 total	25	30 ²
IPV6 total	28	32 ³
Statistics overhead	24	24

1. best case - port not configured
2. worst case for IPv4 packets - port configured, TC flag = 0, remark flag = 0
3. worst case for IPv6 packets - port configured, TC flag = 0, remark flag = 1

Table 33-7. I/O Latency Analysis Table

Operation	I/O type	Accesses	Transfer size (in 32-bit long words)
Result entry read from SRAM	DRAM/SRAM read	1	2
Byte counter update	SRAM atomic test_and_add	0 or 1 ¹	1
Packet counter update	SRAM atomic test_and_incr	0 or 1 ¹	1

1. depends on compile-time option that determines whether the microblock supports statistics

Table 33-7 does not take into account the SRAM atomic increment operation on the most significant 32-bit part of the counters. The most significant part of the byte counter is updated at most every 466034 packets and the most significant part of the packet counter is updated every 4294967296 packets.

33.2.7.2 Memory Footprint Analysis

Tables 33-8 and 33-9 show the SRAM footprint and code store footprint.

Table 33-8. SRAM Footprint

DRAM/SRAM Data structures	Size (bytes)
SRAM Classification Result table (entry size is 2 long words)	131 072 B (256 ports ¹ 64 DSCP values = 16384 entries, each entry has 8 bytes)
SRAM statistics table (entry size is 4 long words)	0* or 262 144 B (256 ports ¹ 64 DSCP values = 16384 entries, each entry has 16 bytes)
Total	131 072 or 393 216B

1. depends on compile-time option that determines whether the microblock supports statistics

Table 33-9. Code Store Footprint

Code Store	Size (instruction)
Total	36 (56 with statistics)

Weighted Random Early Detection (WRED) Microblock

34

34.1 Overview

The Random Early Detection (RED) algorithm is a recommended approach to active queue congestion avoidance in “IP routers Recommendations on Queue Management and Congestion Avoidance in the Internet” (www.ietf.org/rfc/rfc2309.txt). RED drops packets with a probability increasing along with the persistent average queue length. In DiffServ networks, the Assured Forwarding PHB, “Assured Forwarding PHB Group” (www.ietf.org/rfc/rfc2597.txt), requires a RED-like algorithm for minimizing the long-term congestion on AF queues.

Weighted RED (WRED) refers to several RED instances that operate on the same queue. Considering Assured Forwarding PHB in particular, there are three RED instances per queue. A RED instance is configured for each packet drop precedence level—that is, color: green, yellow or red. The Default Forwarding PHB can also employ WRED to best-effort traffic. Only one WRED instance is used in this case because DF PHB does not distinguish packet colors. There is no point in applying WRED to Expedited Forwarding PHB. By definition, this DiffServ class should experience low delay—that is, non-congested, empty queues—while WRED works on congested queues.

34.1.1 RED 93

The original RED algorithm was first described in “Random Early Detection Gateways for Congestion Avoidance” (<http://citeseer.nj.nec.com/floyd93random.html>). It was intended to minimize the incipient congestion that occurs due to TCP window mismatch between senders and receivers. A RED-compliant node detects congestion by computing the average queue size. When the average queue size exceeds a preset threshold, the node starts randomly dropping packets. The drop probability is a function of the average queue size. As a result of packet drops, the TCP sender reduces its window and invokes fast recovery. [In fact, Random Early Detection Gateways for Congestion Avoidance (<http://citeseer.nj.nec.com/floyd93random.html>) suggests that packets are marked, not dropped. Packet dropping is the most extreme case of “marking”. However, DiffServ networks exploit only the dropping feature, and so marking is not considered.]

The RED algorithm comprises two independent stages:

- Average queue length update
- Packet dropping probability calculation

Random Early Detection Gateways for Congestion Avoidance (<http://citeseer.nj.nec.com/floyd93random.html>) combines both stages in a single procedure invoked upon a packet arrival.

34.1.1.1 Average queue length

The averaging function determines how a node responds to temporary bursts. According to Random Early Detection Gateways for Congestion Avoidance (<http://citeseer.nj.nec.com/floyd93random.html>), a node calculates the average queue length using a low-pass filter based on Exponential Weighted Moving Average (EWMA). The filter absorbs transient burst, while keeps the long-term average low in case of non-congested network. The following equation is used:

$$\text{avg_len}(t) = \text{avg_len}(t - 1) + \text{weight} * (\text{cur_len} - \text{avg_len}(t - 1))$$

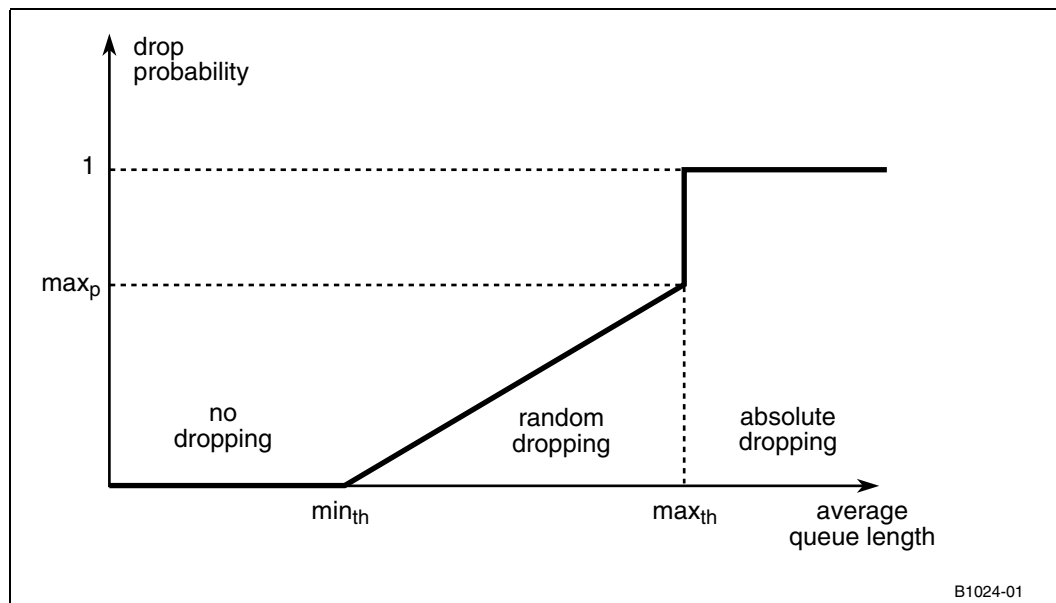
where	weight	represents the filter gain
	avg_len(t)	represents the new average queue length
	avg_len(t-1)	represents the previous queue length, on former packet arrival
	cur_len	represents the current, instantaneous queue length just sampled

Random Early Detection Gateways for Congestion Avoidance (<http://citeseer.nj.nec.com/floyd93random.html>) assumes queue sampling upon each packet arrival. This requires a special handling for the case when an instantaneous queue length is zero, and knowledge of the average queue servicing rate. Refer to subsequent design chapters for details.

34.1.1.2 Packet Drop Probability

The second stage determines how frequently a node drops packets, given the current queue congestion level—that is, average queue length. Upon each packet arrival, the average queue is compared to two thresholds: min_{th} and max_{th} . When the average is below min_{th} , then a packet is forwarded unconditionally, and when it exceeds max_{th} , a packet is always dropped. When avg_len varies from min_{th} to max_{th} , the packet drop probability increases linearly from 0 to max_p , as illustrated below. [In fact, in Random Early Detection Gateways for Congestion Avoidance (<http://citeseer.nj.nec.com/floyd93random.html>), the dropping probability also increases slowly with the count of non-dropped packets. This is to achieve uniform distribution of packet drop intervals.]

Figure 34-1. RED Dropping Function



34.1.2 RED99

The Random Early Detection Gateways for Congestion Avoidance (<http://citeseer.nj.nec.com/floyd93random.html>) paper presents an effective version of the algorithm that is easy to implement in networking devices. In brief, the “effective” algorithm uses pre-computed constants and an approximated, tabularized dropping function. Nevertheless, some research works claim that the original algorithm called “effective” is not that. Especially, this applies to performance concerns because of per-packet average queue calculation on high-speed links.

The most recent RED99 or “RED in a Different Light” (<http://citeseer.nj.nec.com/jacobson99red.html>) reference implementation requires very simple operations in the forwarding path—just before an arriving packet is to be queued.

```
drop_counter--
if (drop_counter == 0) {
    drop(packet)
    drop_counter = drop_counter_limit
}
```

The `drop_counter_limit` is setup in a queue sampling routine. Unlike in the original “Random Early Detection Gateways for Congestion Avoidance” (<http://citeseer.nj.nec.com/floyd93random.html>), this routine runs in background and not in the forwarding path. The queue sampling happens infrequently compared to packet forwarding—typically 10-100 times per second. This makes computations independent of link speed and scalable to high bandwidth routers.

Additionally, the “RED in a Different Light” (<http://citeseer.nj.nec.com/floyd93random.html>) algorithm attempts to evaluate a persistent queue length rather than the mathematical average. The subtle difference refers to handling an empty queue condition. Basically, the new algorithm follows RED93 rules when the queue is building up—refer to Random Early Detection Gateways for Congestion Avoidance (<http://citeseer.nj.nec.com/floyd93random.html>). However, if a queue drains, the persistent length is set to the instantaneous value:

```
if (cur_len < avg_len)
    avg_len = cur_len                // instantaneous
else
    avg_len = weight * (cur_len - avg_len) // EWMA filter applied
```

34.2 Functionality

- Three RED instances are configurable per queue, respectively for three packet colors.
- RED implementation is compatible with Random Early Detection Gateways for Congestion Avoidance (<http://citeseer.nj.nec.com/floyd93random.html>) specification. The microblock updates the average queue length on every packet arrival.
- The microblock gathers drop statistics for each queue and color: 64-bit long packet and byte counters.
- The microblock does not generate exception packets.

34.3 Assumptions and Dependencies

- The EWMA weight is limited to negative powers of 2, in range 2^{-0} to 2^{-15} . In fact, RED93 recommends setting weight to 2^{-9} (~0.002) for queues serviced with 1.5 Mb/s, while RED99 suggests about 2^{-5} or 2^{-4} . In either case, the weight value can be approximated as negative power of 2 to facilitate computations.
- The queue service rate, needed to handle a queue empty condition in RED93, is rounded to the nearest positive power of 2. This attribute is not configurable by an end user—that is, not visible in MIB. However, it should match a DRR bandwidth percentage configured for a queue.
- The RED93 version assumes knowledge of the instantaneous queue length and the last idle timestamp. Queue Manager should flush both values to a queue descriptor.
- The microblock uses `output_port` and `flow_id` to select a WRED instance. Both variables cannot be simultaneously zero; otherwise, WRED may fail on CAM lookup.
- The statistics can be disabled by a compile-time option (all statistics) or dynamically in run-time (selected queues).
- The implementation shall not preclude a solution, where a core component computes the average queue length in background (similar to RED99).

34.3.1 Configuration Options

34.3.1.1 Build Switches

Table 34-1 identifies compile time build switches, which can be turned on or off as per the requirements of a specific application.

Table 34-1. Build Switches for WRED Microblock

Symbol	Description
WRED_PACKET_COUNTER_FEATURE	Enables statistics gathering
WRED_QUEUE_SAMPLING_IN_CC	Enables queue sampling in core component
WRED_INGRESS_MODE	Indicates whether the WRED is used in ingress or egress. The microblock uses this switch to decide which dispatch loop variable to use to calculate queue identifier (<code>dl_fabric_port</code> or <code>dl_output_port</code>).
WRED_DISABLE_FIN_SYNC	Disables final thread synchronization. The flag should be used when the next microblock in the pipeline starts with thread synchronization.
WRED_INIT_RANDOM	Seed to init random generator.
WRED_MAX_QUEUE	Determines the number of queues per port in the system. (<code>queue_id = (output_port_id << WRED_MAX_QUEUE) + class_id</code>)

34.3.1.2 Default Configuration

The build switches with which the microblock will be released are as follows:

- WRED_INIT_RANDOM = 0x78922634
- WRED_MAX_QUEUE = 4

34.4 Microblock Interfaces

34.4.1 Input Microblock Variables

Table 34-2. Input Variables Consumed by WRED

Input variable	Size in bits	Type ¹	Description
dl_next_block	8	D	Either hardcoded in previous block, or obtained during the classification stage. It should be equal to BID_WRED. Otherwise, WRED executes an empty bypass path.
dl_output_port or dl_fabric_port	16	D	Identifier of an output port where to send a packet. Depending on hardware location, the variable is dl_output_port for egress NP, dl_port_fabric for ingress NP. There is a preprocessor variable deciding which dispatch loop variable is used.
dl_class_id	16	D	A relative queue number within an output port. A queue descriptor is uniquely identified by a combination of output_port and class_id.
dl_color_id	2	D	A packet color (0 = green, 1 = yellow, 2 = red). Only two least significant bits of dl_color_id are important. Other bits are masked out. If dl_color_id equals 3, red color is applied.
dl_packet_size	16	D	Number of bytes in the currently processed packet, used for statistics.

1. D = dispatch loop variable

34.4.2 Output Microblock Variables

Table 34-3. Output Variables Modified by WRED

Output variable	Size	Type ¹	Description
dl_next_block	8 bits	D	Should be hard-coded in system.h as WRED_NEXT1—packet passed, typically BID_QM—or WRED_NEXT2—packet dropped, typically IX_DROP.

1. D = dispatch loop variable

34.4.3 Imported Variables

Table 34-4. Variables Imported by this Block

Variable	Default	Description
WRED_TABLE_SRAM_BASE	—	Base address of the WRED table maintained in SRAM
QD_SRAM_BASE	—	Base address of queue descriptors table maintained in SRAM by Queue Manager
WRED_64BIT_STAT_SRAM_BASE	—	Base address of the statistics table storing most significant long words of 64-bit long counters.

34.5 Data Structures

The microblock uses two SRAM data structures:

- WRED table storing parameters of three RED instances for each queue. This table is maintained exclusively by a WRED block. The layout of entries in the table depends on the microblock version. Figures 34-2 and 34-3 show entry structure used by the microblock versions for low-speed and high-speed queues. Both illustrations present entry layouts with and without statistics. In the version for high-speed queues, the microblock reads only the part of the entry specific for the packet color, while the version for low-speed queues must read the whole entry.
- Queue descriptor table storing instantaneous queue length and last idle timestamp. This table is managed by Queue Manager ([Section 21.4, “Data Structures” on page 333](#)), but accessed by WRED. This table is used only by the low speed version of the WRED microblock.

Figure 34-2. WRED Data Structures in SRAM for Low-speed Queues

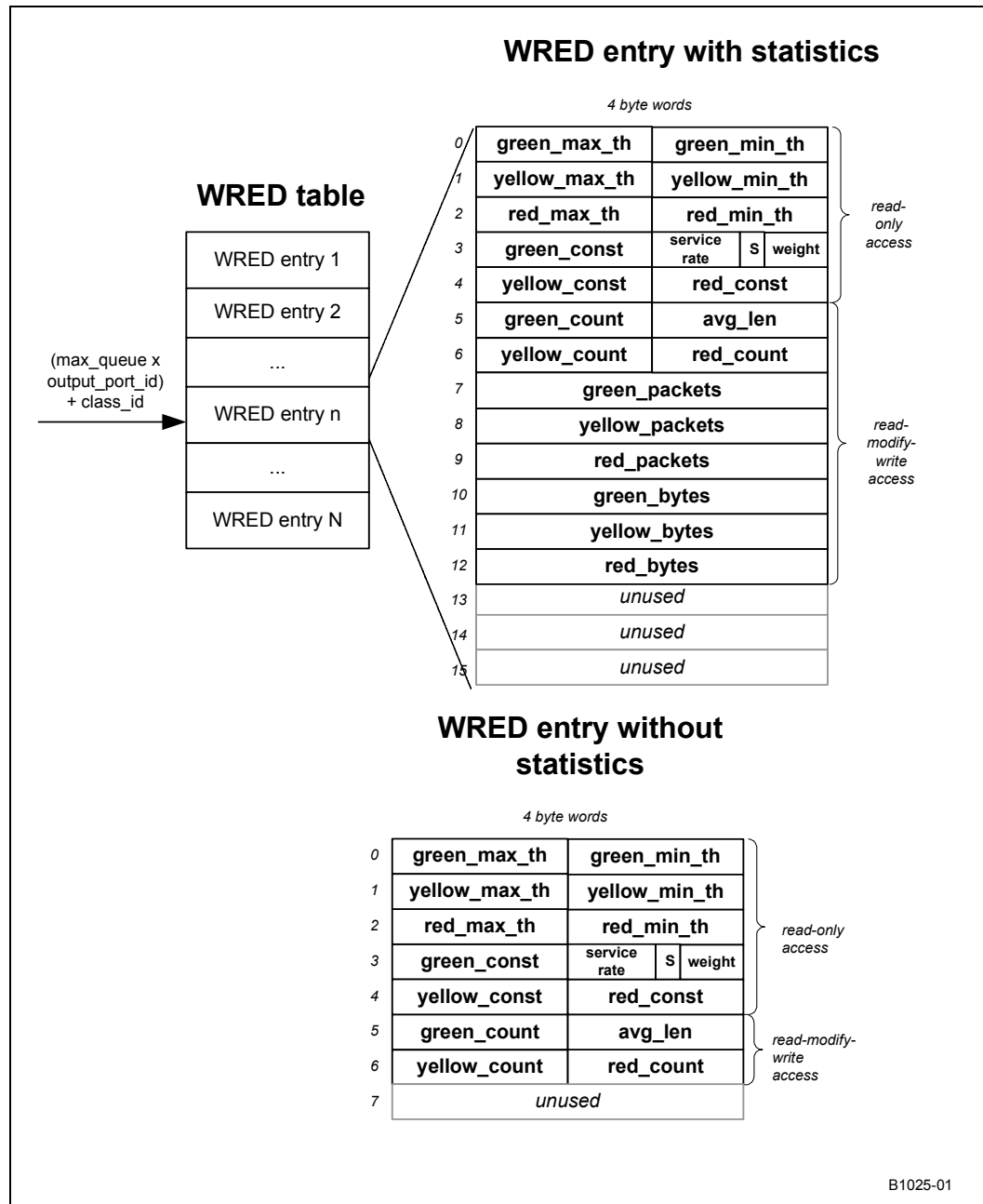
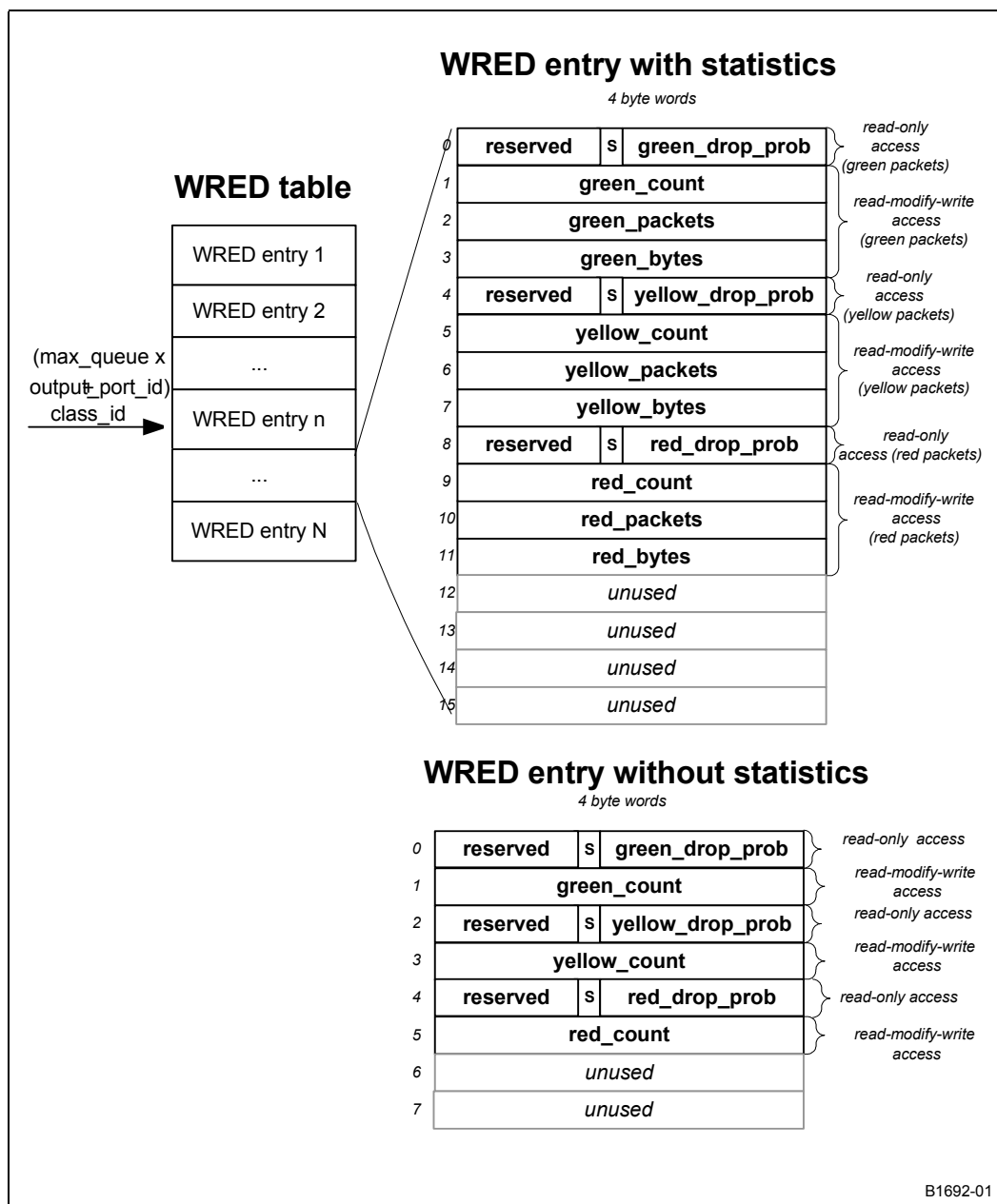
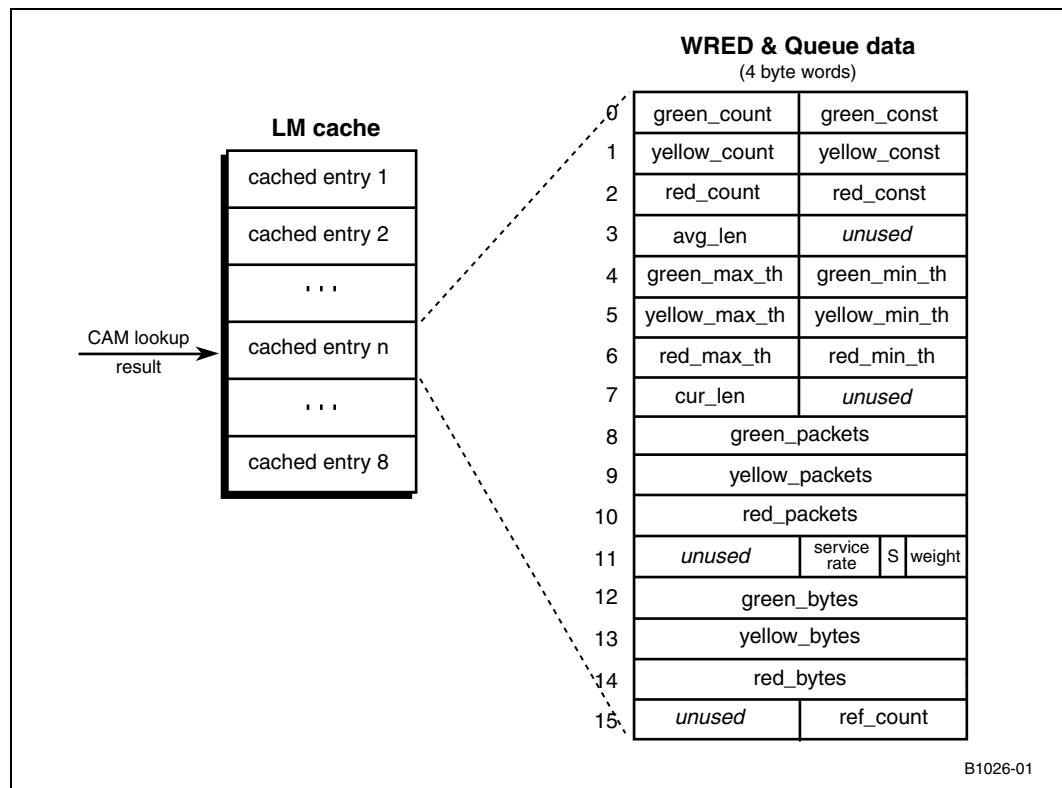


Figure 34-3. WRED Data Structures in SRAM for High-speed Queues



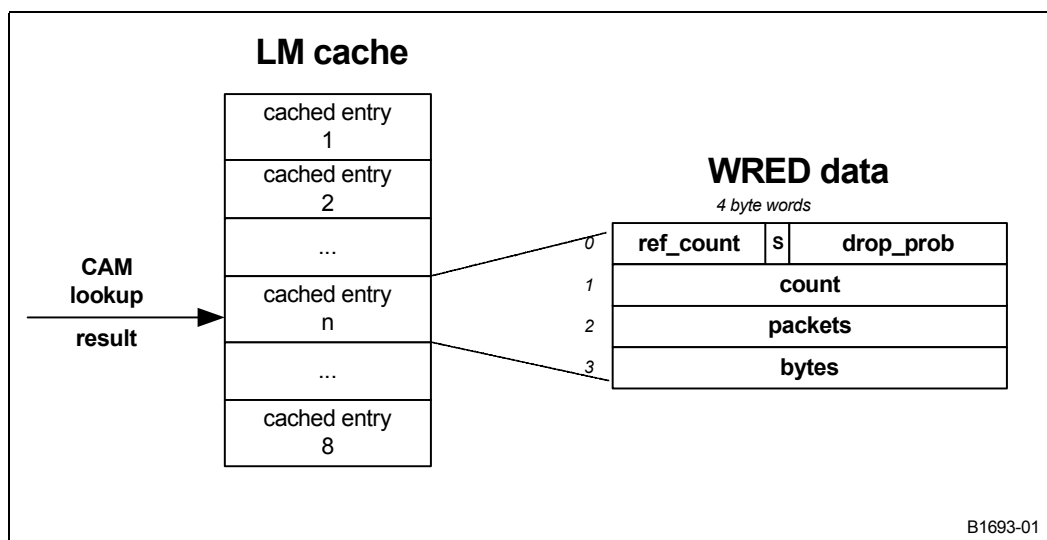
A thread executing RED93 algorithm needs to combine information from both tables. The combined data structures are cached in Local Memory, as illustrated in [Figure 34-4](#).

Figure 34-4. Local Memory Data Structures for Low-speed Queues



Note: The LM entry layout differs from SRAM entry layout. The rearrangement assures that color-specific entries can be selected by setting one LM index register to `color_id`. Moreover, WRED microblock does not store `last_timestamp` in Local Memory. Only a thread that misses on CAM lookup needs to know this value. On subsequent CAM hits—if any—threads calculate the average length according to the formula for a non-empty queue.

Figure 34-5. Local Memory Data Structures for High-speed Queues



Each thread of WRED for high-speed queues reads from SRAM only the part specific for color of the processed packet. Therefore an LM entry contains WRED parameters for a single color.

Both versions use addresses of the data read from SRAM as a CAM lookup key and evict the cached data basing of reference counter (`ref_count` field).

Table 34-5. WRED Table Entry Definition

Field name	Size	Description
avg_len (low-speed only)	16 bits	Average queue length, calculated using EWMA low-pass filter. It is encoded as a fixed-point value with 8 bits used for integer part.
green_count	16 bits (low-speed) or 32 bits (high-speed)	Number of green packets passed since last dropped green packet (initial value 0) - used if enabled "uniform distribution" option
yellow_count	16 bits (low-speed) or 32 bits (high-speed)	Number of yellow packets passed since last dropped yellow packet (initial value 0) - used if enabled "uniform distribution" option
red_count	16 bits (low-speed) or 32 bits (high-speed)	Number of red packets passed since last dropped red packet (initial value 0) - used if enabled "uniform distribution" option
green_packets	32 bits	Number of green packets dropped since last statistics reset
green_bytes	32 bits	Number of green bytes dropped since last statistics reset
yellow_packets	32 bits	Number of yellow packets dropped since last statistics reset
yellow_bytes	32 bits	Number of yellow bytes dropped since last statistics reset
red_packets	32 bits	Number of red packets dropped since last statistics reset
red_bytes	32 bits	Number of yellow packets dropped since last statistics reset

Table 34-5. WRED Table Entry Definition (Continued)

Field name	Size	Description
service_time (low-speed only)	8 bits	Average queue service rate, expressed in MEv2 timestamps (16*ME cycle) per packet. The service rate is approximated by a positive power of 2, and the effective rate equals to 2 service_time. The service_time value must be in range from 0 to 31.0xFF value means that the entry is empty.
statistics (S)	1 bit	If toggled on, statistics gathering is enabled. In case of WRED for high-speed queues, the flag is repeated in each per-color entry section.
weight (low-speed only)	5 bits	Weight of the EWMA filter, used to calculate the average queue length. It must be a value in range from 0 to 15. The actual EWMA filter gain is calculated as 2- weight (negative power of 2)
green_constyellow_const red_const(low-speed only)	16 bits (each)	Constant factor used to calculate packet drop probability, when the average queue length is between minimum and maximum thresholds. It is a decimal number equal to $\text{max_prob} \cdot 65536 / (\text{max_th} - \text{min_th})$.
green_min_thyellow_min_th hred_min_th(low-speed only)	16 bits (each)	Minimum threshold for average queue length in packets. It is a fixed-point value with 8 higher bits holding the integer part, and fraction part set to 00. All packets are enqueued, if the average length is below this threshold.
green_max_thyellow_max_th thred_max_th(low-speed only)	16 bits (each)	Maximum threshold for average queue length in packets. It is a fixed-point value with 8 higher bits holding the integer part, and fraction part set to 00. All packets are dropped, if the average length exceeds this threshold.
green_drop_prob yellow_drop_prob red_drop_prob(high-speed only)	16 bits(each)	Dropping probability calculated by the core component for each of the packet colors.

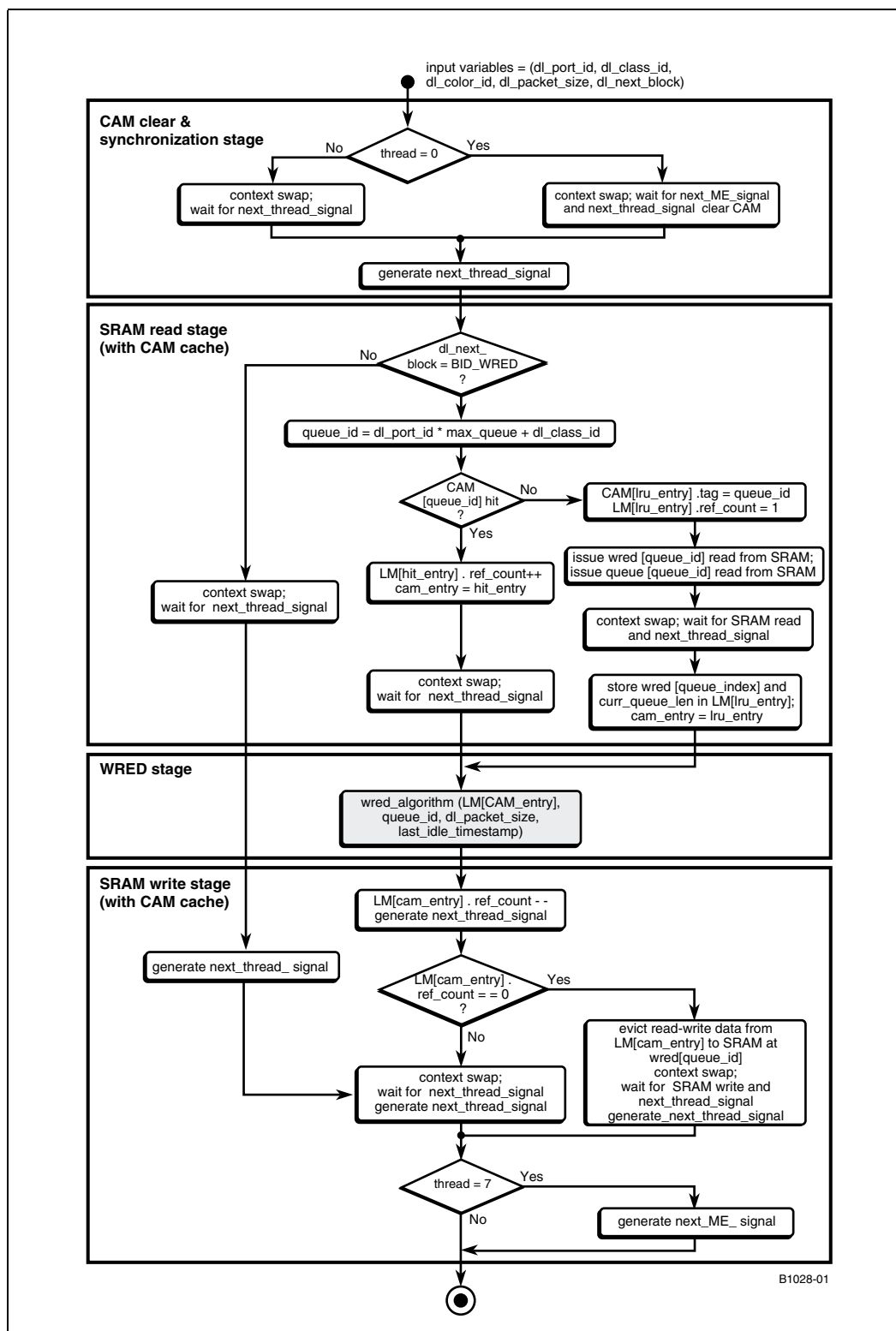
As per DS MIB Management Information Base for the Differentiated Services Architecture (www.ietf.org/rfc/rfc3289.txt), the statistics counters are 64-bit long and they refer to dropped packet/bytes. The WRED table stores only less significant 32 bits. The most significant bits are maintained in a separate table (base address of WRED_64BIT_STAT_SRAM). The 64-bit statistics counters are handled in the same way, as described in [Section 31.6.2, “SRTCM Algorithm” on page 553](#).

34.6 Flow Chart

34.6.1 Synchronization

All threads execute the WRED algorithm in parallel. A folding technique, combined with CAM lookup, implements a critical section.

Figure 34-6. WRED Inter-thread Synchronization



The final context swap handshake (grayed code) assures that the critical section is kept as short as possible. This code can be compiled-out, if the threads are to be synchronized on entry to the subsequent block anyway.

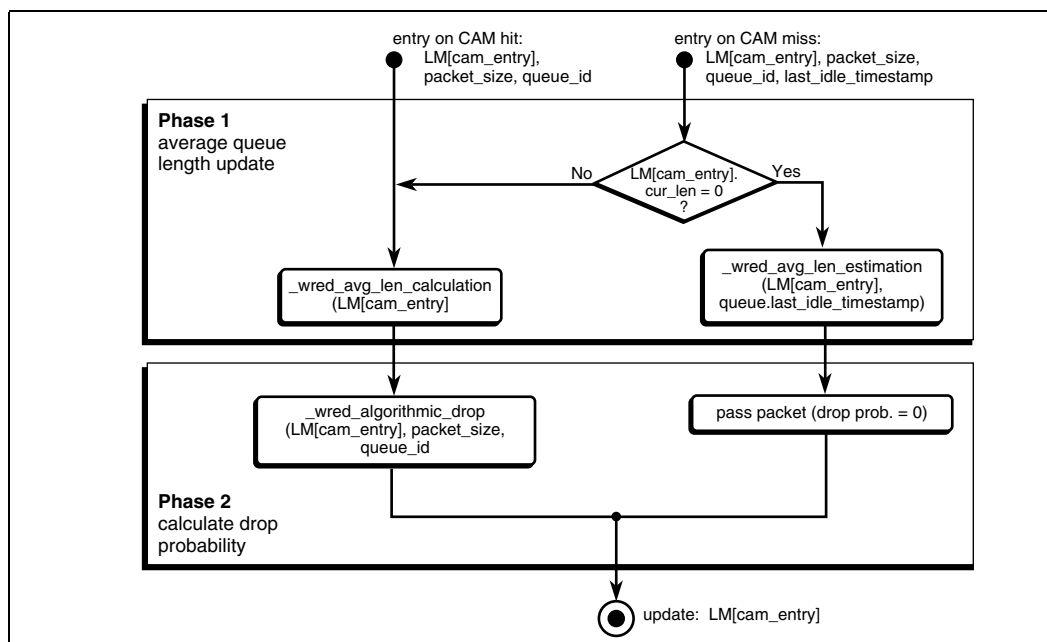
34.6.2 WRED Algorithm

34.6.2.1 WRED for low-speed queues

Figure 34-7 illustrates two phases of a WRED algorithm, executed by a single thread. As in the original Random Early Detection Gateways for Congestion Avoidance (<http://citeseer.nj.nec.com/floyd93random.html>), the first phase concerns updating of an average queue length. In the second stage, a thread calculates packet dropping probability. The algorithm follows a reference implementation proposed in Fig. 17 in Random Early Detection Gateways for Congestion Avoidance (<http://citeseer.nj.nec.com/floyd93random.html>). However, the original algorithm has been adapted DiffServ requirements, and optimized for the IXP2400 and IXP2800 Network Processor environments. In particular:

- The `dl_color_id` variable selects one of three RED instances, corresponding to a packet color. The `dl_color_id` is used to setup Local Memory index.
- The algorithm utilizes knowledge on CAM lookup result. If a CAM hit occurs, the queue can be considered non-empty because another thread must have received a packet to this queue. As a result, the queue state can be checked only on CAM miss.
- An arriving packet is not dropped, if a queue is empty. This approach follows RED in a Different Light (<http://citeseer.nj.nec.com/jacobson99red.html>), rather than the original Random Early Detection Gateways for Congestion Avoidance (<http://citeseer.nj.nec.com/floyd93random.html>). However, it is used because it significantly shortens the longest path of the algorithm execution.
- Packets are dropped, and not marked as suggested in Random Early Detection Gateways for Congestion Avoidance (<http://citeseer.nj.nec.com/floyd93random.html>). For a dropped packet, the `dl_next_block` variable is set to `IX_DROP`.

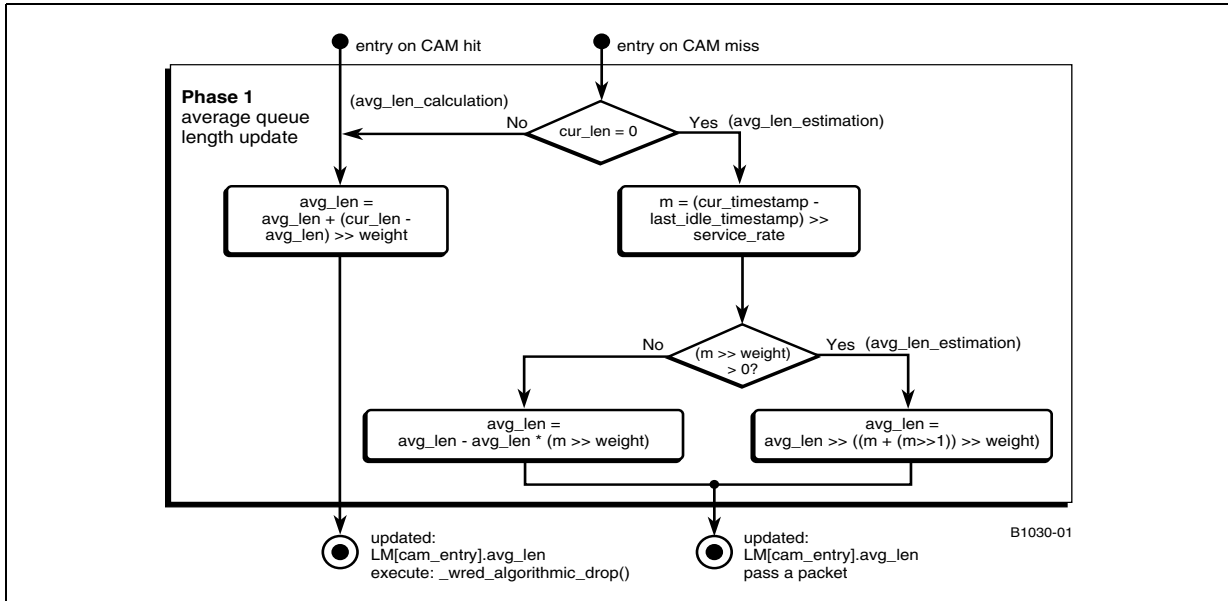
Figure 34-7. WRED Algorithm for Low-speed Queues



34.6.2.1.1 PHASE 1: Average Queue Length Update

Random Early Detection Gateways for Congestion Avoidance (<http://citeseer.nj.nec.com/floyd93random.html>) specifies two update formulas. A canonical EWMA equation (1) is applied if a queue holds packets. In the IXP2400 and IXP2800 Network Processors, the canonical equation can be also used on CAM hits regardless of the queue state. A CAM hit indicates that the previous packet has arrived no earlier than 8 functional pipe-stages before. The inter-arrival time cannot be bounded on a queue empty condition in CAM miss. In such case, Random Early Detection Gateways for Congestion Avoidance (<http://citeseer.nj.nec.com/floyd93random.html>) simulates EWMA filter invocations by estimating the number of packets that could arrive in an idle period.

Figure 34-8. WRED algorithm - phase 1



According to Random Early Detection Gateways for Congestion Avoidance (<http://citeseer.nj.nec.com/floyd93random.html>), if the queue is nonempty,

$$\text{avg_len} = \text{avg_len} + W * (\text{cur_len} - \text{avg_len}) \quad (1)$$

To ensure efficient implementation, EWMA gain value, W , is restricted to negative powers of 2—that is, $W = 2^{-\text{weight}}$. This is a commonly adopted industry solution. Note also that cur_len cannot go beyond avg_len by more than 2^{weight} . Since the maximum avg_len is 255, and the maximum weight is 15, the instantaneous length cannot exceed $255 + 2^{15}$. In other words, it must be a 16-bit value.

The average queue length, avg_len , is encoded on 2 bytes, as a fixed point value. The most significant byte stores an integer part, and the lower byte encodes a fraction part. Therefore, the instantaneous cur_len should be shifted 8 bits left prior to subtraction:

$$\text{avg_len} = \text{avg_len} + ((\text{cur_len} \ll 8) - \text{avg_len}) \gg \text{weight} \quad (1a)$$

A more challenging issue concerns handling an empty queue condition. In Random Early Detection Gateways for Congestion Avoidance (<http://citeseer.nj.nec.com/floyd93random.html>), a node estimates a number of packets that could have been transmitted in the idle period. It does it using average service time of a packet, S .

$$m = (\text{cur_timestamp} - \text{last_idle_timestamp}) / S \quad (2)$$

Then, the average queue length can be computed by

$$\text{avg_len} = (1 - W) m * \text{avg_len} \quad (3)$$

Providing again that W is a negative power of 2, equation (3) can be expressed as:

$$\text{avg_len} = (1 - 2^{-\text{weight}}) m * \text{avg_len} \quad (3a)$$

Random Early Detection Gateways for Congestion Avoidance (<http://citeseer.nj.nec.com/floyd93random.html>) suggests that it would be reasonable to express the multiplicand in equation (3a) by a power of 2. The presented design follows this approach:

- Approximation of typical packet transmission time, S , to a power of $2^{\text{service_time}}$.
The `service_time` should be configured so that the resulting S value is expressed in MEv2 timestamp units (16 ME cycles). Thus, the number of packets that could have been transmitted equals:

$$m = (\text{cur_timestamp} - \text{last_idle_timestamp}) \gg \text{service_time}$$

- Approximation of $(1 - 2^{-\text{weight}})^m$ expression to a power of 2^{-p}
Here, we are looking for an approximation value for p so that the approximation can be done with a simple, indirect shift MEv2 instruction.

$$\begin{aligned} (1 - 2^{-\text{weight}})^m &= 2^{-p} \\ \Rightarrow m * \ln(1 - 2^{-\text{weight}}) &= -p * \ln(2) \\ \Rightarrow -m * \ln(1 - 2^{-\text{weight}}) &= p * \ln(2) \\ \Rightarrow p &= -1.44 * m * \ln(1 - 2^{-\text{weight}}) \end{aligned}$$

$$\text{We know } \ln(1 + x) = x - (x)^2/2 + (x)^3/3 - \dots \text{ for } (-1 < x < 1)$$

For the approximation purpose, we consider only $\ln(1 + x) = x$. The approximation error is small, because typical weights result in $2^{-\text{weight}}$ close to 0. Thus, p approximates to:

$$p \approx 1.5 * m * 2^{-\text{weight}}$$

The p value can be efficiently estimated with one addition and two shift operations:

$$\begin{aligned} p &= (m + (m \gg 1)) \gg \text{weight} = ((m \ll 1) + m) \gg (\text{weight} + 1) \\ \Rightarrow \text{avg_len} &\leftarrow \text{avg_len} \gg ((m + (m \gg 1)) \gg \text{weight}) \end{aligned}$$

The above estimate works fine if the shifted p value remains non-zero—that is, m is higher than 2^{weight} . However, as long as $m < 2/3 * 2^{\text{weight}}$, the truncated p value is zero. As a result, the average queue length does not change. Therefore, for m values smaller than 2^{weight} , we use another approximation, derived from the Newton formula. The estimation is based on three first elements of the Newton sum, so that:

$$(1 - 2^{-\text{weight}})^m \approx 1 - m * 2^{-\text{weight}} + m * (m - 1) / (2 * 2^{-\text{weight}*2})$$

Assuming the approximation:

$$m - 1 \approx 2^{-\text{weight}}$$

we get the formula:

$$(1 - 2^{-\text{weight}})^m \approx 1 - m * 2^{-\text{weight}} + m * 2^{-(\text{weight}+1)}$$

The above equation can be converted to one multiplication and one shift operation, so finally the result is:

$$(1 - 2^{-\text{weight}})^m \approx 1 - (m \gg (\text{weight} + 1))$$

$$\Rightarrow \text{avg_len} \leftarrow \text{avg_len} - ((\text{avg_len} \bullet m) \gg (\text{weight} + 1))$$

34.6.2.1.2 PHASE 2: Packet Dropping

In this stage, a decision is made whether to pass or drop a packet. Three sets of dropping parameters are configured, respectively for green, yellow and red packet color.

In the original Random Early Detection Gateways for Congestion Avoidance (<http://citeseer.nj.nec.com/floyd93random.html>) algorithm, a packet drop probability is in range $\langle 0, 1 \rangle$. For the IXP2400 and IXP2800, this range is scaled up to $\langle 0, 65535 \rangle$ and only integer values are used. The constants used to calculate the dropping probability are also scaled accordingly:

$$\text{drop_prob} = (\text{avg_len} - \text{min_th}) * \text{max_prob} * 65536 / ((\text{max_th} - \text{min_th}) \gg 8)$$

The two latter factors do not change on a per-packet basis, so they can be calculated upon WRED configuration. The resulting constant is then stored in WRED tables (const field). Random Early Detection Gateways for Congestion Avoidance (<http://citeseer.nj.nec.com/floyd93random.html>) suggests that this constant value could be a power of 2, so that multiplication can be replaced with a shift instruction. The present design does not follow this suggestion, so:

$$\text{drop_prob} = (\text{avg_len} - \text{min_th}) * \text{const}$$

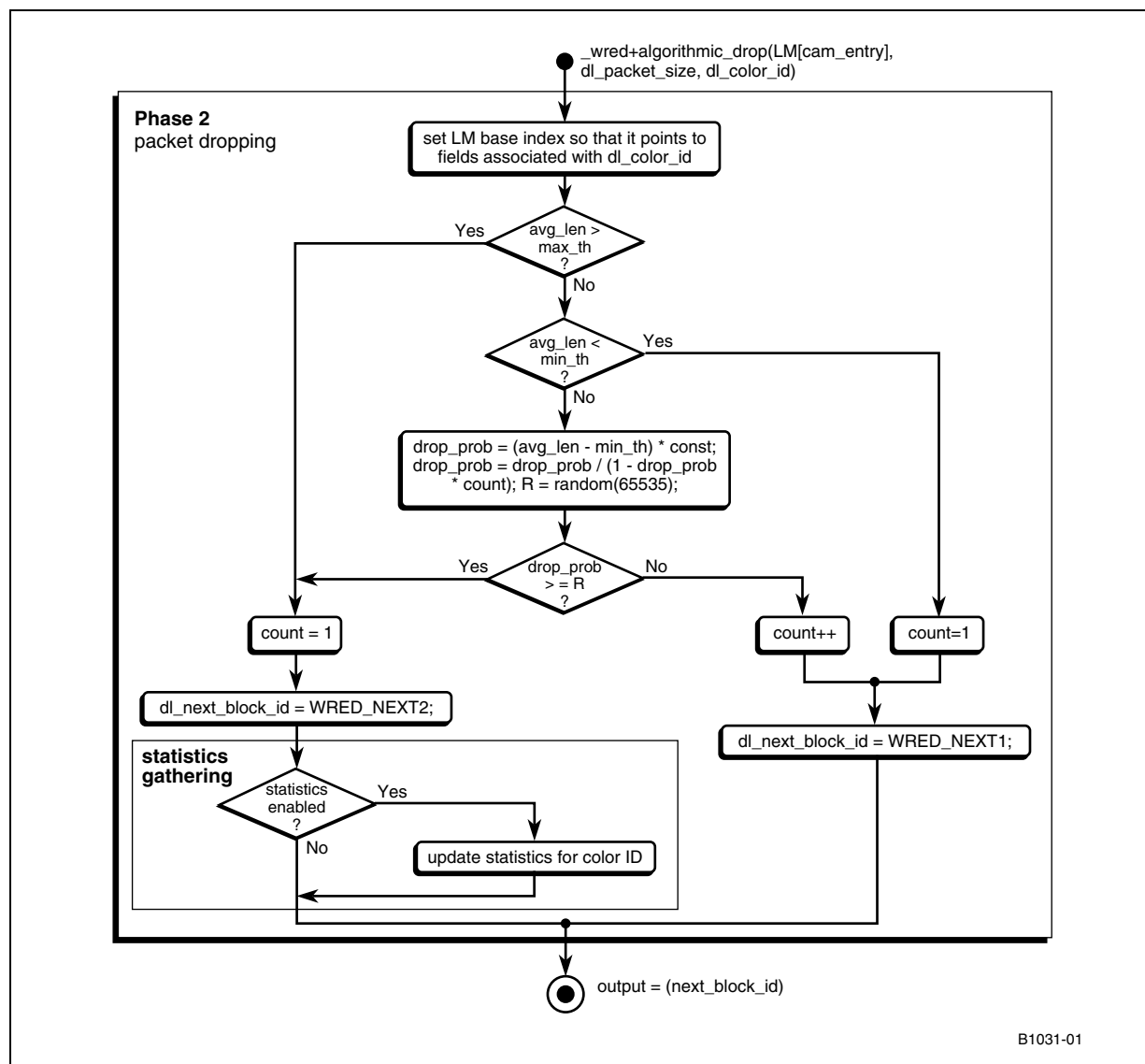
In order to avoid clustering of packet drops, Random Early Detection Gateways for Congestion Avoidance (<http://citeseer.nj.nec.com/floyd93random.html>) suggests that the dropping probability increases with the number of non-dropped packets so that intervals between packet drops adhere to a uniform distribution. As per Random Early Detection Gateways for Congestion Avoidance (<http://citeseer.nj.nec.com/floyd93random.html>), this is achieved by:

$$\text{drop_prob} \leftarrow \text{drop_prob} / (1 - \text{count} * \text{drop_prob})$$

Finally, the algorithm compares the resulting drop probability with a random value. A random value R is drawn upon each packet, and not on packet drops only as suggested in Random Early Detection Gateways for Congestion Avoidance (<http://citeseer.nj.nec.com/floyd93random.html>). This is more efficient because MEv2 has embedded a random number generator. Strict Random Early Detection Gateways for Congestion Avoidance (<http://citeseer.nj.nec.com/floyd93random.html>) implementation would require keeping R value in SRAM memory.

Due to performance goals, the uniform drops distribution feature should be available as a compile time option. In an OC-48 application, it can be omitted to fit into the cycle budget.

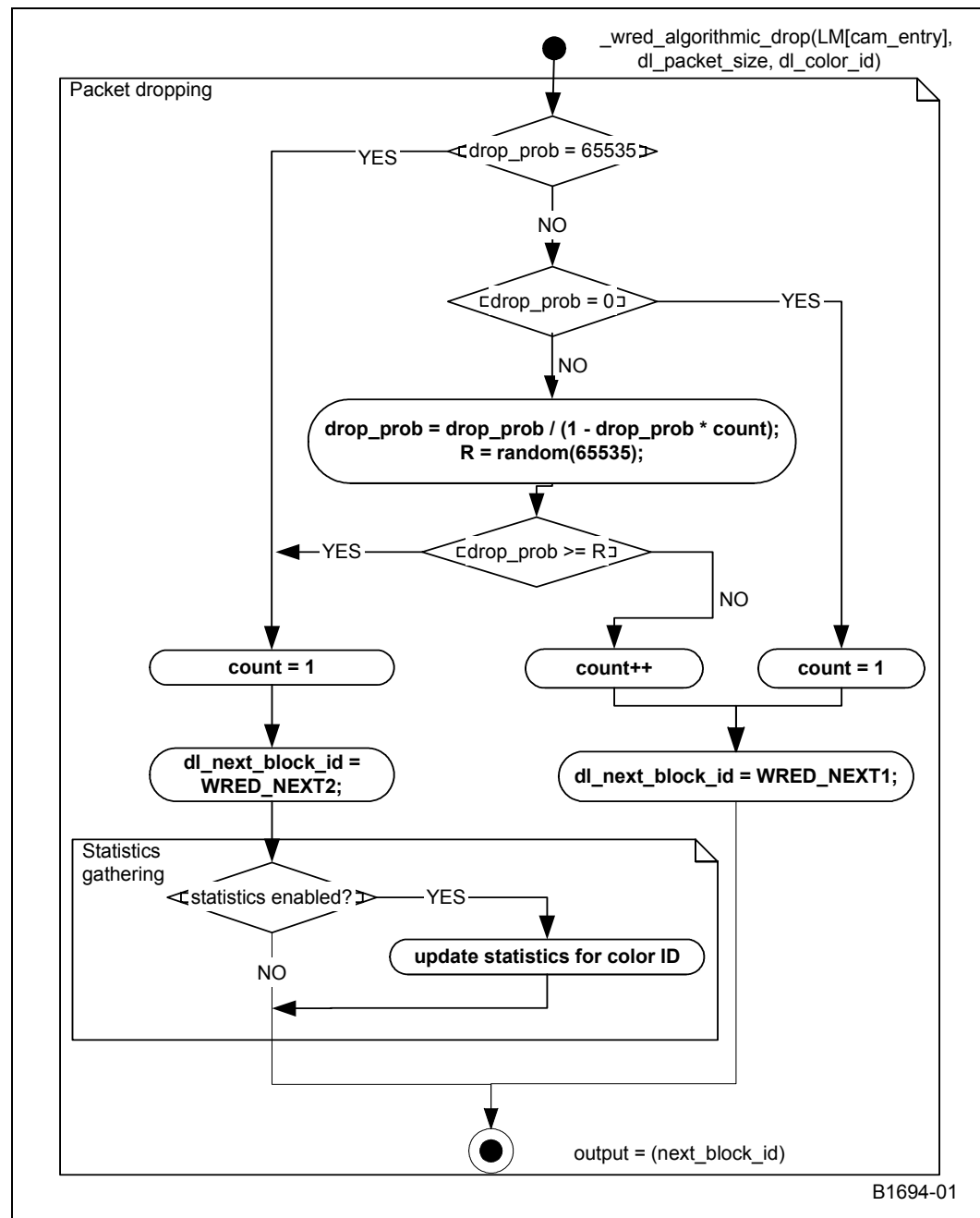
Figure 34-9. WRED algorithm - phase 2



34.6.2.2 WRED for high-speed queues

In this version of the microblock, the algorithm contains only the second phase (packet dropping), since the first phase (calculating the average queue count) is done by the core component). Figure 10 10 presents the algorithm. The uniform distribution feature is available as a compile-time option.

Figure 34-10. WRED Algorithm for High-speed Queues



34.7 Micro-code Budget

This section presents micro-code budget for two versions of the microblock—WRED for low speed queues and WRED for high speed queues.

34.7.1 Micro-code Budget for WRED Low-speed Queues

34.7.1.1 Performance Analysis

Table 34-6. Cycle Count Table (including unfilled defers)

Phase	Best case (all threads) ¹	Worst case (all threads)	Best Case (one thread) ²	Worst case (one thread) ³
Synchronization & CAM clear	6 1/8	6 1/8	6 (CTX > 0)	7 (CTX = 0)
SRAM read	20 4/8	31	19 (CAM hit)	31 (CAM miss)
WRED avg. length estimate(excludes dropping stage)	7 2/8	30	4 (CAM hit or queue len > 0)	30 (CAM miss & queue len = 0)
WRED avg. length calculate(excludes avg. len estimate)	12	13	12 (new avg len <= 216)	13 (new avg len > 216)
WRED dropping stage(excludes avg. len estimate)	8	32	8 (avg len > max_th)	32 (random < new_drop_prob)
WRED stage (including all the three above)	24	49	24 (4+12+8)**	49 (4+13+32)***
SRAM write	8	16	8 (no write)	16 (CTX = 7 or writeout)
Total	58 5/8	102 1/8	57	103
+ statistics overhead	6 4/8	17	5	17

1. best case for all threads corresponds to 1 miss and 7 hits; worst case - 8 misses

2. worst case for one thread for the whole WRED stage: CAM hit or queue len > 0; new avg len < 216; avg_len > max_th

3. worst case for one thread for the whole WRED stage: CAM hit or queue len > 0; new avg len > 216; random < new_drop_prob

Table 34-7 does not include SRAM atomic increments to update 64-bit statistics. These operations are very rare (~ once per 1 million packets).

Table 34-7. I/O Latency Analysis Table—WRED Low-speed Queues

Operation	I/O type	Accesses	Transfer size (in 32-bit long words)
Load WRED entry on CAM miss	SRAM read	1	13 or 7 (statistics compiled in or not)
Load QD entry on CAM miss	SRAM read	1	3
Flush WRED entry on exit	SRAM write	1	8 or 2 (statistics compiled in or not)

Table 34-8. Memory Access Summary Table

I/O type	Accesses
SRAM	3

34.7.1.2 Memory Footprint Analysis

Table 34-9. SRAM Footprint

SRAM Data structures	Size (bytes)
WRED table (entry size is 8 or 16 long words ¹)	8 kB (256 queues x 8 long words) or 16 kB (256 queues x 16 long words)
64-bit statistics table (entry size is 8 long words)	8 kB (256 queues x 8 long words)
Total	8 or 24 kB

1. depending on whether the microblock implements statistics

Table 34-10. Code Store Footprint

Code Store	Size (instruction)
WRED algorithm without statistics	157
Statistics overhead	34
Total	191

34.7.2 Micro-code Budget for WRED High-speed Queues

34.7.2.1 Performance Analysis

Table 34-11. Cycle Count Table (including unfilled defers)

Phase	Best case (all threads) ¹	Worst case (all threads)	Best Case (one thread)	Worst case (one thread)
Synchronization & CAM clear	6 1/8	6 1/8	6 (CTX > 0)	7 (CTX = 0)
SRAM read	16 5/8	21	16 (CAM hit)	21 (CAM miss)
WRED dropping stage(excludes avg. len estimate)	11	25	11 (drop_prob = 0)	25 (random < new_drop_prob)
SRAM write	7 4/8	10 1/8	7 (no writeback)	11 (CTX = 7)10 (writeback)
Total	41 2/8	62 2/8	57	103
+ statistics overhead	6 4/8	17	5	17

1. best case for all threads corresponds to 1 miss and 7 hits; worst case - 8 misses

Table 34-12 does not include SRAM atomic increments to update 64-bit statistics. These operations are very rare (~ once per 1 million packets).

Table 34-12. I/O Latency Analysis Table - WRED for High-Speed Queues

Operation	I/O type	Accesses	Transfer size (in 32-bit long words)
Load WRED entry on CAM miss	SRAM read	1	4 or 2 (statistics compiled in or not)
Load QD entry on CAM miss	SRAM read	1	3
Flush WRED entry on exit	SRAM write	1	3 or 1 (statistics compiled in or not)

Table 34-13. Memory Access Summary Table

I/O type	Accesses
SRAM	3

34.7.2.2 Memory Footprint Analysis

Table 34-14. SRAM Footprint

SRAM Data structures	Size (bytes)
WRED table (entry size is 8 or 16 long words ¹)	8 kB (256 queues x 8 long words) or 16 kB (256 queues x 16 long words)
64-bit statistics table (entry size is 8 long words)	8 kB (256 queues x 8 long words)
Total	8 or 24 kB

1. Depending on whether the microblock implements statistics

Table 34-15. Code Store Footprint

Code Store	Size (instruction)
WRED algorithm without statistics	76
Statistics overhead	26
Total	102

MPLS Microblocks

The MPLS microblocks include the following:

- [Chapter 35, “FTN Forwarder Microblock”](#)
- [Chapter 36, “ILM Forwarder Microblock”](#)

The microblocks described in this section handle MPLS packets. The table below summarizes the MPLS microblocks supported on the SDK:

Microblock	Description	Usage	Cycle Budget
MPLS ILM (Switching)	Provides LSR and Egress LER functionality	Runs on multiple microengines in parallel. Number of microengines used depends on required data rate (OC-12 to OC-192)	97 for OC48 POS57 for OC192 POS94 10 GBE
MPLS FTN (Marking)	Provides Ingress LER functionality.	Runs on multiple microengines in parallel. Number of microengines used depends on required data rate (OC-12 to OC-192)	97 for OC48 POS57 for OC192 POS94 10 GBE

35.1 Overview

The FTN (FEC to NHLFE Map) Forwarder microblock implements (in co-operation with the IPv4 forwarder) the functionality of an ingress LER.

35.2 Functionality

The FTN Forwarder receives IP packets processed by either the 6-tuple classifier, or the LPM part of the IPv4 forwarder. It processes only packets that have the `dl_nexthop_id_type` variable set to `NHID_MPLS(2)`. In this case the `dl_next_hop_id` variable of the dispatch loop contains a FEC ID (set by the 6-tuple classifier or the LPM) that is an index to the NHLFE table, and the `dl_header_type` variable is set to the value of `IPV4_TYPE(6)`.

First, the FTN Forwarder reads the NHLFE pointed to by the FEC ID. If the entry constitutes an NHLFE set, it chooses one element of this set. The algorithm used to determine an NHLFE from a set can implement load sharing or be based on DiffServ information for the packet. In this version of the FTN Forwarder the first element of the set is taken.

The NHLFE specifies the next hop data for the packet (the egress blade, output port, L2 Table index), the DiffServ flags and tunnel model of the LSP, and the label(s) to be pushed on the label stack.

Then, the FTN Forwarder determines the outgoing TTL (`oTTL`) for the packet, based on the IP header TTL field value and the TTL decrement value specified in the NHLFE. If the `oTTL` is lower than 1, the packet is sent to the MPLS core component with the exception code `TTL_EXPIRED`. Otherwise, the packet is labeled according to the label stack specified in the NHLFE, the packet's meta-data (packet size, buffer offset, etc.) is updated, and the dispatch loop variables specifying the next hop (`dl_next_hop_id`, `dl_output_port_egress`, `dl_output_port_fabric`) are set according to the values in the NHLFE.

Finally, when the labeled packet is ready for transmission, its total length is checked against the `maxLabPktSize` parameter value specified in the NHLFE. At this point the packet may be sent to the MPLS CC with the `LABELLED_PACKET_TOO_BIG` exception code. Otherwise, optional segment (LSP) statistics counters are updated, and the packet is directed to the next microblock (Queue Manager or WRED) for sending.

35.3 Assumptions and Dependencies

35.3.1 Assumptions

It is assumed that the FTN Forwarder operates within the same operational environment as the IPv4 forwarder. This operational environment includes:

- Usage of hardware resources
 - Mapping to microengine
 - Usage of scratch rings
 - Usage of DRAM and SRAM
- Format of entries in scratch rings
 - Receive scratch ring from POS RX microblock
 - Transmit scratch ring to Ingress Queue Manager and Scheduler
 - Meta data exchanged with preceding and following microblocks
 - Format of the meta data prepended to the data buffer, while traveling through the CSIX fabric to an Egress Network Processor.

The FTN Forwarder does not perform IP header validation as it is performed by the IPV4 forwarder microblock.

35.3.2 Component Interfacing

On input, the FTN Forwarder microblock depends on classifiers implemented in the following components:

- IP filters microblock - 6-tuple classifier
- IPv4 microblock - IPv4 LPM classifier

In case of classifying a packet as belonging to an MPLS FEC, these classifiers should set the `dl_nexthop_id_type` dispatch loop variable to the value of `NHID_MPLS(2)`, and return the lookup result in the `dl_next_hop_id` variable.

The details of component interfacing are as follows.

In the case of the ingress LER, the `DL_Source` microblock gets an IP packet from the scratch ring populated by an Rx microblock, and initializes the dispatch loop variables accordingly to the packet's properties. For this discussion it is essential that the `dl_next_hop_id` variable is set to an illegal value (-1). Then the packet is passed to the 6-tuple classifier, or the IPv4 forwarder, if the 6-tuple classifier is not present. If the 6-tuple classifier finds an MPLS rule matching the packet, it modifies the `dl_next_hop_id` variable with the lookup result, sets the `dl_nexthop_id_type` variable to the value of `NHID_MPLS(2)`, and hands over the packet to the IPv4 forwarder.

The IP forwarder checks the `dl_next_hop_id` value - if it is equal to (-1), the forwarder performs the longest prefix match (LPM) lookup on the packet's IP destination address. The lookup result is an index to the next hop information for the packet that is saved to the `dl_next_hop_id` variable. Otherwise (`dl_next_hop_id != -1`), the LPM is skipped - the packet has already been classified. At this moment the `dl_next_hop_id` value is used to retrieve an entry from the next hop database

(NHD) table. The IP forwarder sets the `dl_next_hop_id_type` and modifies `dl_next_hop_id` variables according to the corresponding fields of the NHD entry. The `next_hop_id_type` value discriminates between NHD entries created by the IP and MPLS core components. If the retrieved entry belongs to MPLS, the `dl_nexthop_id_type` is set to `NHID_MPLS` and the IP forwarder microblock passes control to the FTN Forwarder, otherwise it continues processing the next hop information.

The FTN Forwarder microblock first checks the `dl_next_hop_id_type` value. If the `dl_next_hop_id_type` value is not equal to `NHID_MPLS`, the FTN Forwarder executes an empty bypass. Otherwise, it begins to process the packet in the way described in [Section 35.2, “Functionality” on page 605](#).

On output, the FTN Forwarder microblock depends on the L2 encapsulator microblock on the egress NP. This microblock, according to the meta-data generated by the FTN Forwarder, should prepend an appropriate L2 header to the data packet before passing it to a transmitting microblock.

35.4 Microblock Interfaces

35.4.1 Input Microblock Variables

This section lists all variables read by the FTN Forwarder block.

Table 35-1. Input Variables Consumed by FTN Forwarder

Input variable	Size	Type ¹	Description
<code>dl_nexthop_id_type</code>	4 bits	M	Indicates the <code>dl_next_hop_id</code> contents, if: <ul style="list-style-type: none"> <code>NHID_IP(0)</code>—index into L2 table <code>NHID_MPLS(2)</code>—index into NHLFE table (FEC ID or outSegId) The FTN Forwarder block executes an empty bypass path if <code>dl_next_hop_id_type != NHID_MPLS(2)</code>
<code>dl_next_hop_id</code>	16 bits	M	FEC ID or outSegId - indicates an entry in the NHLFE table that should be used to find next hop information for the packet.
<code>dl_packet_size</code>	16 bits	M	Number of bytes in the currently processed packet.
<code>dl_packet_offset</code>	16 bits	M	Offset to the beginning of the packet in the buffer.
<code>dl_input_port</code>	16 bits	M	Logical port number on which the packet was received.

1. M and S letters indicates that this variable is part of the packet metadata and is transported over switch fabric to the egress NP.

35.4.2 Output Microblock Variables

This section lists all variables modified and set by the FTN Forwarder block.

Table 35-2. Output Variables Modified by FTN Forwarder

Input variable	Size	Type ¹	Description
dl_next_block	8 bits	M	Specifies the output of the FTN Forwarder. It can be one of the following: <ul style="list-style-type: none"> IX_DROP - the packet should be dropped IX_EXCEPTION - the packet should be passed to the MPLS core component. In this case the variable dl_exception_code specifies further details about the exception BID_MPLSFTN_NEXT1 - the packet should be passed to the DI_Sink block and normally forwarded to the egress NP
dl_exception_code	8 bits	M	If dl_next_block is equal to IX_EXCEPTION, it can be one of the following: <ul style="list-style-type: none"> TTL_EXPIRED - the packet's outgoing TTL value is lower than 10 LABELED_PACKET_TOO_BIG—the MPLS-labeled packet's length exceeds the "Effective Maximum Frame Payload Size for Labeled Packets" (maxLabPktSize) parameter of the output port
dl_exception_id	8 bits	M	If dl_next_block is equal to IX_EXCEPTION, this field is set to the FTN Forwarder microblock ID (BID_MPLSFTN)
dl_next_hop_id_type	4 bits	M	Set to NHID_IP(0), indicates that the dl_next_hop_id variable contains and index to the L2 encapsulation table on the egress node
dl_next_hop_id	16 bits	MS	Indicates an entry in the egress L2 table that should be used to perform an appropriate L2 encapsulation of the data packet.
dl_output_port_fabric	8 bits	M	Egress blade id.
dl_output_port	16 bits	MS	Number of the port interface on which the packet is to be transmitted in a given blade
dl_packet_size	16 bits	MS	Number of bytes in the currently processed packet.
dl_packet_offset	16 bits	MS	Offset to the beginning of the packet in the buffer.

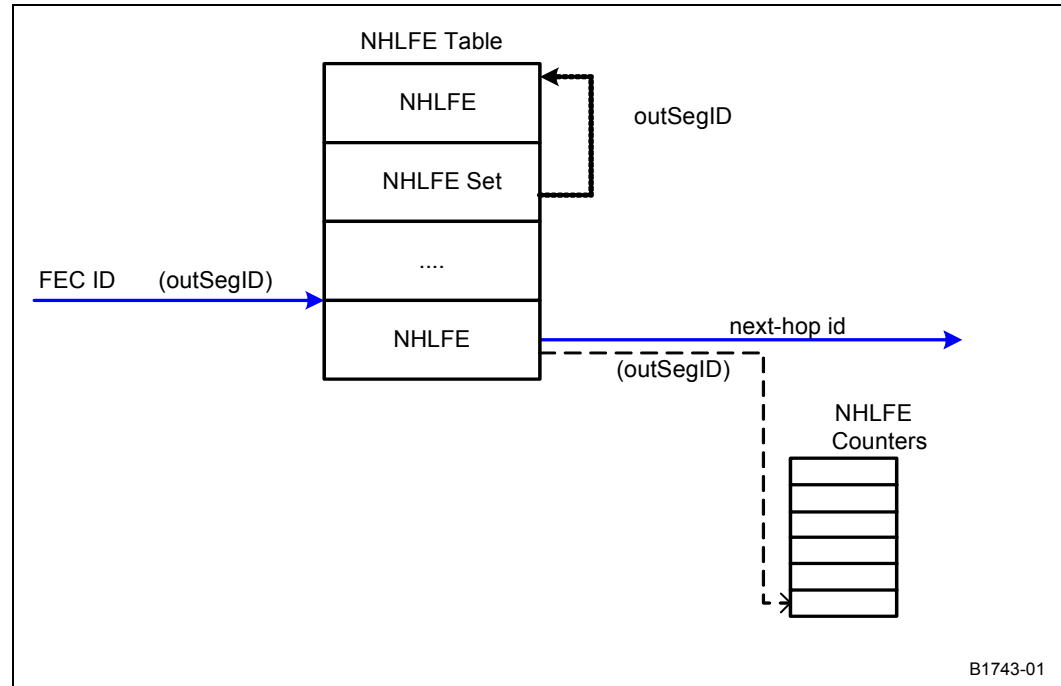
1. M and S letters indicates that this variable is part of the packet metadata and is transported over switch fabric to the egress NP.

35.5 Data Structures and Forwarding Algorithm

35.5.1 Overview

Figure 35-1 illustrates the FTN Forwarder microblock main data structures and their interconnections.

Figure 35-1. FTN Forwarder Data Structures



The NHLFE table entries map FECs to NHLFE entries. It is a linear table, indexed by the outSegID or FEC ID (conveyed in the next_hop_id metadata variable) value, obtained as the result of packet classification performed in either the 6-tuple Classifier or IP LPM part of the IPv4 forwarder microblocks. Allocation of outSegID (FEC ID) values is performed by the MPLS forwarder core component. According to the MPLS standard, a FEC can be assigned to one or more NHLFEs. The latter case is implemented by introducing an intermediate structure - NHLFE set, stored in the NHLFE table in the place of a regular NHLFE entry. The NHLFE set can be distinguished from a regular NHLFE entry by a dedicated flag value.

The NHLFE set is a table comprising 4 elements. Each element of this table contains an index to an entry in the NHLFE table (outSegID) and a criteria field (e.g. phbMask), used for choosing an actual NHLFE from the set. In case of the DiffServ/MPLS application, the criteria field could be a bit mask denoting the PHBs supported by the NHLFE pointed to by this set element. The FTN Forwarder would choose the proper NHLFE by matching the dl_class_id variable value with the bit position of the phbMask field.

A regular NHLFE entry specifies the tunnel mode, DiffServ flags, and the label(s) that should be applied to a packet, as well as the next hop data for the packet.

The NHLFE Counters table comprises per-NHLFE entry structures accumulating various statistics relevant to MPLS packet forwarding. It is a linear table, indexed by the outSegID value. It is separated from the NHLFE table for efficiency reasons - it can be compiled out of the code.

Note that the FTN Forwarder does not gather MPLS statistics for output ports - they will be maintained by the Layer 2 Encapsulation microblock.

35.5.2 NHLFE Table

The NHLFE table is a linear table indexed by the FEC ID or outSegID value. It can be placed in DRAM or SRAM, depending on a compilation constant. Each entry of this table can be either a regular NHLFE or an NHLFE set. [Table 35-3](#) shows a regular NHLFE layout and [Table 35-4](#) shows the layout of an NHLFE set.

Table 35-3. NHLFE Entry

LW	Bits	Size	Field	Description
0	31:24	8	flags	bits 31:22 hold the following bit or 2-bits flags: bit 31: MPLS_NHLFE_VALID_BIT—if set, the entry is valid, otherwise the entry is unallocated and bits 31:24 of the flags field should be equal to 0 bit 30: MPLS_NHLFE_SET_BIT—entry type: regular NHLFE if 0, NHLFE set if 1 bits 29:28 - MPLS_NHLFE_TUNNEL_MODEL: <ul style="list-style-type: none"> 00—Pipe model 01—Uniform model 10—Short Pipe model without PHP 11—Short Pipe model with PHP bits 27:26 MPLS_NHLFE_LSP_TYPE—LSP type: <ul style="list-style-type: none"> 00—non-DiffServ LSP 10—E-LSP 11—L-LSP bit 25: MPLS_NHLFE_TC_ENABLED_BIT—traffic conditioning enabled bit 24: MPLS_NHLFE_PRECONFIGURED_EXP_MARKING
0	23:22	2	reserved	Reserved(0)
0	21:19	3	pushCount	Number of labels to be added to the label stack, from 1 to 4
0	18:16	3	l3Protocol	The L3 protocol type of the MPLS payload: 0—UNKNOWN 6—IP_v4 7—IP_v6
0	15:0	16	nextHopID	ID used on the Egress Intel ^(R) IXP2400 Network Processor to lookup the outgoing link layer information
1	31:16	16	maxLabPktSize	Effective Maximum Frame Payload Size for labeled packets (LSP MTU), for the outPortID port
1	15:0	16	outPortID	Outgoing port ID—local port index on outgoing blade
2	31:12	20	topLabel	Top of stack label—either mpls shim or atm label
2	11:8	4	reserved	Set to 0
2	7:4	4	outPortType	Outgoing port type

Table 35-3. NHLFE Entry (Continued)

LW	Bits	Size	Field	Description
2	3:0	4	ttl	TTL field decrement value
3	31:12	20	label1	Label to be pushed below the topLabel, if the pushCount field value is greater than 1
3	11:8	4	reserved	Set to 0
3	7:0	8	bladeID	Outgoing blade ID w.r.t. CSIX fabric
4	31:12	20	label2	Label to be pushed below Label1, if the pushCount field value is greater than 2
4	11:8	4	reserved	Set to 0
4	7:0	8	flowIdHigh	MSB 8 bits of 20 bit flow id (not used in this release)
5	31:12	20	label3	Label to be pushed below Label2, if the pushCount field value field equals to 4
5	11:0	12	flowIdLow	LSB 12 bits of flow id
6	31:16	16	Exp2phb	Index into the Exp2Phb table (not used in this release)
6	15:0	16	Phb2Exp	Index into the Exp2Phb table (not used in this release)
7	31:0	32	reserved	Set to 0

The size of the NHLFE table depends on the number of entries in the IP FIB and the MPLS control plane aggregation capabilities. In the worst case, when each FEC is assigned to a separate FEC ID, the size of the NHLFE table can be calculated as:

$$(\text{no_of_IPv4_FIB_entries} + 4 * \text{no_of_set_entries} + \text{no_of_IPfilters_entries}) * 32 \text{ bytes}$$

35.5.3 NHLFE Set Table

Table 35-4 shows the layout of the NHLFE set entry.

Table 35-4. NHLFE Set Entry

LW	Bits	Size	Field	Description
0	31:24	8	flags	bits 31:24 hold the following bit flags: bit 31: MPLS_NHLFE_VALID_BIT - if set, the entry is valid, otherwise the entry is unallocated and the the whole flags byte should be equal to 0 bit 30: MPLS_NHLFE_SET_BIT - entry type: regular NHLFE if 0, NHLFE set if 1 Bits 30:24 - Reserved(0)
0	23:8	16	exp2phb	Index into the Exp2Phb table (not used by the FTN Forwarder)
0	7:0	8	reserved	Set to 0
1	31:0	32	phbMask1	Criterion used by the FTN forwarder for choosing one of the set elements.
2	31:0	32	phbMask2	Criterion used by the FTN forwarder for choosing one of the set elements.
3	31:0	32	phbMask3	Criterion used by the FTN forwarder for choosing one of the set elements.
4	31:0	32	phbMask4	Criterion used by the FTN forwarder for choosing one of the set elements.

Table 35-4. NHLFE Set Entry (Continued)

LW	Bits	Size	Field	Description
5	31:12	20	outSegID1	Index to the 1st NHLFE belonging to the set
5	11:0	12	outSegID2	Index to the 2nd NHLFE belonging to the set (12 most significant bits)
6	31:24	8	outSegID2	Index to the 2nd NHLFE belonging to the set (8 least significant bits)
6	23:4	20	outSegID3	Index to the 3rd NHLFE belonging to the set
6	3:0	4	outSegID4	Index to the 4th NHLFE belonging to the set (4 most significant bits)
7	31:16	16	outSegID4	Index to the 4th NHLFE belonging to the set (16 most significant bits)
7	15:0	16	reserved	Set to 0

35.5.4 NHLFE Counters Table

The NHLFE Counters table is placed in SRAM. It gathers statistics connected with packets forwarded according to a given NHLFE entry. Its entries correspond 1:1 to the NHLFE table entries. Table 35-5 shows the layout of the NHLFE Counters table entry.

Table 35-5. NHLFE Counters Table Entry

LW	Bits	Size	Field	Description
0	31:0	32	mplsNhlfeOctets	32-bit counter accumulating the number of octets received through this entry. A 64-bit version of this counter is maintained by the core component.
1	31:0	32	mplsNhlfePkts	a 32-bits counter accumulating the number of packets received through this entry
2	31:0	32	mplsNhlfeErrorOctets	a 32-bits counter accumulating the number of octets that were not sent through this entry because their TTL expired or they required fragmentation but had the DF bit set in the IP header
3	31:0	32	mplsNhlfeErrorPkts	a 32-bits counter accumulating the number of packets that were not sent through this entry because their TTL expired or they required fragmentation but had the DF bit set in the IP header

The size of memory needed for the NHLFE Counters table can be calculated as:

$$\text{number_of_NHLFE} * 16 \text{ bytes}$$

35.5.5 Forwarding Algorithm

This section presents the diagrams of the FTN Forwarder microblock forwarding algorithm.

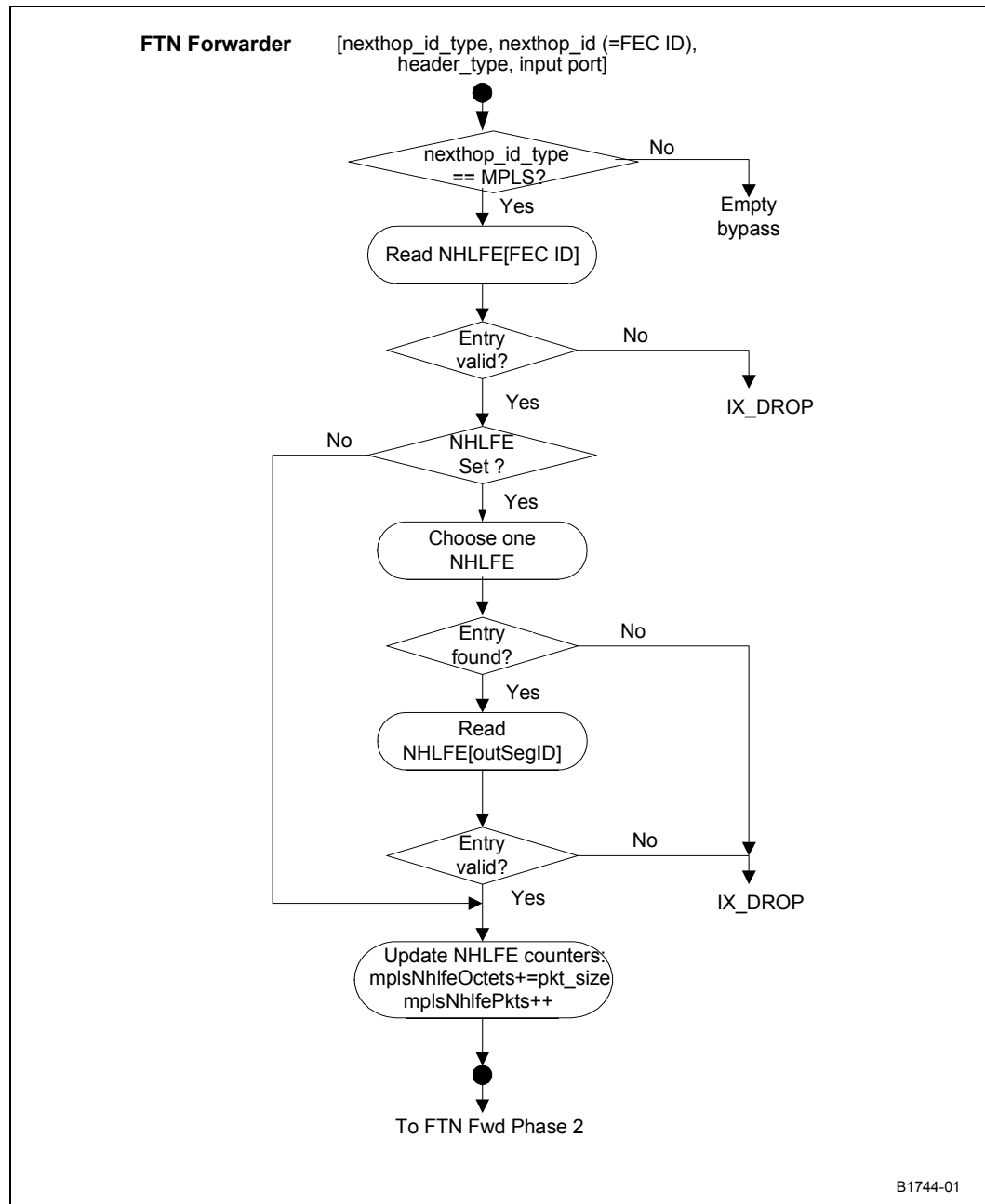
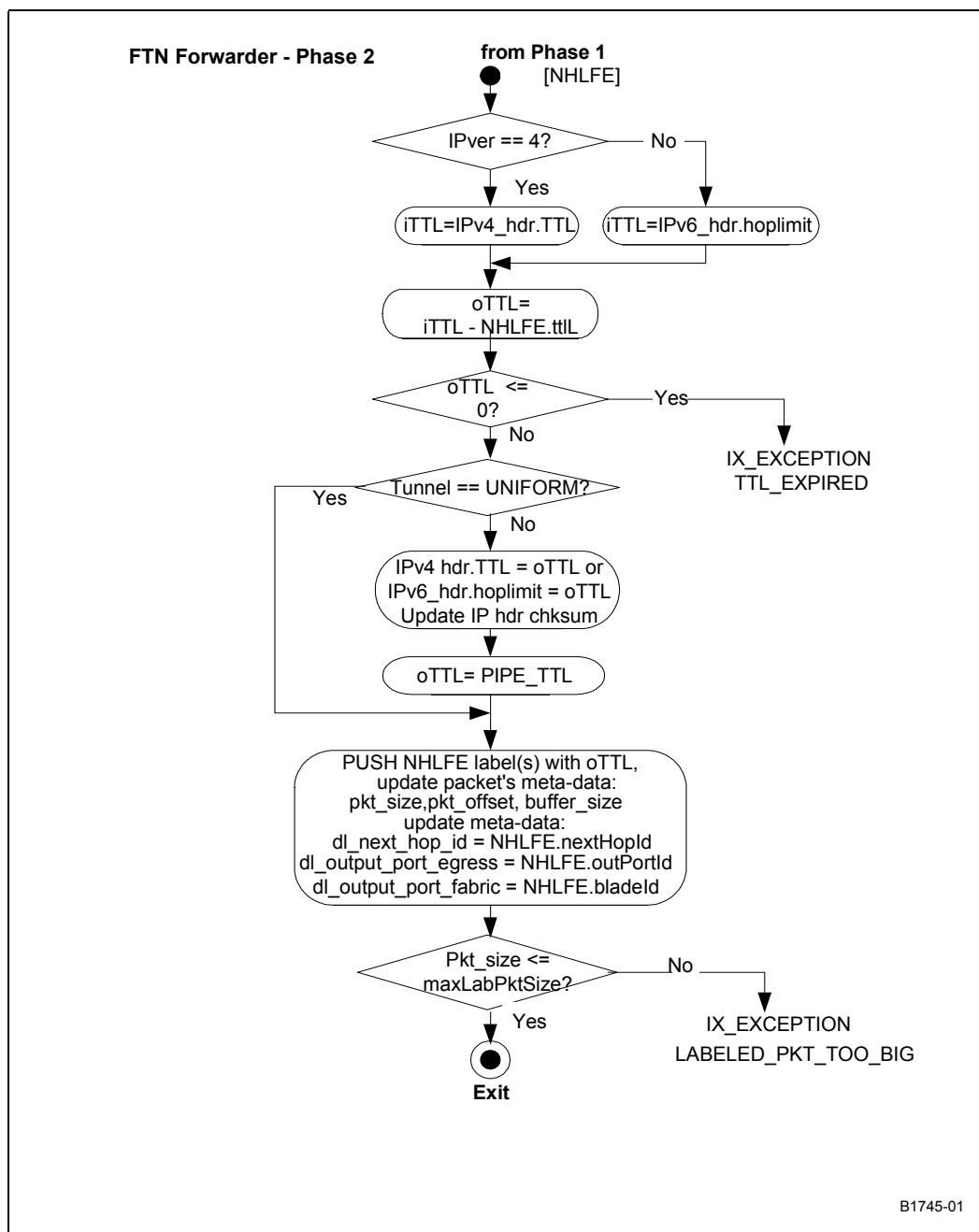
Figure 35-2. FTN Forwarder Forwarding Algorithm - Phase 1


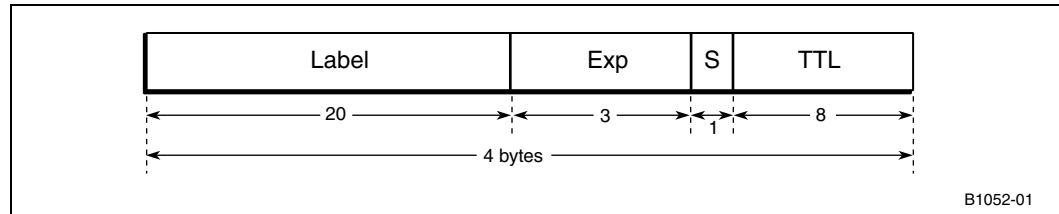
Figure 35-3. FTN Forwarder Forwarding Algorithm - Phase 2



35.5.6 Initial Label Stack Building

On an ingress LER, an IP packet is initially labeled, what means that one (or in special cases more) MPLS label stack entry (MPLS shim) is prepended before the IP header. The format of a single MPLS stack entry is shown in [Figure 35-4](#).

Figure 35-4. MPLS Stack Entry Format



The `Label` field carries the actual value of the MPLS label. The `Exp` field is used for implementing DiffServ over MPLS. The `S` bit is set to one to mark the bottom of the stack (the last entry). The `TTL` field carries the time-to-live value.

According to the value of the `pushCount` field in the NHLFE table entry, the dispatch loop buffer offset variable is decremented by 4, 8, 12, or 16, to accommodate 1, 2, 3, or 4 MPLS stack entries. It is assumed that the RX block has left enough headroom in the packet buffer for up to four MPLS stack entries.

The `label-i` field of the NHLFE table entry stores the first three bytes of the stack entry, with the `S` bit always set to 0. These bytes are copied to the first three bytes of the appropriate stack entry in the packet. The `Exp` bits can be further modified for DiffServ-aware LSPs. Then, the `TTL` field of the top stack entry is set, and the `S` bit of the bottom stack entry is set to 1. In absence of nested tunnels, an ingress LER pushes only one label on the label stack, and this entry is both the top and bottom.

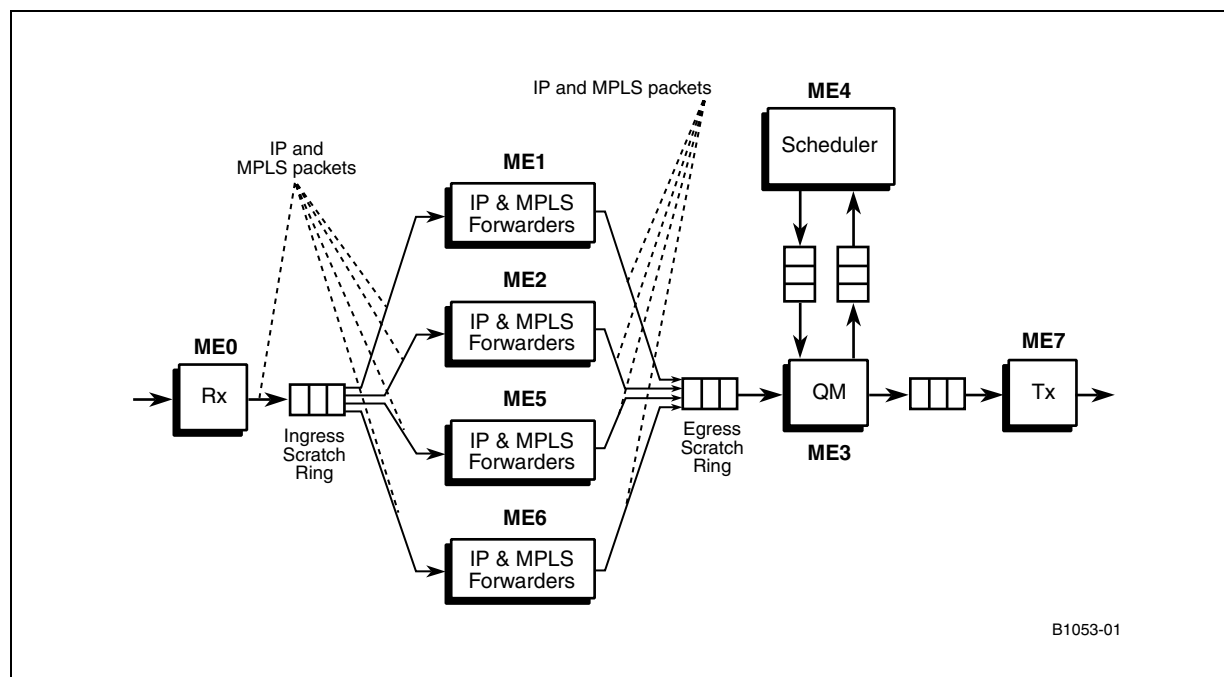
The MPLS Marker does not depend on the output port type. It sets appropriate variables of packet's meta data based on the contents of the `bladeID`, `outPortID` and fields from an NHLFE. According to the IXA Portability Framework software architecture, those variables are carried over the fabric to the egress blade, where the `L2 nextHopID` value is used as an index into the L2 table containing the whole L2 encapsulation needed for a particular port type (Ethernet, POS). A special case is only needed for cell mode MPLS over ATM, in which the top MPLS shim should be copied to the ATM Cell Header, so that cells can be switched based on the label, instead of standard VPI/VCI information.

35.5.7 Allocation to Microengines

This section describes allocation of microengines to the MPLS forwarder (comprising both the ILM and FTN Forwarders), and the IP forwarder. As it was shown in [Section 10.3, "Cooperation with IP and QoS Microblocks"](#) on page 129, the MPLS forwarder microblocks are elements of the IP functional pipeline. Therefore, in general, they are subject to the same patterns of microengine allocation as the IP forwarder.

In the design shown in [Chapter 2, "System Data Structures and Design Choices,"](#) the IP forwarder (and hence the MPLS forwarder) functional pipeline runs on 4 microengines (or 32 threads) of the ingress NP. The rest of the microengines are allocated to the RX, QM, Scheduler and TX microblocks. This is shown in [Figure 35-5](#).

Figure 35-5. IP and MPLS Microblocks on the Same Microengines



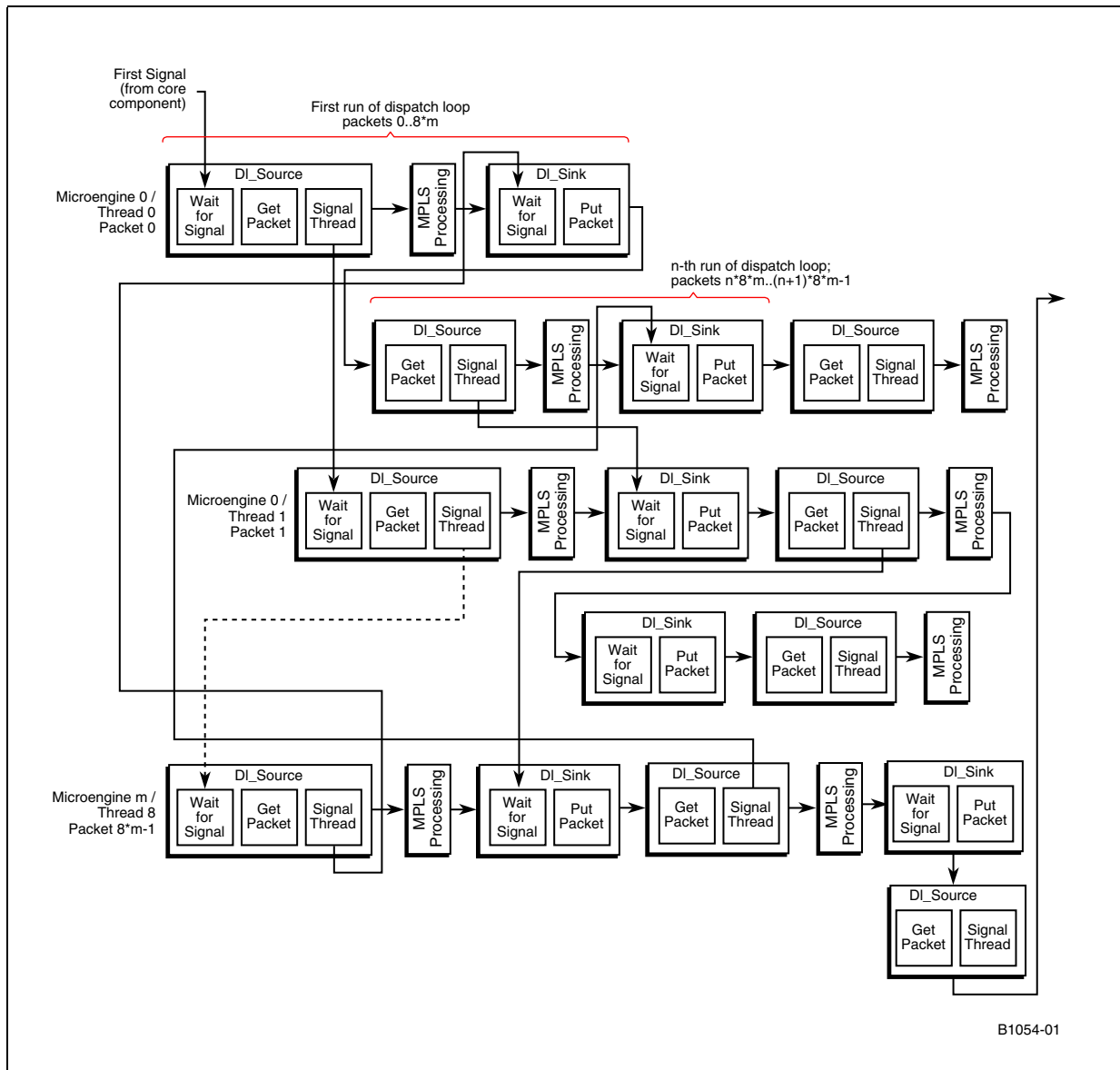
In the above arrangement, the RX microblock puts the received packets to one ingress scratch ring, common for all microengines running the IP/MPLS code. The demultiplexing of IP and MPLS packets is done in `Dl_Source` microblocks of individual threads on each microengine, based on the `dl_header_type` variable of the packet's metadata. After processing, IP and MPLS packets are put to a common egress scratch ring.

35.5.8 Thread Ordering and Synchronization

The MPLS forwarder microblocks (the ILM and FTM Forwarders) have no critical sections. Statistics counters (if enabled) are updated through atomic SRAM operations.

Thread ordering and synchronization is achieved by the `Dl_Source` and `Dl_Sink` microblocks signaling the beginning and end of each dispatch loop in the way explained below.

Figure 35-6 presents the operation of the dispatch loop. In general, the dispatch loop consists of `Dl_Source` microblock, microblocks containing the MPLS processing functionality and `Dl_Sink`.

Figure 35-6. FTN Forwarder Thread Synchronization


The construction of the dispatch loop ensures that packets are handled in a sequence. With an exception of the first dispatch loop run, the synchronization is shared between DL_Source and DL_Sink microblocks. DL_Sink microblock is allowed to post a packet into the Queue Manager queue only after the preceding microblock retrieved its packet from the RX queue. This ensures that given thread is allowed to post processed packet and read in new packet after the preceding thread done the same. In consequence, packets are processed in the order of arrival.

The first run of the microblock's dispatch loop is handled differently since there are no preceding DL_Sink microblocks. Here ordering is achieved in DL_Source microblocks, as indicated in [Figure 35-6](#).

35.5.9 FTN Forwarder Micro-code Budget

35.5.9.1 Performance Analysis

The following data reflects the main forwarding path of the current version of code.

Table 35-6. Cycle Count Table (including unfilled defers) - UNIFORM Tunnel Mode

Phase	Without statistics PUSH 1 label stack	Without statistics PUSH 4 label stack	With statistics PUSH 1 label stack ¹	With statistics PUSH 4 label stack ¹
Phase 1 without NHLFE set	15	15	15	15
Phase 2 - UNIFORM tunnel	42	57	51	66
Total	57	72	66	81

1. Can be compiled out of code

Table 35-7. Cycle Count Table (including unfilled defers) - PIPE Tunnel Mode

Phase	Without statistics, PUSH 1 label stack	Without statistics PUSH 4 label stack	With statistics, PUSH 1 label stack ¹	With statistics, PUSH 4 label stack ¹
Phase 1 without NHLFE set	15	15	15	15
Phase 2 PIPE tunnel	49	63	58	72
Total	64	78	73	87

1. Can be compiled out of code

Table 35-8. I/O Latency Analysis Table

Operation	I/O type	Accesses	Transfer size (in 32-bit long words)
NHLFE read without/with NHLFE set	SRAM or DRAM read	1/2	8
NHLFE counters update ¹	SRAM increment and SRAM add	2	1

1. Can be compiled out of code

35.5.9.2 Memory Footprint Analysis

Table 35-9. Data Structures Footprint

Data structures	Size (bytes)
Local Memory Cache	512 per ME (shared with ILM Forwarder)
NHLFE Table (either in SRAM or in DRAM)	2 MB (64 K entries, 32 bytes/entry)
NHLFE Counters (SRAM) ^{1*}	1,5 MB (64K entries, 24 bytes/entry)
Total	3,5 MB - 2 MB²

1. Can be compiled out of code
2. Without statistics

Table 35-10. Code Store Footprint

Code Store	Size (words)
FTN Forwarder code without statistics	124
FTN Forwarder statistics code	8
Total	132

35.5.10 FTN Forwarder Characterization Data

Table 35-11. FTN Forwarder Microblock Characterization Data

Data	Value
General:	
Microblock Name	MPLS_FTN_UC
Microblock Version Number	1.1
Implementation Language	microcode
Configuration Options use to gather this set of data	FTN_SRAM, FTN_NON_DIFFSERV
Measurement Environment (tool settings)	
Performance: (cycle counts, latencies, bandwidths)	
Instruction cycle count per packet (common case packet / worst case cycle count)	66/71
Common-case packet/path assumptions to be documented here	<p>Assumptions: Operate within the same operational environment as IPv4 Forwarder.</p> <p>Plus: 1. Packet header (or portion of packet header) is cached in local memory.</p> <p>2. NextHopID_Type matches with that of MPLS.</p> <p>3. Index into FTN NHLFE table is set in NextHopID field of packet metadata.</p>

Table 35-11. FTN Forwarder Microblock Characterization Data (Continued)

Data	Value
Scratch Memory	
# of longwords read (for bandwidth calculations)	0
# of longwords written (for bandwidth calculations)	0
# and type of each atomic operation performed (for bandwidth calculations)	0
SRAM	
# of longwords read	6
# of longwords written	0
# and type of each atomic operation performed (for bandwidth calculations)	0
Co-processors	
bytes read (per channel)	
bytes written (per channel)	
DRAM	
# of quadwords read	0
# of quadwords written	0
Other I/O instructions (i.e. MSF, PCI, CAP, and Hash)	0
List of dependent I/O accesses in the longest latency path	Read 6 LWs from SRAM memory

Per-Microengine Resources:

Control-Store Usage (# of instructions used)	191
Local Memory Footprint (# of long words used)	128
Local Memory Configuration (shared, or per-context pointer)	per-context
Local Memory - # of LM pointers used	1 (LM pointer0)
GPR Usage – minimum, static usage (absolute, static, globals)	6 (statics), 5 (absolute)
Transfer Reg. Usage – minimum, static usage	7 SRAM
Next Neighbor Reg. Usage – minimum, static usage	0
Signal Usage – minimum, static usage	0
CAM used? (yes or no)	no

Global Resources:

Scratch footprint (# of longwords used) – constant or formula ...	0
SRAM footprint (# of longwords used) – constant or formula ...	2MB FTN NHLFE (64KB entries, 32B per entry). 1.5MB counters (64KB entries, 24B per entry)
DRAM footprint (# of quadwords used) – constant or formula ...	0

Table 35-11. FTN Forwarder Microblock Characterization Data (Continued)

Data	Value
Q-Array usage - # of queues used and if they need to be cached	0
CRC Unit used?	0
Hash Unit used? (yes or no)	no
MSF Usage Information:	
Media Bus Configuration	-
RBUF, TBUF usage	-
CBus signals	-
Other Information:	
Critical Section Length (compute cycles + memory accesses)	0
# of phases	1
Packet Metadata - fields read	0
Packet Metadata - fields written	0
Header - fields read	0
Header - fields written	0
Documentation:	
Thread Ordering Requirements	none
OS dependencies	VxWorks, Linux
Chip/hardware dependencies (i.e. Crypto Unit in 2850)	2800, 2850
Tested on which SDK Release(s)	PR-6
Tested on hardware? Which hardware configuration?	Yes, IXDP 2400, IXDP 2800
Tested in which applications (not an all inclusive list)	Several IXA SDK 3.1 applications
Possible Configuration Options	
Major limitations (i.e. Rx/Tx microblocks written to use entire ME)	Support up to 4 MPLS labels
Packet Sequencing Issues (esp. in POT applications)	none
Core Component or Interface requirements or dependencies	MPLS FTN CC, Scripts to initialize FTN NHLFE table

36.1 Overview

The ILM Forwarder microblock implements the functionality of the MPLS Transit Label Switched Router (LSR) and Exit Label Edge Router (LER) nodes.

36.2 Functionality

The ILM Forwarder microblock receives MPLS-labeled packets with their meta data from the Dl_Source microblock. The ILM Forwarder processes only packets that have the dl_header_type dispatch loop variable set to the value of MPLS_TYPE(6).

First, the ILM Forwarder checks whether the top label on the packet's label stack belongs to the reserved label range <0, 15>. The reserved labels are:

- A value of 0.

It represents the "IPv4 Explicit NULL Label". This label value is only legal at the bottom of the label stack. It indicates that the label stack must be popped, and the forwarding of the packet must then be based on the IPv4 header.

The ILM Forwarder checks if the stack bit of the MPLS header is toggled on. If true, it reads a special ILM entry for label 0 and continues further packet processing. Otherwise, the packet is dropped.

- A value of 1.

It represents the "Router Alert Label". This value is legal anywhere in the label stack except at the bottom. When a received packet contains this label value at the top of the label stack, a packet is delivered to a local software module for processing.

The ILM Forwarder checks if the stack bit of an MPLS header is toggled off. If true, the MPLS packet with "Route alert option" (value 1) is sent to the MPLS core component as an exception. Otherwise, a packet is dropped.

- A value of 2.

It represents the "IPv6 Explicit NULL Label". This label value is only legal at the bottom of the label stack. It indicates that the label stack must be popped, and the forwarding of the packet must then be based on the IPv6 header.

The ILM forwarder checks if the stack bit of an MPLS header is toggled on. If true, it reads a special ILM entry for label 2 and continues further packet processing. Otherwise, the packet is dropped.

- A value of 3 to 15.

These are reserved values. Packets with such labels are dropped.

If the packet's topmost label is not one of the reserved range, the ILM Forwarder performs a lookup in the ILM table using the label value as an index. Each MPLS label is mapped to one or more NHLFE entries. For efficiency reasons the ILM table is actually combined with NHLFE table so

that its entries contain all information needed for packet forwarding, such as DiffServ flags for the LSP, the code of operation to be performed on the label stack, the outgoing label stack for the packet, and the next hop information.

An ILM entry can be either a regular entry or an NHLFE set. In the latter case, one element of the set is chosen. The algorithm used to determine an NHLFE from a set can implement load sharing or be based on DiffServ information for the packet. In this version of the ILM Forwarder the first element of the set is taken.

After the ILM lookup, the optional and LSP (input segment) statistics are updated. The next steps of packet's processing depend on the label stack operation specified in the ILM entry. The operation on the label stack can be one of the following:

- POP—strips off the top label from the label stack and, if the stack is not empty, perform another lookup in the ILM table, otherwise pass the packet to the IP forwarder for further processing,
- SWAP—replaces the label from the top of the stack with another label, specified in the ILM entry, and forward the packet to the next hop specified in the this entry,
- SWAP_PUSH—replaces the label from the top of the stack with another label, push one or more additional labels on the label stack, and forward the packet to the next hop specified in the ILM entry,
- POP_FORWARD—strips off the top label from the label stack and forward the packet to the next hop specified in the ILM entry - this action implements so called penultimate hop popping.

The ILM Forwarder performs the required operation on the packet's label stack, updates the packet's meta data, and passes the packet either to the IPv4 Forwarder (if the operation was POP and the last label was removed from the label stack), or to the Queue Manager for sending to the output interface.

36.3 Assumptions and Dependencies

36.3.1 Assumptions

It is assumed that the ILM Forwarder operates within the same operational environment as IPv4 forwarder. This operational environment specifies:

- Usage of hardware resources
 - Mapping to microengine
 - Usage of scratch rings
 - Usage of DRAM and SRAM
- Format of entries in scratch rings
 - Receive scratch ring from POS RX microblock
 - Transmit scratch ring to Ingress Queue Manager and Scheduler
 - Meta data exchanged with proceeding and following microblocks
 - Format of the meta data prepended to the data buffer, while traveling through the CSIX fabric to an Egress Network Processor.

36.3.2 Component Dependencies

On input, the ILM Forwarder microblock depends on the DI_Source microblock that should recognize the incoming packet as an MPLS packet (by examining the protocol type field of the packet's meta-data, set by the Rx microblock), and set the dl_header_type dispatch loop variable to the value of MPLS_TYPE(6), and the dl_next_block variable to the ILM Forwarder block ID (BID_MPLSILM).

If the MPLS node does not perform penultimate hop popping, the ILM Forwarder on an egress LER requires that the IPv4 Forwarder microblock is present, because in this case IP packets obtained after popping the last label from the label stack are subject to normal IP forwarding.

36.4 Microblock Interfaces

36.4.1 Input Microblock Variables

This section lists all variables read by the ILM Forwarder block.

Table 36-1. Input Variables Consumed by ILM Forwarder

Input variable	Size	Type ¹	Description
dl_next_block	8 bits	M	Set by DI_Source microblock. It should be equal to BID_MPLSILM. Otherwise, the ILM Classifier block executes an empty bypass path.
dl_header_type	8 bits	M	Incoming packet type, should be: MPLS_TYPE(6)
dl_packet_size	16 bits	M	Number of bytes in the currently processed packet.
dl_packet_offset	16 bits	M	Offset to the beginning of the packet in the buffer.
dl_input_port	16 bits	M	Logical port number on which the packet was received.

1. M and S letters in the Type column mean, respectively, that this variable is part of the packet metadata and is transported over switch fabric to the egress NP.

36.4.2 Output Microblock Variables

This section lists all variables modified and set by the ILM Forwarder block.

Table 36-2. Output Variables Modified by ILM Forwarder

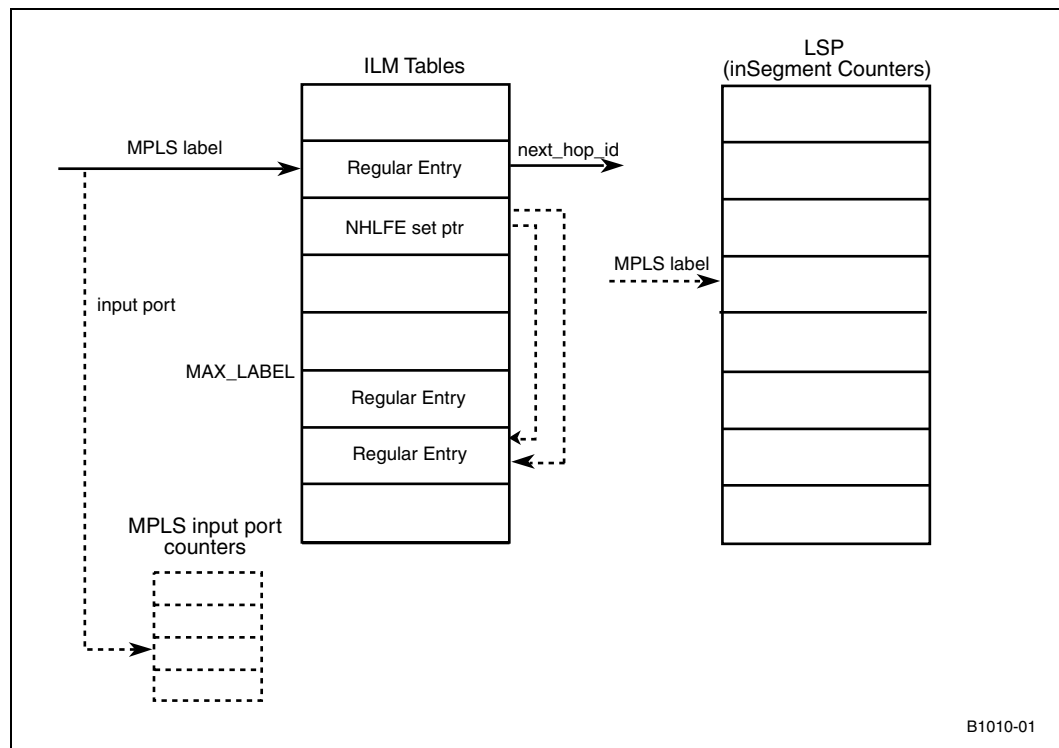
Input variable	Size	Type ¹	Description
dl_next_block	8 bits	M	Specifies the output of the ILM Forwarder, which can be one of the following: <ul style="list-style-type: none"> IX_DROP—the packet should be dropped IX_EXCEPTION—the packet should be passed to the MPLS core component. In this case the variable dl_exception_code specifies further details about the exception MPLSILM_NEXT1—the packet should be passed to the Queue Manager MPLSILM_NEXT2—the packet should be passed to the IPv4 forwarder for further processing
dl_exception_code	8 bits	M	If dl_next_block is equal to IX_EXCEPTION, it can be one of the following: <ul style="list-style-type: none"> ROUTER_ALERT - an MPLS packet with the router alert label has been detected (label value 1) TTL_EXPIRED—the packet's Time-to-live (TTL) value dropped to zero TOO_MANY_POPS—after popping 3 labels from the packet's label stack, the next ILM entry indicates the POP operation (such a packet should be forwarded by the slow path to avoid excessive performance degradation). LABELED_PACKET_TOO_BIG—the MPLS-labeled packet's length exceeds the "Effective Maximum Frame Payload Size for Labeled Packets" (maxLabPktSize) parameter of the output port
dl_exception_id	8 bits	M	If dl_next_block is equal to IX_EXCEPTION, this field is set to the ILM Forwarder microblock ID (BID_MPLSILM)
dl_nexthop_id_type	4 bits	M	Indicates the dl_next_hop_id contents:NHID_IP(0) - index into L2 table
dl_next_hop_id	16 bits	S	In case of normal forwarding, indicates an entry in the egress L2 table that should be used to perform an appropriate L2 encapsulation of the data packet.In case of passing the packet to the IPv4 Forwarder set to(-1)
dl_header_type	8 bits	M	Ongoing packet type, one of:IPV4_TYPE(4) MPLS_TYPE(6)
dl_packet_size	16 bits	S	Number of bytes in the currently processed packet.
dl_packet_offset	16 bits	S	Offset to the beginning of the packet in the buffer.

1. M and S letters in the Type column mean, respectively, that this variable is part of the packet metadata and is transported over switch fabric to the egress NP

36.5 Data Structures and Forwarding Algorithm

36.5.1 Overview

The ILM Forwarder main data structures and their interconnections are shown in Figure 37-1.

Figure 36-1. ILM Forwarder Data Structures


The ILM table contains all information needed for incoming MPLS packets forwarding. It is a linear array indexed by the incoming label value. The control plane determines the size of the ILM table by specifying the minimum and maximum allowable label value. The ILM table can be shared across all input ports in case of per-platform label space, or can be private for each port in case of per-interface label space.

The ILM table comprises regular ILM entries and ILM_NHLFE sets, which elements point to up to four regular ILM entries assigned to one label value and placed above the "MAX_LABEL" entry. The algorithm used to choose one entry from a set can implement load sharing or be based on DiffServ information for the packet.

Regular ILM table entries contain all information needed for packet forwarding, such as DiffServ flags for the LSP, the code of operation to be performed on the label stack, the outgoing label stack for the packet, and the next hop information.

The inSegment counters are per-ILM entry structures accumulating various statistics relevant to MPLS packet forwarding. They are separated from the ILM table for efficiency reasons - they can be compiled out of the code. The incoming statistics for a given LSP are complemented by some delta counters to facilitate calculating the outgoing LSP statistics based on the inSegment counter values.

The InPort Counters table gathers MPLS-related per-port statistics. It is a linear table indexed by the input port number value. It is compiled conditionally. Note that the ILM Forwarder does not gather MPLS statistics for output ports - they will be maintained by the Layer 2 Encapsulation microblock.

36.5.2 ILM Table

The ILM table is a linear table indexed by the MPLS label value. It is located in DRAM or SRAM, based on a compilation constant. [Table 36-3](#) shows the layout of the ILM table entry.

Table 36-3. ILM Table Entry

LW	Bits	Size	Field	Description
0	31:24	8	flags	bit 31: MPLS_ILM_VALID_BIT - if set, the entry is valid, otherwise the entry is unallocated bit 30: - MPLS_ILM_NHLFE_SET_BIT - this entry points to an NHLFE setbits 29:28 - MPLS_ILM_TUNNEL_MODEL: <ul style="list-style-type: none"> • 00—Pipe model • 01—Uniform model • 10—Short Pipe model without PHP • 11—Short Pipe model with PHP bits 27:26 - MPLS_ILM_LSP_TYPE -LSP type: <ul style="list-style-type: none"> • 00—non-DiffServ LSP • 01—E-LSP • 11—L-LSP bit 25: MPLS_ILM_TC_ENABLED_BIT—traffic conditioning enabled bit 24: MPLS_ILM_PRECONFIGURED_EXP_MARKING—currently not used
0	23:22	2	operation	The code of operation to be performed on the label stack, one of: <ul style="list-style-type: none"> • IX_MPLS_LABELOP_SWAP = 0 • IX_MPLS_LABELOP_SWAP_PUSH = 1 • IX_MPLS_LABELOP_POP = 2 • IX_MPLS_LABELOP_POP_FORWARD = 3
0	21:19	3	pushCount	Number of labels to be added to the label stack, from 1 to 4
0	18:16	3	l3Protocol	The L3 protocol type of the MPLS payload: <ul style="list-style-type: none"> • 0—UNKNOWN • 6—IP_v4 • 7—IP_v6
0	15:0	16	nextHopID	ID used on the Egress Sausalito to lookup the outgoing link layer information
1	31:16	16	maxLabPktSize	Effective Maximum Frame Payload Size for labeled packets (LSP MTU), for the outPortID port
1	15:0	16	outPortID	Outgoing port ID - local port index on outgoing blade
2	31:12	20	topLabel	Top of stack label - either mpls shim or atm label
2	11:8	4	reserved	Set to 0
2	7:4	4	outPortType	Outgoing port type
2	3:0	4	ttl	TTL field decrement value
3	31:12	20	label1	Label to be pushed below the topLabel, if the pushCount field value is greater than 1
3	11:8	4	reserved	Set to 0
3	7:0	8	bladeID	Outgoing blade ID w.r.t. CSIX fabric

Table 36-3. ILM Table Entry (Continued)

LW	Bits	Size	Field	Description
4	31:12	20	label2	Label to be pushed below Label1, if the pushCount field value is greater than 2
4	11:8	4	reserved	Set to 0
4	7:0	8	flowIdHigh	MSB 8 bits of 20 bit flow id (not used in this release)
5	31:12	20	label3	Label to be pushed below Label2, if the pushCount field value field equals to 4
5	11:0	12	flowIdLow	LSB 12 bits of flow id
6	31:16	16	Exp2phb	Index into the Exp2Phb table (not used in this release)
6	15:0	16	Phb2Exp	Index into the Exp2Phb table (not used in this release)
7	31:0	32	reserved	Set to 0

The size of memory needed for the ILM table on each blade can be calculated as:

$$(\text{max_number_of_incoming_LSPs} * 8) * 4 \text{ bytes}$$

or as in case of private ILM tables for each input port.

$$(\text{num_of_input_ports} * \text{max_number_of_incoming_LSPs} * 8) * 4 \text{ bytes}$$

36.5.3 ILM_NHLFE Set

Table 36-4 shows the layout of the ILM_NHLFE set entry.

Table 36-4. ILM_NHLFE Set Entry

LW	Bits	Size	Field	Description
0	31:24	8	flags	Bits 31:24 hold the following bit flags: <ul style="list-style-type: none"> bit 31: MPLS_NHLFE_VALID_BIT—if set, the entry is valid, otherwise the entry is unallocated and the whole flags byte should be equal to 0 bit 30: MPLS_NHLFE_SET_BIT—entry type: regular ILM if 0, ILM_NHLFE set if 1 Bits 30:24—Reserved(0)
0	23:8	16	exp2phb	Index into the Exp2Phb table (not used in this release)
0	7:0	8	reserved	Set to 0
1	31:0	32	phbMask1	Criterion used by the ILM forwarder for choosing one of the set elements.
2	31:0	32	phbMask2	Criterion used by the ILM forwarder for choosing one of the set elements.
3	31:0	32	phbMask3	Criterion used by the ILM forwarder for choosing one of the set elements.
4	31:0	32	phbMask4	Criterion used by the ILM forwarder for choosing one of the set elements.
5	31:12	20	outSegID1	Index to the 1st NHLFE belonging to the set
5	11:0	12	outSegID2	Index to the 2nd NHLFE belonging to the set (12 most significant bits)

Table 36-4. ILM_NHLFE Set Entry (Continued)

LW	Bits	Size	Field	Description
6	31:24	8	outSegID2	Index to the 2nd NHLFE belonging to the set (8 least significant bits)
6	23:4	20	outSegID3	Index to the 3rd NHLFE belonging to the set
6	3:0	4	outSegID4	Index to the 4th NHLFE belonging to the set (4 most significant bits)
7	31:16	16	outSegID4	Index to the 4th NHLFE belonging to the set (16 most significant bits)
7	15:0	16	reserved	Set to 0

36.5.4 Input Segment Counters

The input segment counters is a conditionally compiled table of per-ILM entry structures accumulating various statistics relevant to MPLS packet forwarding. Table 36-5 shows the layout of the statistics block.

Table 36-5. InSegment Counters Table Entry

LW	Bits	Size	Field	Description
0	31:0	32	mplsInSegmentOctets	32-bits counter accumulating the number of octets received by this segment.
1	31:0	32	mplsInSegmentPkts	32-bits counter accumulating the number of packets received by this segment
2	31:0	32	mplsInSegErrorOctets	32-bits counter accumulating the number of erroneous octets received by this segment (e.g. with TTL expired, missed lookups, fragmentation errors)
3	31:0	32	mplsInSegErrorPkts	32-bits counter accumulating the number of erroneous packets received by this segment (e.g. with TTL expired, missed lookups, fragmentation errors)
4	31:0	32	mplsInSegFragmOctDelta	32-bits counter accumulating the number of octets being the difference between the sum of fragments lengths and the fragmented packet's length
5	31:0	32	mplsInSegFragmPktDelta	32-bits counter accumulating the number of fragments sent on this segment, excluding the first fragment

The size of memory needed for the Input Segment Counters table on each blade can be calculated as:

$$(\text{number_of_incoming_LSPs} * 6) * 4 \text{ bytes}$$

36.5.5 InPort Counters

The InPort Counters table is a linear table indexed by the input port number values. It is placed in SRAM. The counters having 64-bit counterparts are grouped at the end to simplify their implementation by the MPLS Ports core component, and, alternatively, to be compiled conditionally. Table 36-6 shows the layout of the InPort Counters table.

Table 36-6. Input Port Counters Table

LW	Bits	Size	Field	Description
0	31:0	32	mplsInlfcRxPkts	32-bit counter accumulating the number of MPLS labeled packets received on this interface
1	31:0	32	mplsInlfcFailedLookupPkts	32-bit counter accumulating the number packets that were discarded due to failed MPLS label lookup on this interface
2	31:0	32	mplsInlfcErrorPkts	32-bit counter accumulating the number packets that were discarded due to errors other than failed MPLS label lookup on this interface
3	31:0	32	mplsInlfcRxOctets	32-bit counter accumulating the number of octets received in MPLS labeled packets on this interface. For efficiency reasons, the 64-bit version of this counter should be maintained by the MPLS Core Component.

The size of memory needed for the Input Port Counters table on each blade can be calculated as:

$$(\text{number_of_ports_on_this_blade} * 4) * 4 \text{ bytes}$$

36.5.6 Forwarding Algorithm

This section presents the diagrams of the ILM Forwarder microblock forwarding algorithm.

Figure 36-2. ILM Forwarder Algorithm - Phase 1

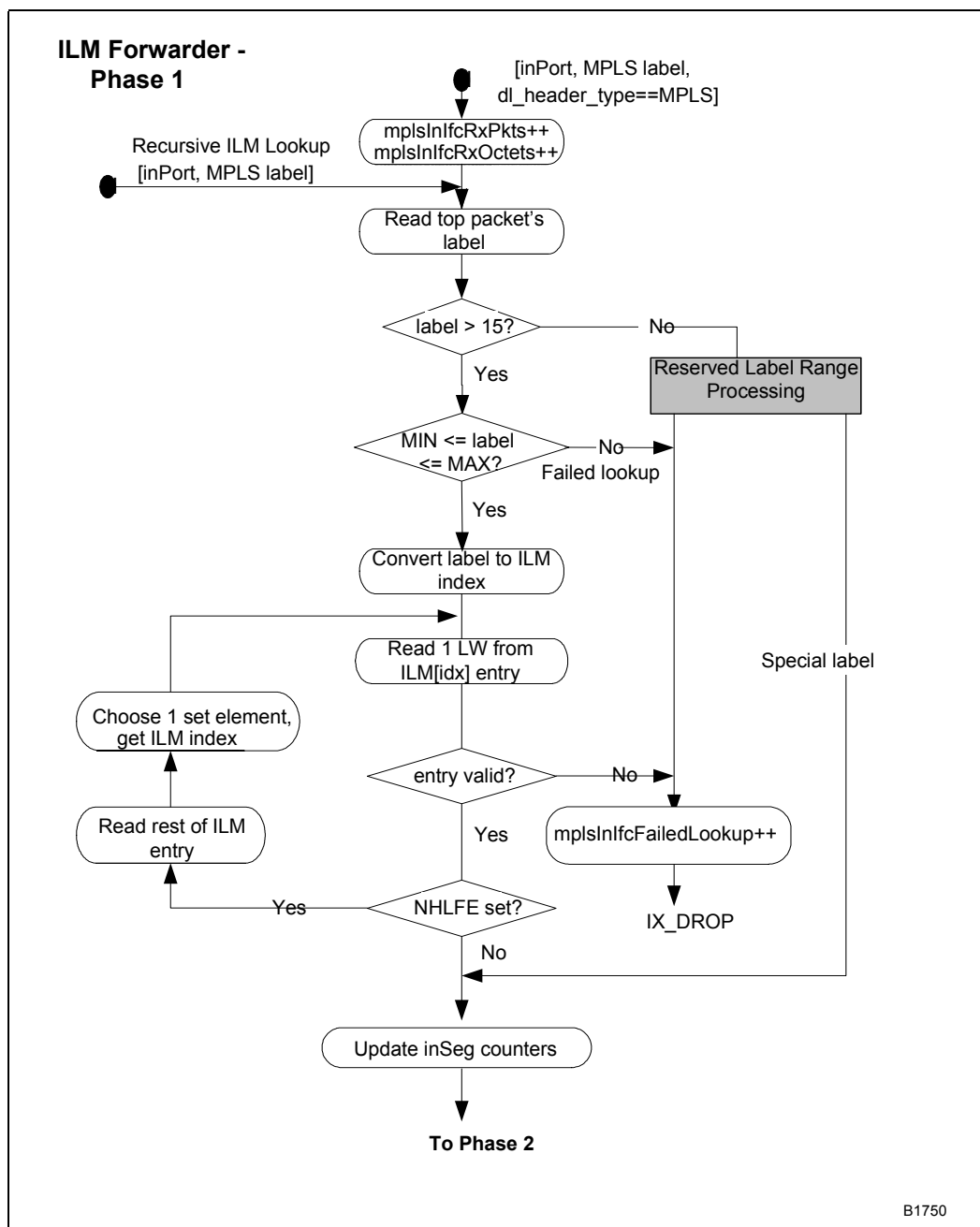


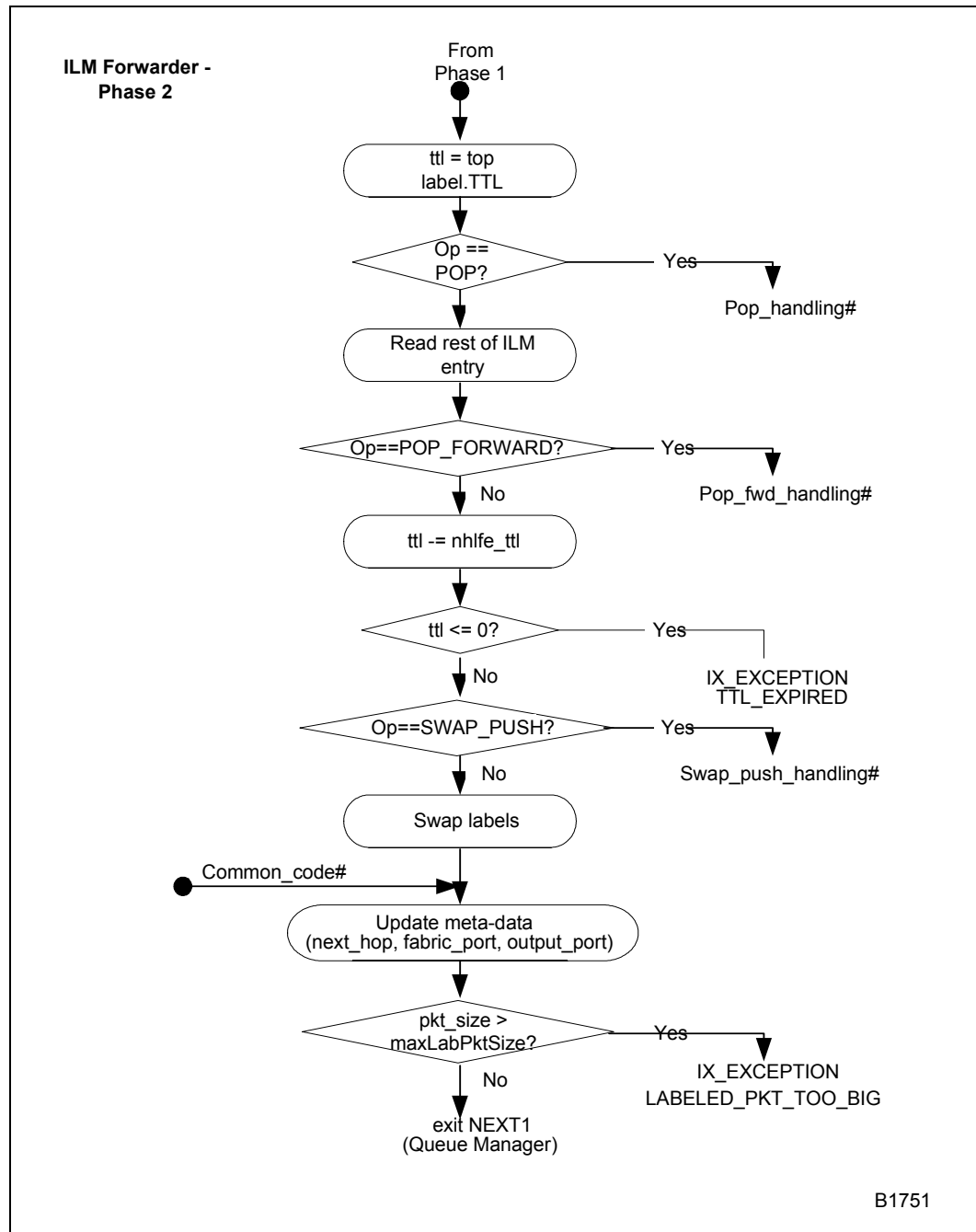
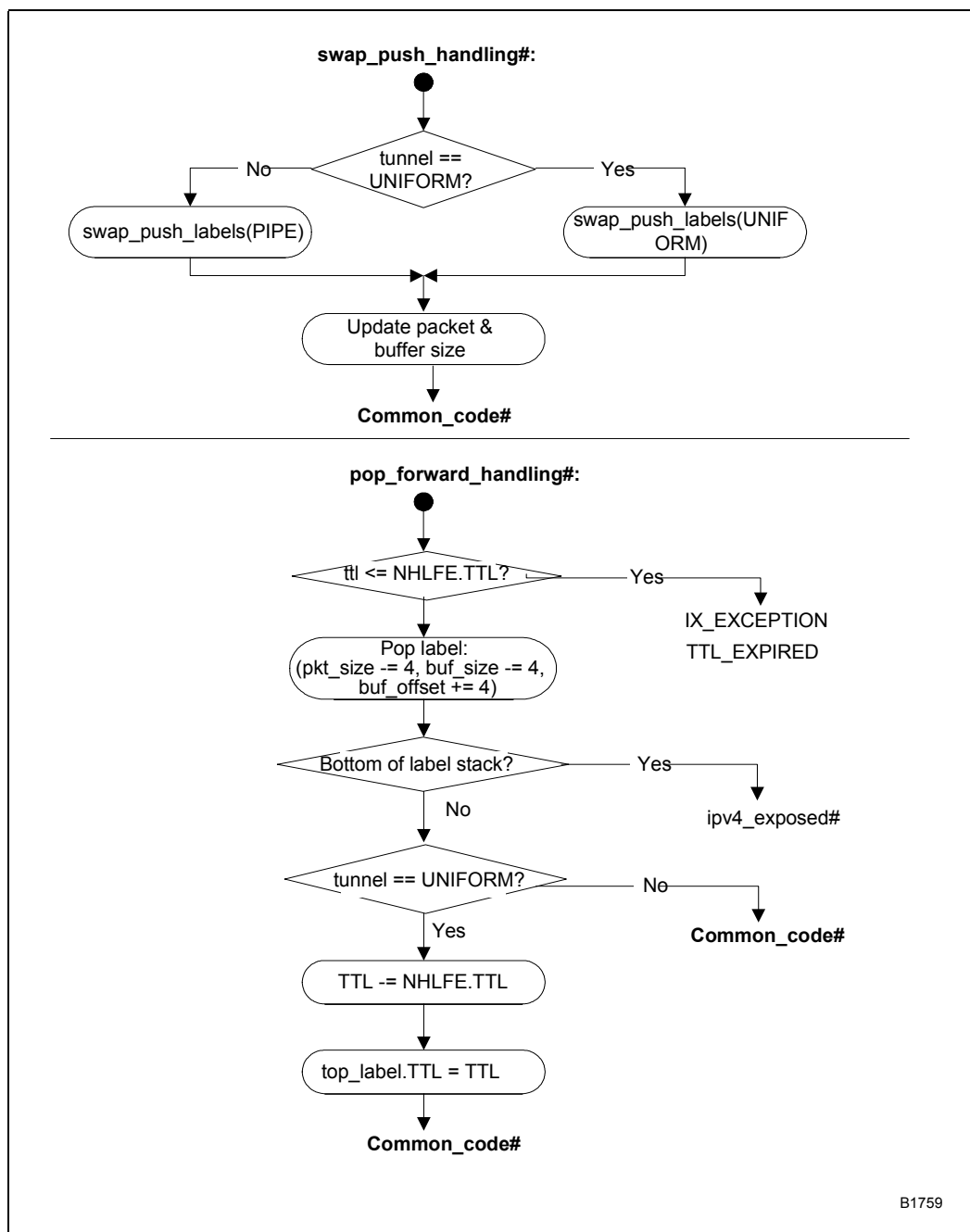
Figure 36-3. ILM Forwarder Algorithm - Phase 2 (Page 1 of 4)


Figure 36-4. ILM Forwarder Algorithm - Phase 2 (Page2 of 4)



B1759

Figure 36-5. ILM Forwarder Algorithm - Phase 2 (Page 3 of 4)

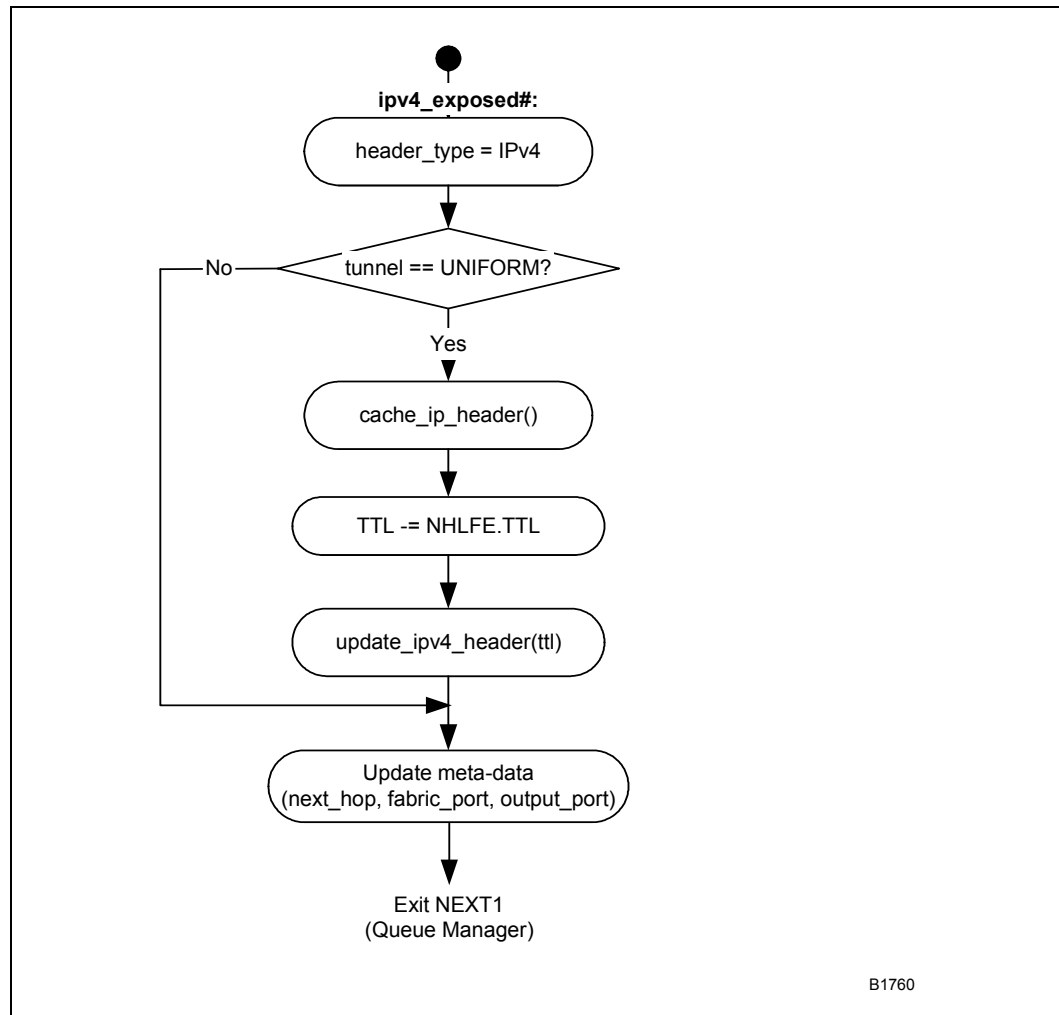


Figure 36-6. ILM Forwarder Algorithm - Phase 2 (Page 4 of 4)

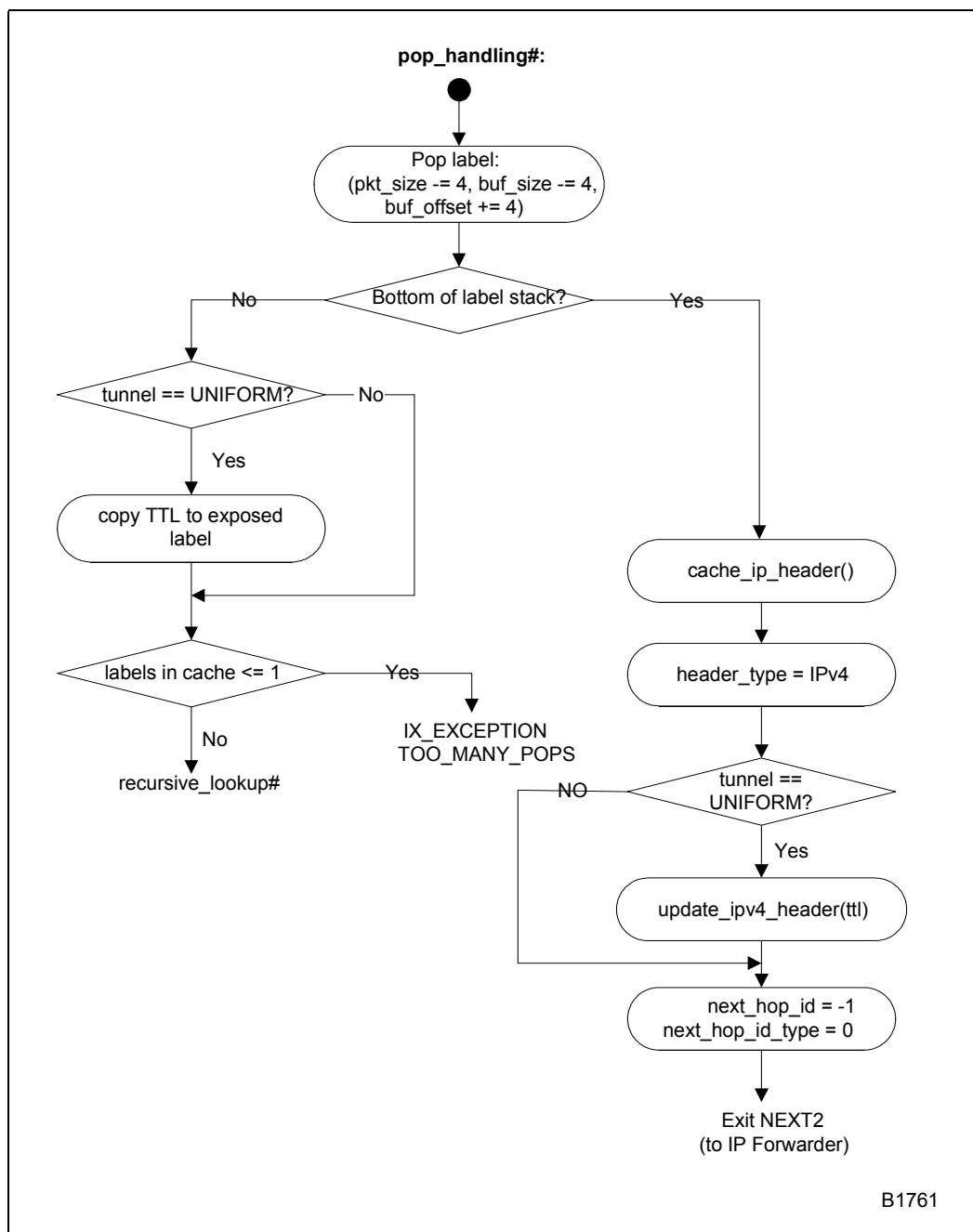
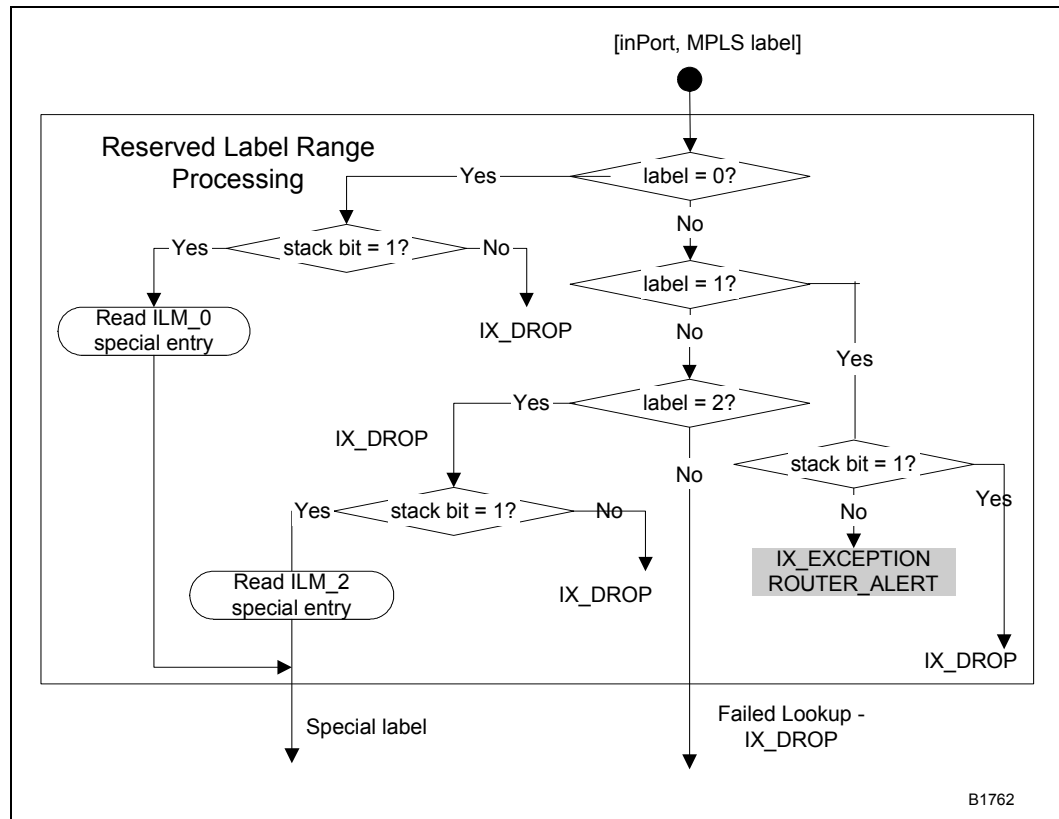


Figure 36-7. ILM Reserved Label Range Processing



36.5.7 Allocation to Microengines

See Section 35.5.7, “Allocation to Microengines” on page 615.

36.5.8 Thread Ordering and Synchronization

See Section 35.5.8, “Thread Ordering and Synchronization” on page 616.

36.5.9 ILM Forwarder Micro-code Budget

36.5.9.1 Performance Analysis

The following data reflects the main forwarding path of the current version of code prototyping:

Table 36-7. Cycle Count Table (including unfilled defers) for Operation SWAP

Phase	SWAP	
	PER PLATFORM LABEL SPACE	PER INTERFACE LABEL SPACE
ILM Phase 1 processing	28	32
ILM Phase 2 processing	28	28
Statistics overhead	17	16
Total	56 (73)	60 (76)

Table 36-8. Cycle Count Table (including unfilled defers) for Operation POP

Phase	POP one label and SWAP		POP three labels and SWAP	
	PER PLATFORM LABEL SPACE	PER INTERFACE LABEL SPACE	PER PLATFORM LABEL SPACE	PER INTERFACE LABEL SPACE
ILM Phase 1 processing	$28 + 18 = 46$	$32 + 17 = 49$	$28 + 3 \times 18 = 82$	$32 + 3 \times 17 = 83$
ILM Phase 2 processing	$26 + 28 = 54$	$26 + 28 = 54$	$3 \times 26 + 28 = 106$	$3 \times 26 + 28 = 106$
Statistics overhead	$8 + 2 \times 9 = 26$	$7 + 2 \times 9 = 25$	$8 + 4 \times 9 = 44$	$7 + 4 \times 9 = 43$
Total	100 (126)	103 (128)	188 (232)	189 (232)

Table 36-9. Cycle Count Table (including unfilled defers) for Operation POP_FORWARD

Phase	POP one label and FORWARD MPLS packet		POP one label and FORWARD IP packet	
	PER PLATFORM LABEL SPACE	PER INTERFACE LABEL SPACE	PER PLATFORM LABEL SPACE	PER INTERFACE LABEL SPACE
ILM Phase 1 processing	28	32	28	32
ILM Phase 2 processing	43	43	50 - 77	50 - 77
IP header update	--	--	10	10
Statistics overhead	17	16	17	16
Total	71 (88)	75 (91)	78 (95) - 115 (132)	82 (98) - 119 (135)

Table 36-10. Cycle Count Table (including unfilled defers) for Operation SWAP_PUSH

Phase	SWAP labels and PUSH one label onto stack		SWAP labels and PUSH three labels onto stack	
	PER PLATFORM LABEL SPACE	PER INTERFACE LABEL SPACE	PER PLATFORM LABEL SPACE	PER INTERFACE LABEL SPACE
ILM Phase 1 processing	28	32	28	32
ILM Phase 2 processing	55 - 57	55 - 57	66 - 68	66 - 68
Statistics overhead	17	16	17	16
Total	83(100) - 85(102)	87 (103) - 89 (105)	94 (111) - 96 (113)	98(114) - 100(116)

Table 36-11. I/O Latency Analysis Table

Operation	I/O type	Accesses	Transfer size (in 32-bit long words)
ILM entry read	SRAM or DRAM read	1 - 4	1 - 2 (POP)8 (other operations)
Cache IP header	DRAM	0 - 1	6 - 8
Input port statistics update ¹	SRAM increment and SRAM add	2	1
InSegment statistics update ^{1*}	SRAM increment and SRAM add	2	1

1. Can be compiled out of code

36.5.9.2 Memory Footprint Analysis

Table 36-12. Data Structures Footprint

Data structures	Size (bytes)
Local Memory Cache	512 (shared with FTN Forwarder) + number of input ports * 16 bytes
ILM Table (SRAM or DRAM)	2 MB (64 K entries * 32 bytes/entry)
InSegment statistics (SRAM) ¹	1,5 MB (64 K entries * 24 bytes/entry)
Port statistics (SRAM)	16 bytes per input port
Total	3,5 MB or 2 MB ²

1. Can be compiled out of code
2. Without statistics

Table 36-13. Code Store Footprint

Code Store	Size (words)	
	PER PLATFORM LABEL SPACE	PER INTRFACE LABEL SPACE
ILM Forwarder code without statistics	306	310

Table 36-13. Code Store Footprint

Code Store	Size (words)	
ILM Forwarder statistics code	29	28
Total	335	338

36.5.10 ILM Forwarder Microblock Characterization Data

Table 36-14. ILM Forwarder Microblock Characterization Data

Data	Value
General:	
Microblock Name	MPLS_ILM_UC
Microblock Version Number	1.16
Implementation Language	microcode
Configuration Options use to gather this set of data	ILM_SRAM
Measurement Environment (tool settings)	
Performance: (cycle counts, latencies, bandwidths)	
Instruction cycle count per packet (common case packet / worst case cycle count)	46 (SWAP) / 232 (POP4)
Common-case packet/path assumptions to be documented here	Assumptions: Operate within the same operational environment as IPv4 Forwarder. Plus: Packet header (or portion of packet header) is cached in local memory.
Scratch Memory	
# of longwords read (for bandwidth calculations)	0
# of longwords written (for bandwidth calculations)	0
# and type of each atomic operation performed (for bandwidth calculations)	0
SRAM	
# of longwords read	8 (POP4 case)
# of longwords written	0
# and type of each atomic operation performed (for bandwidth calculations)	0
Co-processors	
bytes read (per channel)	
bytes written (per channel)	
DRAM	
# of quadwords read	3 (POP cases)
# of quadwords written	0
Other I/O instructions (i.e. MSF, PCI, CAP, and Hash)	0

Table 36-14. ILM Forwarder Microblock Characterization Data (Continued)

Data	Value
List of dependent I/O accesses in the longest latency path	Read 5 LWs from SRAM memory
Per-Microengine Resources:	
Control-Store Usage (# of instructions used)	528
Local Memory Footprint (# of long words used)	128 + (Number_of_interface * 3 LWs)
Local Memory Configuration (shared, or per-context pointer)	per-context
Local Memory - # of LM pointers used	1 (LM pointer0)
GPR Usage – minimum, static usage (absolutes, static, globals)	12 (statics), 5 (absolutes)
Transfer Reg. Usage – minimum, static usage	7 SRAM
Next Neighbor Reg. Usage – minimum, static usage	0
Signal Usage – minimum, static usage	2
CAM used? (yes or no)	no
Global Resources:	
Scratch footprint (# of longwords used) – constant or formula ...	0
SRAM footprint (# of longwords used) – constant or formula ...	2MB ILM NHLFE (64KB entries, 32B per entry). 1.5MB counters (64KB entries, 24B per entry)
DRAM footprint (# of quadwords used) – constant or formula ...	0
Q-Array usage - # of queues used and if they need to be cached	0
CRC Unit used?	0
Hash Unit used? (yes or no)	no
MSF Usage Information:	
Media Bus Configuration	-
RBUF, TBUF usage	-
CBus signals	-
Other Information:	
Critical Section Length (compute cycles + memory accesses)	0
# of phases	1
Packet Metadata - fields read	0
Packet Metadata - fields written	0

Table 36-14. ILM Forwarder Microblock Characterization Data (Continued)

Data	Value
Header - fields read	Read next 3 QWs from packet header and cache to local memory in order to update IPv4 TTL and checksum values (POP/POP_FORWARD exposing IPv4 header)
Header - fields written	0
Documentation:	
Thread Ordering Requirements	none
OS dependencies	VxWorks, Linux
Chip/hardware dependencies (i.e. Crypto Unit in 2850)	2800, 2850
Tested on which SDK Release(s)	PR-6
Tested on hardware? Which hardware configuration?	Yes, IXDP 2400, IXDP 2800
Tested in which applications (not an all inclusive list)	Several IXA SDK 3.1 applications
Possible Configuration Options	
Major limitations (i.e. Rx/Tx microblocks written to use entire ME)	Support up to 4 MPLS labels
Packet Sequencing Issues (esp. in POT applications)	none
Core Component or Interface requirements or dependencies	MPLS ILM CC, Scripts to initialize ILM NHLFE table

Services Microblocks

The Services microblocks include the following:

- [Chapter 37, “Packet Copier Microblock”](#)
- [Chapter 38, “Freelist Manager”](#)

Microblock	Description	Usage	Cycle Budget
Packet Copier	Creates specified number of copies of a packet.	Runs on 2 microengines for OC-192 data rates. Can run on 1 microengine for lower rates.	57 cycles for OC-192 POS
Freelist Manager	Maintains a packet buffer freelist in local memory.	Runs on 1 microengine.	97 for OC-48 POS, 101 for 4Gb Ethernet, 57 for OC-192 POS, 94 for 10Gb Ethernet

The purpose of the Packet Copier microblock is to create required number of copies from a packet. This block is agnostic to type type of packet it is copying, whether it is unicast, multicast, or broadcast. This block can operate in both cell mode and packet mode.

This chapter describes the packet mode version of the microblock. Cell mode operation will be described in future revisions of this chapter.

This microblock receives requests to create copies, and sends the copies back to the requesting pipeline. This block makes use of an additional buffer freelist which contains child buffers.

37.1 Overview

This microblock is designed as a driver block. This microblock can be configured to run on fewer threads for low data rate applications, thus enabling other microblocks to share the same microengine. There is no Core Component for this microblock. This block doesn't maintain any shared data structures in external memory.

The microblock requires the configuration information listed in [Table 37-1](#) at compile time.

Table 37-1. Packet Copier Microblock Build Switches

Build Switch	Description
PACKET_MODE [†]	The microblock works in packet mode.
CELL_MODE [†]	The microblock works in cell mode.
PKT_COPIER_2ME	Enables this block to run as context pipe stage in 2 microengines to support OC-192 data rates.
PKT_COPIER_1ST_ME	Executes 1 st microengine function in 2 ME pipe stage. Requires PKT_COPIER_2ME defined.
PKT_COPIER_2ND_ME	Executes 2 nd microengine function in 2 ME pipe stage. Requires PKT_COPIER_2ME defined.
PKT_COPIER_SPHY	Sets the output port to zero in child meta data. If not defined, then port number is set for each copy according to the bit mask.
[†] Both operating modes are mutually exclusive.	

The microblock requires the configuration information listed in [Table 37-2](#) at load time. Alternately, this information can be provided as compile-time information. This is in addition to parent buffer information which is available as part of the framework.

Table 37-2. Packet Copier Microblock Symbols

Symbols	Description
CHILD_SRAM_META_BASE	Child meta data start address in SRAM. This value encodes the SRAM channel in bits 31:30.
CHILD_SRAM_META_BASE_SPLIT	Child meta data start address in SRAM in case meta data is split across two channels. This value encodes the SRAM channel in bits 31:30.
CHILD_FREELIST_ID	Buffer freelist number to fetch child buffers.

This microblock maintains packet order across requests. This means that all copies of one multicast packet are processed before processing copies from the next multicast packet.

This microblock assumes that the child buffer freelist is initialized a priori. Also, this block doesn't drop any created packets.

37.2 Requirements

The Packet Copier microblock is designed to address the following requirements:

- Zero copy of packet data.
- Ordering has to be preserved across multicast packets.

Note: The current version of the microblock supports 16 ports. Future versions will support a larger number of ports.

37.3 Data Flow

The flow of various packets through the Packet Copier microblock is shown in [Figure 37-1](#) and is described in [Table 37-3](#).

Figure 37-1. Data Flow through Packet Copier

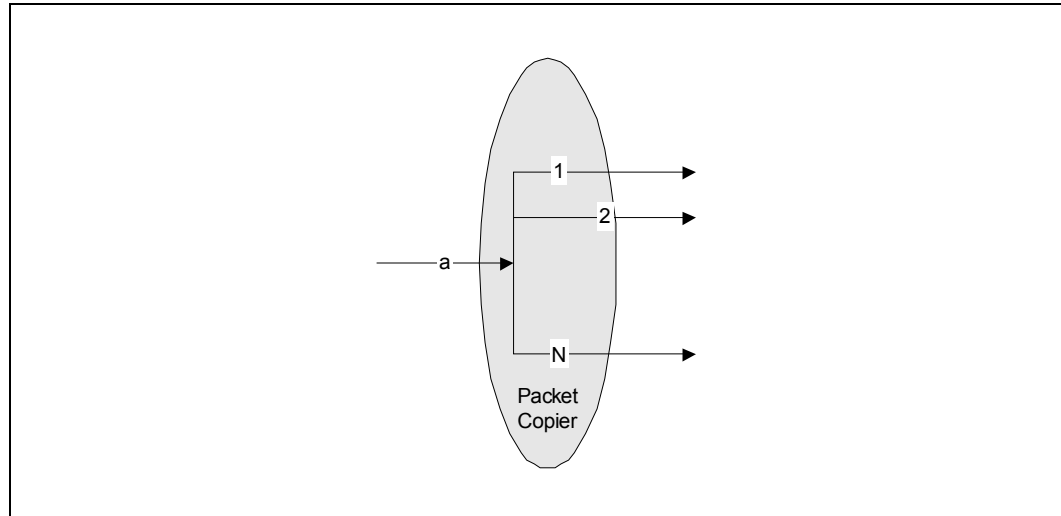


Table 37-3. Data Flow through Packet Copier

I/F Number	Packet Type	Action
a	Request message containing multicast packet	Required copies of this packet are made. Meta data for both parent buffers as well as child buffers is also updated. The required child buffers are allocated from the child buffer freelist.
1 ... N	Copy of multicast packet	Copies of multicast packet are distributed.

37.4 External Interfaces

The Packet Copier microblock's external interfaces are through scratch rings. The message format for request and response messages is described in the following sections.

37.4.1 Packet Copier Request Message

The format of the request message to the Packet Copier microblock is described in [Table 37-4](#). No error checking will be done on these fields.

Table 37-4. Packet Copier Request Message

LW	Bits	Size	Description
0	31:16	15	Packet Size
0	15:0	16	Queue Number
1	31:0	32	SOP Handle of the parent packet
2	31:0	32	EOP Handle of the parent packet
3	31:24	8	Reserved
3	23:16	8	Count of number of bits set in Port mask
3	15:0	16	Port mask. Current implementation assumes that the user data is an output port mask. Port numbers range from 0 through 15 and correspond to bits 0 through 15 respectively.

37.4.2 Packet Copier Response Message

The format of the response message from Packet Copier microblock is described in [Table 37-5](#). For each copy requested for a packet, a response message is sent.

Table 37-5. Packet Copier Response Message

LW	Bits	Size	Description
0	31:16	16	Packet Size
0	15:0	16	Queue Number
1	31:0	32	SOP Handle for the child packet
2	31:0	32	EOP Handle of the parent packet

37.5 Internal Data Structures

The Packet Copier microblock maintains a local queue of copy requests and copy contexts in local memory. Each entry in this queue is described in [Table 37-6](#).

Table 37-6. Packet Copy Context Queue Entry

LW	Bits	Size	Description
0	31:1	31	Reserved
0	0:0	1	Flag to indicate whether reference count has to be set in parent buffer or not.
1	31:0	32	Parent SOP Handle
2	31:0	32	Parent EOP Handle
3	31:0	32	Reserved
4	31:0	32	Reserved

Table 37-6. Packet Copy Context Queue Entry (Continued)

LW	Bits	Size	Description
5	31:0	32	Reserve
6	31:16	16	Count of number of copies required. Copied from the request.
6	15:0	16	Port mask to indicate which ports still require copies
7	31:16	16	Packet size obtained from parent meta data
7	15:0	16	Class ID obtained from parent meta data
8	31:0	32	Reserved
9	31:0	32	Parent meta data LW 1
10	31:0	32	Parent meta data LW 2
11	31:0	32	Parent meta data LW 3
12	31:0	32	Parent meta data LW 4
13	31:0	32	Parent meta data LW 5
14	31:0	32	Parent meta data LW 6. Also used to store parent buffer ID.
15	31:0	32	Reserved

37.6 Packet Replication Algorithm

A high-level algorithm for packet replication running on one microengine is described in [Table 37-7](#). This algorithm runs in two phases.

Table 37-7. Packet Replication Algorithm Running on One Microengine

<p>Phase1:</p> <ol style="list-style-type: none"> 1. Send signal to next thread. 2. Check if there is a valid copy request. If not, go to step 4. 3. If so, read meta data for the SOP handle in the request. 4. Check if we have a pre-fetch handle. If not, go to error processing. 5. Check if we have an entry to process from the head of local queue. If not, go to Phase2. 6. Get the copy number for this entry. 7. Create child meta data. 8. Set the reference count in parent buffer (done only once per request). 9. Flush the child meta data to SRAM. 10. Send response to next block. 11. Pre-fetch buffer for next iteration. 12. Check if all copies are completed for this entry. If so, dequeue next entry from local queue. 13. Wait for meta read to finish, signal from previous thread. <p>Phase2:</p> <ol style="list-style-type: none"> 1. Signal next thread. 2. Check if we issued a meta read in previous stage. 3. If so, enqueue request into local queue, store meta data. 4. Check if we have space in local queue. 5. If so, get another request. 6. Wait for the signals: buffer pre-fetch, previous thread signal, scratch get for request, scratch put for response, child meta write, and parent reference count write.

To achieve OC-192 data rates, the microblock can run on two microengines. For this configuration, the first microengine obtains and sends the requests to the second microengine. The second microengine simply makes copies for each request. The algorithm is described in [Table 37-8](#).

Table 37-8. Packet Replication Algorithm Running on Two Microengines

Microengine1:

The algorithm has two phases.

Phase1:

1. Send signal to next thread.
2. Check if we have a valid request.
3. If yes, read meta data.
4. Wait for meta read to finish, signal from previous thread.

Phase2:

1. Send signal to next thread.
2. Set the reference count in parent SOP buffer.
3. Send request to 2nd microengine via next neighbor ring.
4. Get next request.

Microengine2:

1. Signal next thread.
2. Check if we have a pre-fetch handle.
3. If so, check if we have a request to process.
4. Get next copy number to process.
5. Create child meta data for this copy.
6. Write child meta data.
7. Send response message.
8. Pre-fetch another child buffer.
9. Check if all copies are made, if so fetch another request from NN ring and initialize the context.
10. Wait for signal from previous thread, child meta write complete, pre-fetch buffer allocation complete, response write complete.

37.7 Startup/Shutdown

Before execution, the Packet Copier microblock waits for the system initialization signal. This signal indicates that the buffer freelist creation and scratch rings initialization are complete.

37.8 Performance Analysis

This microblock creates copies at OC-192 data rates when run on two microengines, as described in [Table 37-8 on page 650](#). The cycle counts for this scenario are listed in [Table 37-9](#). Each microengine has a budget of 57 cycles.

Table 37-9. Cycle Counts for Packet Replication on Two Microengines

Microengine	Phase	Cycle Count
1 st microengine	Phase 1	8
	Phase 2	26
2 nd microengine	--	34 (each copy) 48 (last copy, fetch next request)

A more flexible version of the Packet Copier runs in one microengine, though it is not suitable for OC-192 data rates. The characterization data for this block is described in [Table 37-10](#).

Table 37-10. Packet Copier Microblock Characterization Data

Data	Value
General:	
Microblock Name	Packet Copier (Packet Mode)
Microblock Version Number	1
Implementation Language	Microcode
Configuration Options use to gather this set of data	META_CACHE_SIZE=8. PACKET_MODE only. Only one microengine version is used to gather the data. (It is anticipated that this will be the most widely used configuration.)
Measurement Environment (tool settings)	IXA SDK 3.5, Optimization enabled.
Performance: (cycle counts, latencies, bandwidths)	
Instruction cycle count per packet (common case packet / worst case cycle count)	Worst case = 90 cycles, Common case = 67
Common-case packet/path assumptions to be documented here	A copy request arrives only after copies are made for previous request. In this case, the cycle count is simply for creating a copy, and writing the response
Scratch Memory	
# of longwords read (for bandwidth calculations)	4
# of longwords written (for bandwidth calculations)	3
# and type of each atomic operation performed (for bandwidth calculations)	0
SRAM	
# of longwords read	
# of longwords written	
# and type of each atomic operation performed (for bandwidth calculations)	None
Co-processors	N/A
bytes read (per channel)	0
bytes written (per channel)	0
DRAM	
# of quadwords read	0
# of quadwords written	0
Other I/O instructions (i.e. MSF, PCI, CAP, and Hash)	None
List of dependant I/O accesses in the longest latency path	
Per-Microengine Resources:	
Control-Store Usage (# of instructions used)	183

Table 37-10. Packet Copier Microblock Characterization Data (Continued)

Data	Value
Local Memory Footprint (# of long words used)	
Local Memory Configuration (shared, or per-context pointer)	Per-context pointers are used.
Local Memory - # of LM pointers used	Only Index 0 is used.
CAM used? (yes or no)	No

Global Resources:	
Scratch footprint (# of longwords used) – constant or formula ...	Not used
SRAM footprint (# of longwords used) – constant or formula ...	Not used
DRAM footprint (# of quadwords used) – constant or formula ...	Not used
Q-Array usage - # of queues used and if they need to be cached	No
CRC Unit used?	No
Hash Unit used? (yes or no)	No

MSF Usage Information:	
Media Bus Configuration	N/A
RBUF, TBUF usage	N/A
CBus signals	N/A

Other Information:	
Critical Section Length (compute cycles + memory accesses)	Phase 1 is critical section 1. Maximum cycles = 60. Maximum memory accesses are SRAM RD (6 LW), SRAM WR (1 LW), SRAM WR (7 LW), Scratch Put (3 LW), SRAM Deq. Phase 2 is critical section 2. Maximum cycles = 30. Maximum memory accesses are Scratch Get (3 LW)
# of phases	2 phases in 1 ME design.
Packet Metadata - fields read	First 6 LW of meta data.
Packet Metadata - fields written	LW '7' of parent buffer. LW 0 thru 6 of child meta data.
Header - fields read	None.
Header - fields written	None.
Entry and Exit Setup (assumptions on ME context)	The microblock resets the LM pointer to its context. Upon exit, other blocks can't assume the LM pointer values.
# of MEs required to run a single instance of microblock	Usually 1. Depending on the rate, two may be required.

Table 37-10. Packet Copier Microblock Characterization Data (Continued)

Data	Value
Data Rates	Two ME are required to achieve OC-192 rates on IXP2800
Memory Map of Key Data Structures	None.
Performance Considerations - ANY issues that have a direct impact on performance	N/A
Documentation:	
Thread Ordering Requirements	Threads need to run in order.
OS dependencies	N/A
Chip/hardware dependencies (i.e. Crypto Unit in 2850)	No dependency
Tested on which SDK Release(s)	IXA SDK 3.5
Tested on hardware? Which hardware configuration?	No
Tested in which applications (not an all inclusive list)	OC192 POS SPHY IPV4 Multicast Forwarder
Possible Configuration Options	PACKET_COPIER_2ME and related options
Major limitations (i.e. Rx/Tx microblocks written to use entire ME)	In TWO ME configuration, the NN ring is used up. Also threads perform busy wait loop to get/put data on NN ring, which will prevent other threads from running. The TWO ME configuration is primarily to address OC192 rates
Packet Sequencing Issues (esp. in POT applications)	Keeps copy responses in sequence
Core Component or Interface requirements or dependencies	N/A

38.1 .Overview

The Freelist Manager microblock can be used to maintain a packet buffer freelist. This microblock can replace the packet buffer linked list scheme that uses SRAM and the Q-Array hardware for maintaining a packet buffer freelist, which is described in [Section 2.4, “Buffer Chaining” on page 59](#). Both subsections, [Section 2.4.1, “Flat Queueing \(Cell Based Dequeue\)” on page 59](#) and [Section 2.4.2, “Hierarchical Queueing \(Packet Based Dequeue\)” on page 61](#) are relevant.

In this microblock, the packet buffer freelist is maintained in local memory. The advantage of using this microblock is that it eliminates I/O accesses to SRAM for allocating and freeing packet buffers. This eliminates associated I/O latency as well. These savings can be critical for some applications that make extensive use of SRAM bandwidth or target high data rates such as OC-192. The disadvantage of this microblock is that it occupies a microengine.

38.2 Assumptions and Dependencies

The Freelist Manager microblock has the following assumptions and dependencies:

- This microblock assumes a maximum of 16384 packet buffers in the application. This limitation is due to the size of local memory. Local memory stores the list of available buffers.
- This microblock allocates buffers by writing buffer handles on its outgoing NN ring. As a result, the Receive microblock must be ‘next’ to this microblock.
- This microblock frees buffers by reading buffer handles on its incoming NN ring. As a result, the Transmit microblock must be ‘before’ this microblock. This microblock also frees buffers by reading buffer handles on a scratch ring that is assigned as the DROP RING in the application.

38.3 Algorithm

38.3.1 Overview

The Freelist Manager microblock stores available buffers in a 3-level hierarchical list that is maintained in local memory. Level 1 and level 2 are used to index into level 3. Each buffer is represented as a bit in level 3. A value of ‘1’ implies that the buffer is available for allocation. A value of ‘0’ implies that the buffer is currently in use. Bits are grouped together to form a bitmask.

To search for an available buffer within a bitmask, the `ffs[]` instruction is used. On finding the first set bit, the bit is reset to ‘0’ indicating that the buffer is now in use. The corresponding buffer handle is computed and sent out.

To free a buffer, the list is indexed using the buffer handle and the corresponding bit is set to ‘1’.

38.3.2 Three-Level Hierarchical Implementation

Level 1 contains one 16-bit bitmask and each bit represents a level 2 entry. This gives us 16 level 2 entries.

Level 2 contains 16 entries and each entry is a 32-bit bitmask. Each bit represents a level 3 entry. This gives us $(16 \times 32) = 512$ level 3 entries.

Level 3 contains 512 entries and each entry is a 32-bit bitmask. Each bit represents a buffer. This gives us $(512 \times 32) = 16384$ buffers.

38.3.3 Summary

The algorithm can be summarized as shown below.

Table 38-1. Freelist Manager Microblock Algorithm

```

11. Read the DROP scratch ring.
12. If the DROP scratch ring is not empty
    Add the incoming buffer handle to the local memory buffer freelist.
13. If the outgoing NN ring is not full
    Write the prefetched buffer handle to the outgoing NN ring
14. If the incoming NN ring is not empty
    Prefetch a buffer handle from the incoming NN ring
15. Else
    Prefetch a buffer handle from the local memory buffer freelist
16. Goto Step 1.
  
```

38.4 Data Structures

See [Section 38.3.2](#) for details.

38.5 Performance Analysis

Table 38-2. Budget and Cycle Count

Description	Instruction Cycle Count
Worst Case Cycle count	55
600 MHz IXP2400 budget for POS min packet at OC-48 rates	97
600 MHz IXP2400 budget for Ethernet min packet at 4 Gbps rates	101
1.4 GHz IXP2800 budget for POS min packet at OC-192 rates	57
1.4 GHz IXP2800 budget for Ethernet min packet at 10 Gbps rates	94

Table 38-3. I/O Operations for Minimum Packet Worst Case

I/O Operations	Number of Longwords
Scratch read	1

The characterization data for this block is described in [Table 38-4](#)

Table 38-4. Freelist Manager Microblock Characterization Data

Data	Value
General:	
Microblock Name	freelist manager
Microblock Version Number	1
Implementation Language	microcode
Configuration Options use to gather this set of data	
Measurement Environment (tool settings)	IXA SDK 3.5
Performance: (cycle counts, latencies, bandwidths)	
Instruction cycle count per packet (common case packet / worst case cycle count)	55
Common-case packet/path assumptions to be documented here	
Scratch Memory	
# of longwords read (for bandwidth calculations)	1
# of longwords written (for bandwidth calculations)	0
# and type of each atomic operation performed (for bandwidth calculations)	0
SRAM	
# of longwords read	0
# of longwords written	0
# and type of each atomic operation performed (for bandwidth calculations)	0
DRAM	
# of quadwords read	0
# of quadwords written	0
Other I/O instructions (i.e. MSF, PCI, CAP, and Hash)	
List of dependant I/O accesses in the longest latency path	
Per-Microengine Resources:	

Table 38-4. Freelist Manager Microblock Characterization Data (Continued)

Data		Value
Control-Store Usage (# of instructions used)		206
Local Memory Footprint (# of long words used)		512
Local Memory Configuration (shared, or per-context pointer)		shared
Local Memory - # of LM pointers used		1
CAM used? (yes or no)		no

Global Resources:		
Scratch footprint (# of longwords used) – constant or formula ...		None
SRAM footprint (# of longwords used) – constant or formula ...		None
DRAM footprint (# of quadwords used) – constant or formula ...		None
Q-Array usage - # of queues used and if they need to be cached		None
CRC Unit used?		no
Hash Unit used? (yes or no)		no

MSF Usage Information:		
Media Bus Configuration		N/A
RBUF, TBUF usage		N/A
CBus signals		

Other Information:		
Critical Section Length (compute cycles + memory accesses)		
# of phases		2
Packet Metadata - fields read		
Packet Metadata - fields written		
Header - fields read		
Header - fields written		

Documentation:		
Thread Ordering Requirements		no
OS dependencies		VxWorks, Linux
Chip/hardware dependencies (i.e. Crypto Unit in 2850)		IXP2800
Tested on which SDK Release(s)		IXA SDK 3.5
Tested on hardware? Which hardware configuration?		POS

Table 38-4. Freelist Manager Microblock Characterization Data (Continued)

Data	Value
Tested in which applications (not an all inclusive list)	OC-192 POS Ingress applications
Possible Configuration Options	
Major limitations (i.e. Rx/Tx microblocks written to use entire ME)	
Packet Sequencing Issues (esp. in POT applications)	
Core Component or Interface requirements or dependencies	

Core Components

The core components are executed on the Xscale core processor and are counterparts of the microblocks. Core components configure microblocks and handle packets that the microengines don't have cycles to process. They also provide a programming interface to higher-level applications such as the Control Plane PDK.

Core components conform to the rules and APIs of the Core Component Infrastructure. The data flow of core components is designed to reflect the packet pipeline built by the microblocks on the microengines. In addition to core components, there are also support libraries, such as the Route Table Manager and Message Helper Support Library, which are used by the core components.

The System Application is responsible for initializing all modules of the system.

This section included the following core components:

- [Receive Components](#)
 - [Chapter 41, “POS RX Core Component”](#)
 - [Chapter 42, “CSIX RX Core Component”](#)
 - [Chapter 43, “Ethernet RX Core Component”](#)
- [Transmit Components](#)
 - [Chapter 44, “CSIX TX Core Component”](#)
 - [Chapter 45, “ATM/POS TX Core Component”](#)
 - [Chapter 46, “Ethernet TX Core Component”](#)
 - [Chapter 47, “Ethernet ARP Module”](#)
- [Queue Manager Components](#)
 - [Chapter 48, “Queue Manager Core Component”](#)
 - [Chapter 49, “Queue Manager \(DiffServ\) Core Component”](#)
- [Scheduler Components](#)
 - [Chapter 50, “Scheduler Core Component”](#)
 - [Chapter 51, “Scheduler \(DiffServ\) Core Component”](#)
- [Forwarder Components](#)
 - [Chapter 52, “IPv4 Forwarder Core Component”](#)
 - [Chapter 53, “IPv6 Forwarder Core Component”](#)
 - [Chapter 54, “IPv6 To IPv4 Tunneling Core Component”](#)
 - [Chapter 55, “NAT-PT Translation Core Components”](#)

- DiffServ Components
 - Chapter 56, “Six-Tuple Classifier Core Component”
 - Chapter 57, “Three Color Meter Core Component”
 - Chapter 58, “Weighted Random Early Detection (WRED) Core Component”
 - Chapter 59, “DSCP Classifier Core Component”
- Support Libraries
 - Chapter 60, “Route Table Manager”
 - Chapter 61, “Route Table Manager for IPV6 Core Component”
 - Chapter 62, “L2 Table Manager”
 - Chapter 63, “Message Helper and Support Library”
- Stack Driver Component
 - Chapter 64, “Stack Driver”
- SoftSAR Components
 - Chapter 65, “SoftSAR Core Components”

39.1 Overview

This chapter provides a software overview and describes the data flow for core components of the Intel[®] IXA SDK. The Intel[®] IXA SDK provides core components to support the following applications and interfaces:

- IPv4 Application—for details see [Section 39.4.1.1](#)
- DiffServ Application—for details see [Section 39.4.1.2](#)
- MPLS Application—for details see [Section 39.4.1.3](#)

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

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

In the following section, the IPv4 application is used to demonstrate key concepts in the design and use of core components. [Figure 39-1 on page 671](#) shows the core components, libraries, and system application for the IPv4 application. [Section 39.4.1](#) describes the use of core components by each of the sample applications.

39.1.1 Functional and Data Flow

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

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

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

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

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

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

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

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

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

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

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

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

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

- Route Table Manager library
- Message Helper and Support library

On the egress side, there are:

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

These standard C-language libraries provide services to the core components and higher-level applications.

Note: The Route Table Manager is tightly coupled to the IPv4 Forwarder core component and cannot be initialized from other clients due to the shared routing and next hop database tables.

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

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

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

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

39.1.2 Functional APIs Design Concept

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

- Messaging API
- Library API

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

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

39.2 APIs for Dynamic Property Updates

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

39.2.1 Dynamic Properties and Clients

Table 39-1 lists the dynamic properties and property clients..

Table 39-1. Dynamic Properties and Clients

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

39.2.2 Property Updates API and Data Structures

Additional APIs are provided by the core components to complement master/client relationships. Each core component that is master or client of the property implements the generic `ix_cc_<component name>_set_property` API. This API is implemented using both the Messaging API and Library API as described in section [Section 39.1.2, “Functional APIs Design Concept”](#) on page 665.

39.2.2.1 Properties Data Structure

These IP data structures define properties data for IPv4 and IPv6 interface information. For complete information, refer to the *Intel® Internet Exchange Architecture Software Building Blocks Reference Manual*.

Table 39-2. Properties Data Structure

Data Structure	Description
<code>ix_cc_media_type</code>	Media type for a port.
<code>ix_ipv4_properties</code>	Structure to hold IPv4 interface properties.
<code>ix_ipv6_properties</code>	Structure to hold IPv6 interface properties.
<code>ix_cc_ip_version</code>	Identification for IP version.
<code>ix_cc_ip_properties</code>	Container for IP properties of a virtual interface.
<code>ix_cc_physical_if_status</code>	Enumeration for physical interface state.
<code>ix_cc_properties</code>	Structure to hold interface properties, used to update properties, or get properties.
<code>ix_cc_properties_node</code>	Element of linked list of property structures.
<code>ix_cc_properties_msg</code>	Structure to hold property update messages.
<code>ix_cc_init_context</code>	Generic context passed into a core component during initialization.

39.2.3 Property ID

Property IDs are used to identify the property to update in a *set property* call. IDs are used for the first argument in the *set property* call and identify whether the fields from the `ix_cc_properties` structure (second argument) were changed. For a complete list of property IDs, refer to the *Intel® Internet Exchange Architecture Software Building Blocks Reference Manual*.

39.2.4 Property API Generic Prototype

Table 39-3 shows the generic prototype for the property API. For complete details, refer to the *Intel® Internet Exchange Architecture Software Building Blocks Reference Manual*.

Table 39-3. Property API

Name	Description
<code>ix_cc_<core component name>_set_property</code>	Sets the property, defined by <code>arg_PropId</code> to the core component.

39.2.5 Current Behavior of Property API

As described in [Section 39.1.2](#) each core component supports two types of APIs—a Messaging API and a Library API. The Messaging API provides calling applications an interface that does not require usage of the messaging mechanism of the Core Component Infrastructure. The Messaging API can be considered an external interface of the core components. The Library API provides an internal interface to the core components that does not require messaging support and does not depend on the Core Component Infrastructure.

For property interfaces, each core component with an interest in a property implements two functions—one based on the Messaging API and one based on the Library API—each of which takes a property ID and a properties data structure as arguments. Certain rules apply to behavior of these functions.

39.3 Handler Registration

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

Core components need to register different types of handlers.

- Packet Handlers—a core component may need to register one or more packet handlers for various types of packet flow.
- Message Handlers—a core component may need to register one or more Message handlers for different types of message input. For example, a handler might be registered for messages from *Message Helper* functions and others may be registered for properties to which the core component subscribes.

39.3.1 Support for the IXA Portability Framework and Core Components Infrastructure

When constructing core components there are two possible configurations for using the IXA Portability Framework:

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

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

The core components provide three types of APIs:

- Core Component Infrastructure APIs
- Messaging APIs
- Library APIs

For the Resource Manager-only configuration, the Library APIs are used. For the configuration that includes the Core Component Infrastructure, the Messaging APIs are used.

[Section 39.3.1.1](#) describes the configuration usage differences for the `init()` and `fini()` functions.

39.3.1.1 Usage of `init()` and `fini()` Functions

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

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


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

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

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

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

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

39.3.2 Operating System Independence of Core Components

This section discusses operating system independence of core components.

Any software system is dependent on the operating system in several areas—memory management, execution context, threads synchronization, inter-process communication, and access to the system hardware resources.

For file access, core components assume that targeted operating systems do not provide file system support. The functionality of the core components does not rely on the presence of a file system in the underlying operating system.

The combination of three uniform layers of APIs, described below, provides core components portability across VxWorks[®], Linux[®] kernel, and Linux user mode.

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

- Operating System Services Layer (OSSL)—described in the *Intel[®] Internet Exchange Architecture Portability Framework Developer's Manual*, provides operating system independent API for timers, synchronization primitives, thread allocation, and memory management for Intel XScale[®] core memory.
- Resource Manager—described in the *Intel[®] Internet Exchange Architecture Portability Framework Developer's Manual*, provides APIs for hardware resource allocation, initialization and configuration including:
 - Memory - SRAM, DRAM, Scratch and local memory
 - Hardware Queues and Rings
 - RBUFs, TBUFs
 - Microengine management including microcode loading, patching symbols, enable, disable, and so on

- Buffer management including hardware and software buffers shared between microblocks and core components
- Communication with microblocks
- Core Component Infrastructure—described in the *Intel® Internet Exchange Architecture Portability Framework Developer's Manual*, provides a set of APIs for defining the thread of execution for a core component and initialization and termination of the core component. It also provides a mechanism for component-to-component packet and message handling—including inter-process and inter-task communication.

Table 39-4 shows how typical core component functionality maps to the services of these three layers.

Table 39-4. Core Component Functionality Mapped to Different Framework Layers

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

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

There are a few components that have more dependency on the operating system including the Stack Driver and the system application. The system application is responsible for system startup and is operating system dependent with respect to how the system is invoked in VxWorks* or Linux*. The Stack Driver provides network support to the operating system specific to the local TCP/IP stack.

39.4 High-Level Overview of the Core Components

This section describes the building block core components delivered with the IXA SDK 3.1 as well as the applications based with these core components.

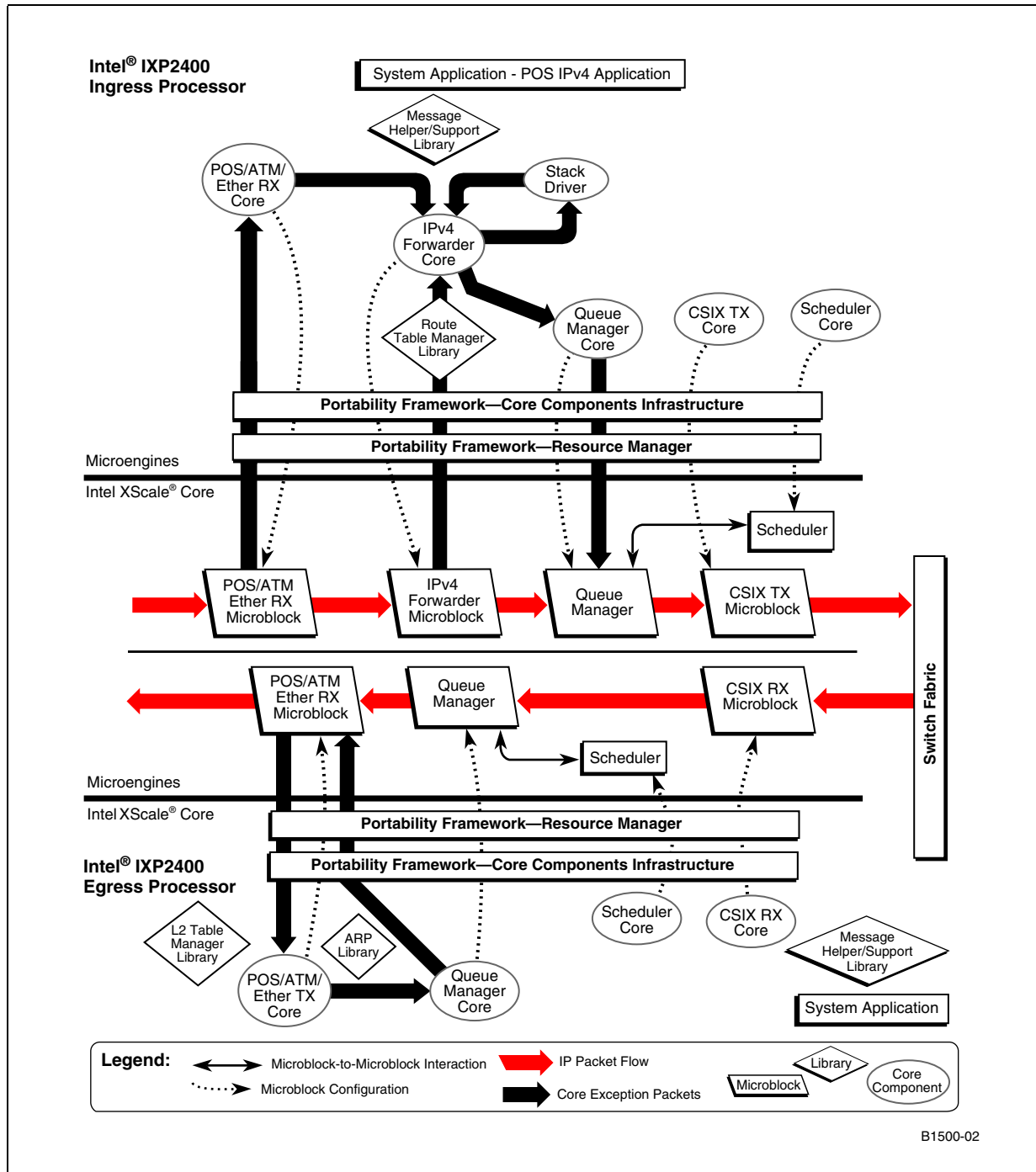
39.4.1 Applications

This section describes the core component usage of the applications that are shipped with the IXA SDK 3.1.

39.4.1.1 IPv4 Application

The software architecture of the POS and Ethernet IPv4 application is shown in Figure 39-1. This diagram depicts—for both ingress and egress network processors—the microblocks, core components, and packet flow used in the IPv4 application.

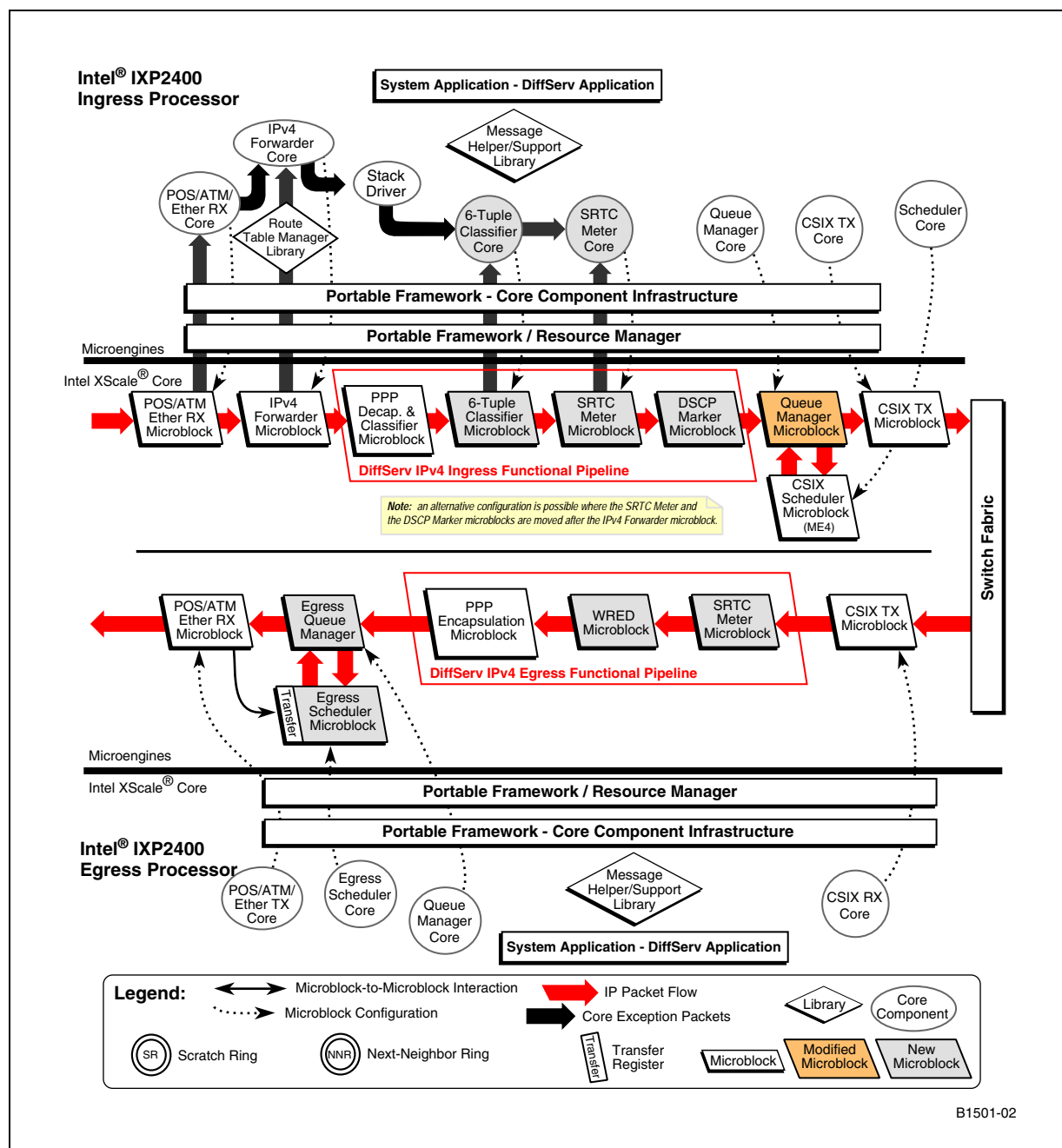
Figure 39-1. POS IPv4 Software Architecture



39.4.1.2 DiffServ Application

Figure 39-2 shows the software architecture of the DiffServ application. It depicts—for the ingress and egress network processor—the microblocks, core components, and packet flow used in the DiffServ application.

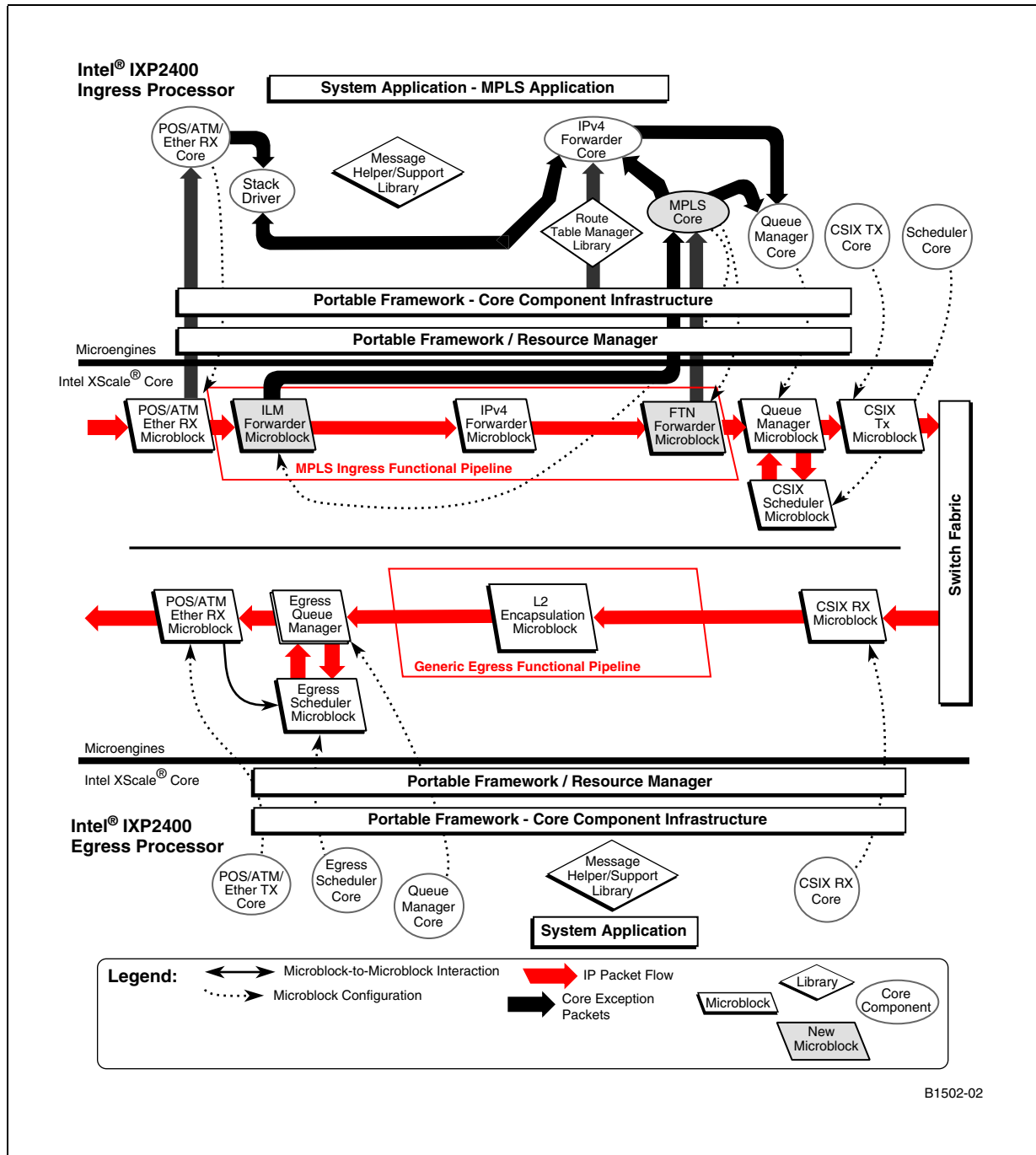
Figure 39-2. DiffServ Application Overview



39.4.1.3 MPLS Application

The software architecture of the MPLS application is shown in Figure 39-3 depicts—for the ingress and egress network processor—the microblocks, core components, and packet flow used in the MPLS application.

Figure 39-3. MPLS Application Overview

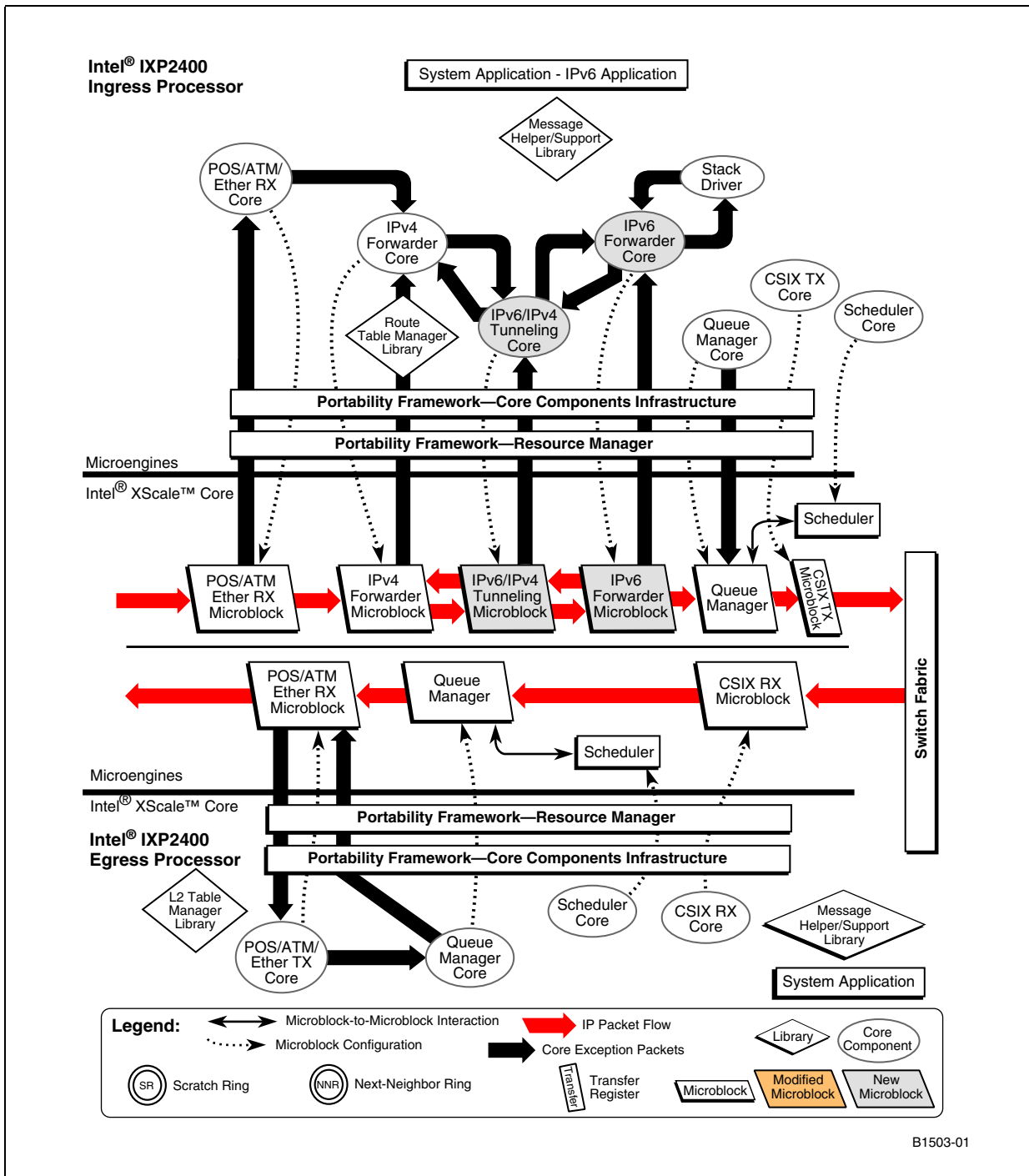


B1502-02

39.4.1.4 IPv6 Application

Figure 39-4 shows the software components needed to implement an IPv4 and IPv6 forwarding application for Ethernet. All the context pipe-stages (e.g. Packet RX, Queue Manager, Scheduler etc.) occupy an entire microengine. Each context pipe-stage is mapped to a single microblock running on a ME with or without a dispatch loop. The functional pipeline runs on four microengines and implements the layer-2 (Ethernet) decapsulation and the IPv4 forwarder, IPv6 forwarder and V6/V4 tunneling blocks. The tunneling microblock is needed when IPv6 packets need to be tunneled over an IPv4 network.

Figure 39-4. Software Components for IPv4/IPv6 Forwarding and V6/V4 Tunneling



39.4.2 Building Block Core Components

This section provides a brief overview of the core components described in subsequent chapters of this document. For a detailed description of the API for each of these core components, see *Intel® Internet Exchange Architecture Software Building Blocks Reference Manual*. The core components are listed in the order in which they appear in subsequent chapters in this document.

39.4.2.1 POS RX

The POS RX core component does the following:

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

For complete details, see [Chapter 41, “POS RX Core Component.”](#)

39.4.2.2 IPv4 Forwarder

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

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

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

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

For complete details, see [Chapter 52, “IPv4 Forwarder Core Component.”](#)

39.4.2.3 Queue Manager (QM)

Two Queue Manager core components exist in the system—one on the ingress side and another on the egress side—with slight differences in typical data processing. These differences are mentioned explicitly throughout this section and in [Chapter 48, “Queue Manager Core Component.”](#) The Ingress Queue Manager core component corresponds with the Cell Based Queue Manager microblock, and the Egress Queue Manager Core component corresponds with the Packet Based

Queue Manager microblock. However, on reference boards that provide full duplex capability, the number of Queue Manager core components depends on the system configuration since there may be either one or two Queue Manager core components.

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

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

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

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

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

39.4.2.4 Scheduler

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

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

39.4.2.5 CSIX TX

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

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

For complete details, see [Chapter 44, “CSIX TX Core Component.”](#)

39.4.2.6 CSIX RX

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

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

For complete details, see [Chapter 42, “CSIX RX Core Component.”](#)

39.4.2.7 ATM/POS TX

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

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

For complete details, see [Chapter 45, “ATM/POS TX Core Component.”](#)

39.4.2.8 Ethernet RX

The Ethernet RX core component performs the following functions:

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

For complete details, see [Chapter 43, “Ethernet RX Core Component.”](#)

39.4.2.9 Ethernet TX

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

For complete details, see [Chapter 46, “Ethernet TX Core Component.”](#)

39.4.2.9.1 Ethernet ARP Module

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

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

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

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

39.4.2.10 L2 Table Manager

The L2 Table Manager provides an API to manage, add, and delete entries in the L2 Table. The L2 Table corresponds to the Route Table and Next Hop Database on the ingress side. See [Chapter 60, “Route Table Manager”](#) for details on the Next Hop Database.

For complete details see [Chapter 62, “L2 Table Manager.”](#)

39.4.2.11 Route Table Manager

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

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

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

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

For complete details, see [Chapter 60, “Route Table Manager”](#).

39.4.2.11.1 Next Hop Database

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

39.4.2.11.2 TCAM

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

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

39.4.2.11.3 Software LPM

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

39.4.2.12 Six-Tuple Classifier

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

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

For complete details see [Chapter 56, “Six-Tuple Classifier Core Component.”](#)

39.4.2.13 Single Rate Three Color Meter

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

- Initializes and configures the Single Rate Three Color Meter microblock by patching symbols.

- Provides an API interface to add, remove and update Single Rate Three Color Meter instances. The instance parameters are stored in a meter table shared between the Single Rate Three Color Meter core component and the microblock.
- Provides an API interface to read statistics associated with a selected Single Rate Three Color Meter instance.

For complete details see [Chapter 57, “Three Color Meter Core Component.”](#)

39.4.2.14 Weighted Random Early Detection (WRED)

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

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

For details see [Chapter 58, “Weighted Random Early Detection \(WRED\) Core Component.”](#)

39.4.2.15 Queue Manager for DiffServ

The Egress Queue Manager core component for DiffServ is similar to the Queue Manager core component detailed in [Chapter 48, “Queue Manager Core Component.”](#)

The only changes are:

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

39.4.2.16 Scheduler for DiffServ

The Scheduler core component for DiffServ is virtually identical to the Scheduler core component described in [Chapter 50, “Scheduler Core Component.”](#) The only difference is that the Scheduler (DiffServ) core component adds an additional patching symbol.

For details see [Chapter 51, “Scheduler \(DiffServ\) Core Component.”](#)

39.4.2.17 IPv6 Forwarder

The IPv6 core component performs the following functions:

- Configures the IPv6 microblock—static configuration.
- Provides message and packet handlers to receive messages and packets from other core components and the IPV6 microblock.

- Generates ICMPv6 error messages.
- Validates an IP header.
- Handles extension headers.
- Supports *neighbor discovery*.
- Supports stateless address automatic configuration.

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

For details see [Chapter 53, “IPv6 Forwarder Core Component.”](#)

39.4.2.18 IPv6 to IPv4 Tunneling

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

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

The following functionality is provided:

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

For details see [Chapter 54, “IPv6 To IPv4 Tunneling Core Component.”](#)

39.4.2.19 Route Table Manager for IPv6

The Route Table Manager for IPv6 (RTMv6) stores and retrieves data on behalf of the IPv6 Forwarder core component. In addition, the RTMv6 stores the data in a format consistent with the requirements of the IPv6 Forwarder microblock. RTMv6 uses the Lookup library API to access the route tables for IPv6. To manage next hops, RTMv6 uses a Next Hop Database (NHDB)—see [Section 39.4.2.11](#). The RTMv6 exposes an API allowing the IPv6 core component to add to, remove from, or lookup items in the route table. This API is implemented as a library supporting a single client.

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

For details on the Route Table Manager for IPv6 see [Chapter 61, “Route Table Manager for IPV6 Core Component.”](#)

39.4.2.20 Stack Driver

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

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

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

For complete details, see [Chapter 64, “Stack Driver”](#).

39.4.2.21 System Application

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

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

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

For complete details, see [Chapter 40, “System Application,”](#).

39.4.2.22 Message Helper and Support Library

The Message Helper and Support library is the layer of translation between messages and core component APIs. This library specifies the handling of function arguments and the returning result back to the client. The Message Helper and Support Library provides a C-language interface to the clients of the core components and use IXA Portability Framework's message mechanism to invoke internal core component API function. The Message Helper library translates the message

API into the messages and enforces the mechanism of delivering messages to the Library APIs of each core component and sending the result back to the clients. It helps clients to simplify the programming model by hiding the marshaling of data inside the messages.

For complete details, see [Chapter 63, “Message Helper and Support Library”](#).

39.4.2.23 SoftSAR Core Components

The SoftSAR core components include the following sub-components:

- [Section 39.4.2.23.1, “SAR Core Components”](#)
- [Section 39.4.2.23.2, “ATM RX Core Components”](#)
- [Section 39.4.2.23.3, “ATM TX Core Components”](#)
- [Section 39.4.2.23.4, “TM4.1 Core Components”](#)

For complete details see [Chapter 65, “SoftSAR Core Components”](#).

39.4.2.23.1 SAR Core Components

The SoftSAR core components provide the following functionalities:

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

39.4.2.23.2 ATM RX Core Components

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

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

39.4.2.23.3 ATM TX Core Components

The ATM TX Core Component performs the following functions:

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

- Initializes and configures the ATM framer device

39.4.2.23.4 TM4.1 Core Components

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

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

39.4.3 MPLS Forwarder Core Component

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

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

- Initializes and maintains the data structures for the ILM Forwarder and FTN Forwarder microblocks
- Configures the ILM Forwarder and FTN Forwarder microblocks (static configuration)
- Provides message and packet handlers to receive messages from the control plane and packets from the ILM Forwarder and FTN Forwarder microblocks
- Handles fragmentation of MPLS packets
- Implements "slow path" forwarding for fragmented MPLS packets
- Handles packets with special MPLS label values

40.1 Overview

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

- Initializes the Core Component Infrastructure.
- Provides a set of hooks into various stages of initialization to allow easy addition of user-provided initialization code.
- Loads microcode and starts the microengines.
- Allocates a set of scratch rings based on the system configuration.
- Allocates various free lists and provides a message helper function to allow core components to access them.
- Creates execution engines based on a user provided configuration.
- Instantiates core components within execution engines based on a user provided configuration.
- Initializes support facilities.
- Provides proper system shutdown.
- Allows re-initialization of the system.

For external APIs see [Chapter 3, “System Application”](#) of the *IXA Software Building Blocks Reference Manual*.

40.2 Assumptions

This System Application utilizes the following services of the:

- Core Component Infrastructure
- Resource Manager
- Operation System Service Layer (OSSL)

The following assumptions about their basic operation are made.

- The combination of Core Component Infrastructure, Resource Manager and OSSL provide sufficient operating system abstraction so that the System Application is completely independent of the OS.
- A method is provided to allow the registry to be pre-loaded with a fixed set of values provided by the developer. These pre-loaded registry values must be available after the return of the Core Component Infrastructure initialization function.

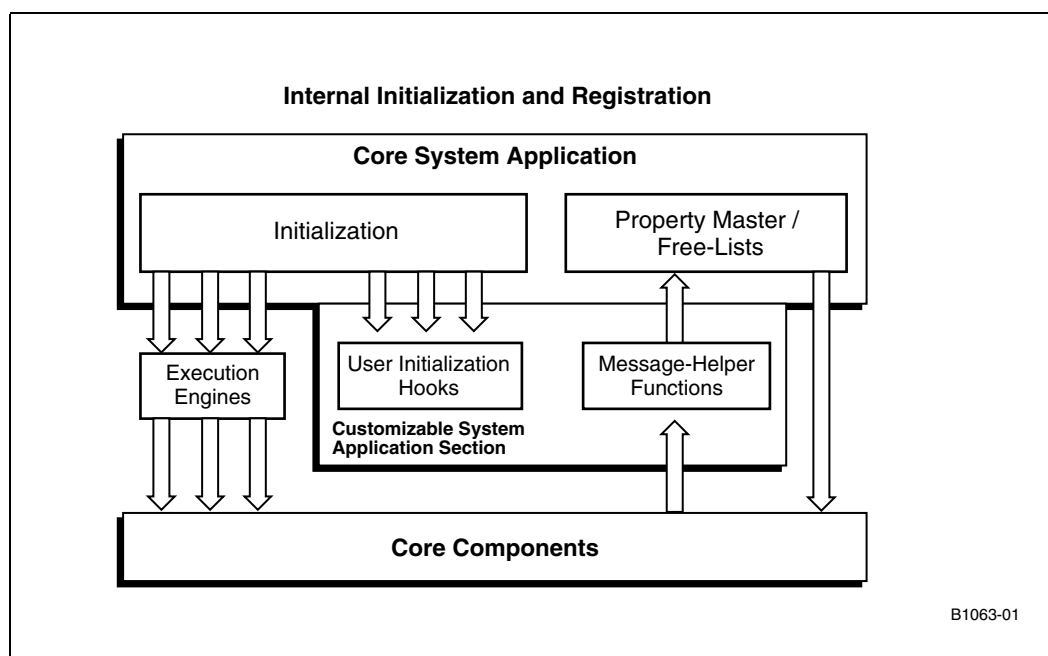
40.3 Design

The design of the System Application provides a configurable way to initialize the Core Component Infrastructure, Core Components and support facilities. It is the intention of this design to minimize code changes to the System Application when a developer needs to change the configuration. Wherever possible, hard-coded actions are replaced with use of configuration data.

The number of execution engines and individual core components running inside each is selected at configuration time. The configuration parameters for these selections are stored in the registry (repository) or, in the absence of a registry, a set of `#defines` in common header files. The initialization code operates in the same way regardless of where the configuration information is stored.

The System Application runs on both the ingress and egress processors. There is no code in the System Application that depends which CPU it is executing on. However, the configuration information is specific to the processor (ingress or egress), such as which core components to load, etc.

Figure 40-1. System Application Overview



40.3.1 Initialization Sequence

The sequence of steps that occur during system initialization are listed below. The details of each step are provided in later sections. The User Initialization Hooks are not explicitly called out in the list. They are detailed in [Section 40.3.4, “User Initialization/Shutdown Hooks”](#) on page 691.

1. Initialize the Core Component Infrastructure (see [Section 40.3.5, “Core Component Infrastructure Initialization”](#)).
2. Set the microcode for each microengine.
3. Create scratch rings (see [Section 40.3.8, “Scratch Rings”](#)).

4. Create the System Application's execution engine (see [Section 40.3.6, "Execution Engines"](#)).
5. Initialize the Property Master module (see [Section 40.3.9, "Property Master"](#)).
6. Start the execution engines (see [Section 40.3.7, "Core Components"](#)).
 - a. Create core components for each execution engine.
 - b. Patch symbols for each core component.
7. Initialize support facilities (see [Section 40.3.10, "Support Facilities"](#)).
8. Load patched microcode onto microengines (see [Section 40.3.3, "Load Microcode and Start Microengines"](#)).
9. Start microengines.

40.3.2 Shutdown and Re-initialization

The system is designed to be static. The System Application does not support run-time loading and shutdown of individual components. The primary goal of the shutting down the System Application is to bring all aspects of the system to a halt in a stable way that allows the system to be either reinitialized or to remain inactive.

The following operations are performed during shutdown.

- Halt microengines and place them in a state which allows new code to be loaded. Once in this state no data from any source is processed.
- Shutdown support services.
- Shutdown all core components.
- Halt execution engines.
- Halt the System Application's execution engine.
- Free scratch rings.
- Deallocate free lists.
- Shutdown the Core Component Infrastructure.
- Clean up all data used by the System Application and end execution.

During each stage of the shutdown process, user shutdown hooks are called. Each hook is passed the context returned by their corresponding initialization hook.

The System Application tracks the success/failure of each of the above steps. If any of the steps indicate that the system has been left in an inconsistent state, re-initialization is not possible.

The shutdown process can be triggered by internal or external sources. Controlling software can make a call to perform a shutdown. Components of the system can also trigger a shutdown due to an unrecoverable error.

The shutdown API call is implemented as a core component safe API. When an error occurs that requires a shutdown, there is no guarantee that basic Core Component Infrastructure resources are available or functioning properly. The System Application pre-allocates a semaphore to be used as a trigger for shutdown. When a core component or other entity calls the shutdown API, the semaphore is cleared. This operation is non-blocking and therefore core component safe.

40.3.2.1 Re-initialization

The re-initialization procedure is configured by the system designer. [Table 40-1](#) lists the available options:

Table 40-1. Re-initialization Procedure Options

Option	Description
Automatic Restart:	When this option is selected, the System Application immediately restarts after the shutdown process. This avoids the need to do a cold-boot.
Restart on command	After the shutdown procedure is complete, the system remains shut down. Other software can manually restart the System Application via an API call.
Restart Disabled	Once a shutdown has occurred, the system must be rebooted to be restarted.

The system default re-initialization policy can be overridden by the caller of the shutdown function.

40.3.2.2 Start and Shut Down API

[Table 40-2](#) shows the function calls that start and shuts down the system application. For complete details, see the *Intel® Internet Exchange Architecture Software Building Blocks Reference Manual*.

Table 40-2. Loading Microcode and Starting Engines

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

40.3.3 Load Microcode and Start Microengines

The System Application facilitates the loading of microcode and starts microengines. Each core component must patch symbols for its corresponding microblock.

The following initialization constraints determine which tasks must be performed:

- Microcode must be set (i.e. sent to Resource Manager) before any core components are started.
- All core components must be initialized before code is loaded onto the microengines.
- Microcode must be loaded before the microengines are started.
- After the microengines are halted, all of the above steps must be performed again before restarting the engines.

The System Application initialization sequence takes these factors into account.

The actual procedure for setting microcode on VxWorks is not provided by the Resource Manager.

40.3.3.1 Loading Microcode and Starting Microengines API

Table 40-3 shows the function call for the loading microcode and starting the microengines. For complete details, see the *Intel® Internet Exchange Architecture Software Building Blocks Reference Manual*.

Table 40-3. Loading Microcode and Starting Engines

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

40.3.4 User Initialization/Shutdown Hooks

A system designer may have a variety of initialization tasks beyond what the default tasks provided by the System Application. A series of initialization/shutdown hooks are provided at various stages of initialization. These hooks allow a system designer to easily add additional custom code without the need to modify the core System Application.

The use of these hooks is by no means required but does provide a well defined, tested way to add additional initialization/shutdown code. The advantages to using these hooks includes:

- The core System Application does not need to be modified.
- Testing can be limited to the new code within the hooks and not the already tested System Application.
- An updated System Application provided by Intel can be used immediately without re-implementing changes to the System Application.

These hooks are implemented as a set of pre-defined functions to which a system designer can add custom code. The functions are implemented by a designer in a library that is linked into the System Application.

These functions have the following prototype where *stage* is replaced with the stage of initialization or shutdown.

40.3.4.1 Initialization/Shutdown Hooks

Table 40-4 shows the user initialization and shutdown hooks. For complete details, see the *IXA Software Building Blocks Reference Manual*.

Table 40-4. User Initialization and Shutdown Hooks

API	Description
<code>ix_sa_init_hook_first()</code>	Called before the Core Component Infrastructure is initialized.
<code>ix_sa_init_hook_pre_ee()</code>	Called before starting any execution engine or core components
<code>ix_sa_init_hook_ee()</code>	Called from the execution engine context.
<code>ix_sa_init_hook_pre_me()</code>	Called before the Microengines are started.
<code>ix_sa_init_hook_last()</code>	Called after all the initialization is complete.

Table 40-4. User Initialization and Shutdown Hooks

<code>ix_sa_shutdown_hook_first()</code>	Called before the shutdown begins but before the system application performs any shutdown.
<code>ix_sa_shutdown_hook_post_me()</code>	Called after the microengines are stopped.
<code>ix_sa_shutdown_hook_post_ee()</code>	Called before the Core Component Infrastructure is shutdown.
<code>ix_sa_shutdown_hook_ee()</code>	Called from the context of each execution engine before any core components are shutdown.
<code>ix_sa_shutdown_hook_last()</code>	Called after all shutdown activity is completed.

Each of the initialization hooks can return a context which is passed to the corresponding shutdown hook.

40.3.5 Core Component Infrastructure Initialization

The Core Component Infrastructure is initialized by a single call to `ix_cci_init()`. This function must be called only once and this call must be before any other Core Component Infrastructure API calls are made.

After `ix_cci_fini()` is called during shutdown, this function can again be called when the system is re-initialized.

40.3.6 Execution Engines

After the Core Component Infrastructure is initialized, the System Application creates the execution engines. The number of execution engines depends on the configuration in the registry.

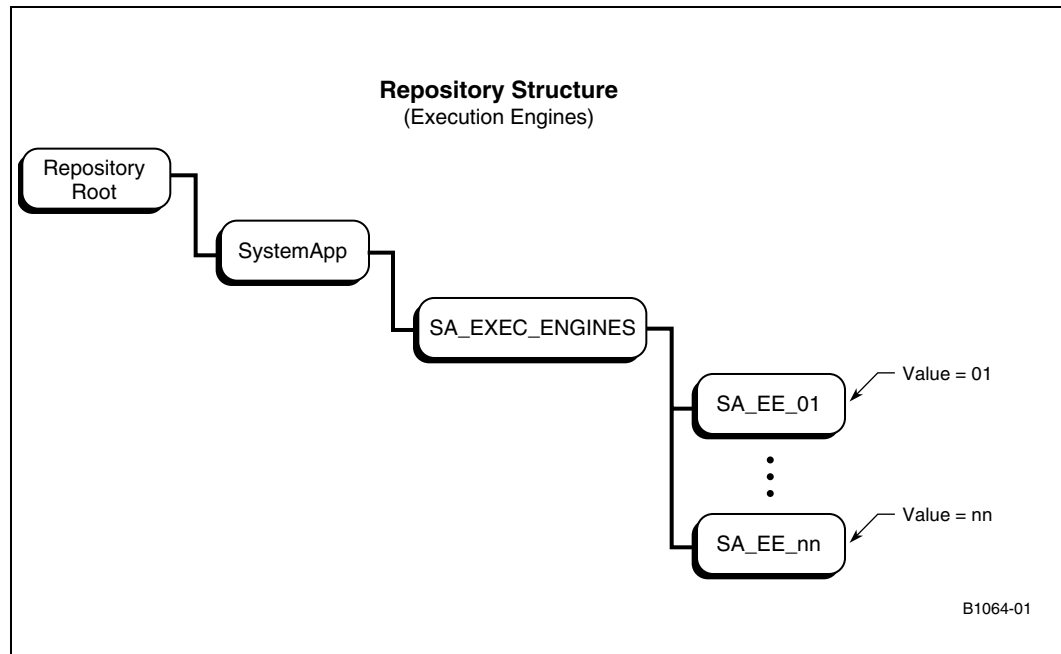
A single set of `init` and `fini` functions are used for all execution engines. These generic functions initialize core components base on registry entries. The detailed behavior of these functions is described in [Section 40.3.7, “Core Components”](#).

The `ix_cci_exe_run()` function is called once for each required execution engine.

The registry contains one property for each execution engine that needs to be created. The System Application iterates through the child properties of `SA_EXEC_ENGINES`, creating an execution engine for each entry in the registry. The name of these properties are determined by the system designer. `SA_EE_nn` is the naming convention used to specify execution engines in the repository.

The value associated with the execution engine properties (`SA_EE_nn` in the diagram) matches the execution engine index at runtime. The values must be sequential starting with 0. When entering the `init` function of the execution engine, the System Application uses a macro to fetch the execution engines unique ID. This ID is used here to allow the `init` function to know which execution engine is starting so it can choose the appropriate configuration information.

Figure 40-2. Execution Engine Repository Structure



Example Execution Engine Registry Entries

```

Root/SystemApp/SA_EXEC_ENGINES/SA_EE_01 à{value=01}
/SA_EE_02 à{value=02}

```

40.3.7 Core Components

The System Application is responsible for creating all core components configured into the system. One or more core components run within the context of an execution engine. The creation process takes place within the `init` function of the execution engines (see [Section 40.3.6, “Execution Engines”](#)).

Within the `init` function, a unique ID is extracted from the execution engine handle via a macro. This ID is used to select the set of core components to create based on the configuration.

There are two main configuration parameters which play a role in the creation of core components.

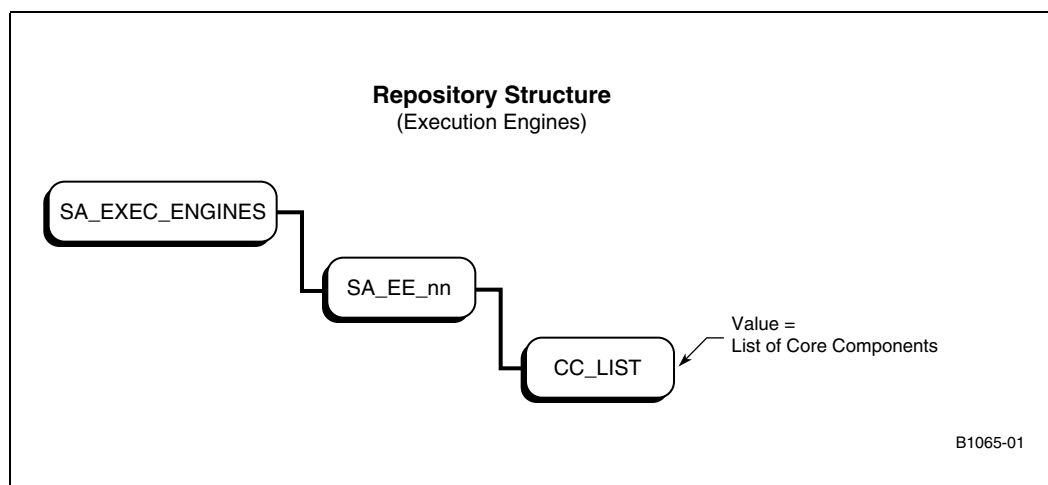
Table 40-5. Core Component Configuration Parameters

Parameter	Description
A function pointer table	A global static function table is defined which contains pointers to the <code>init</code> and <code>fini</code> functions of all core components.
A list of core components	For each execution engine, a list of the core components to launch is provided. This list takes the form of a list of indices into the function pointer table.

Because the list of core components can be changed in the registry, the function table must include all core components. It must also be accessible by the common execution engine `init` function so it can be used during core component creation.

Each execution engine property in the registry contains a single child property, `CC_LIST`. The value of `CC_LIST` is an array of indices into the function pointer table. For each index stored in `CC_LIST`, a core component is created passing in the correct `init` and `fini` pointers from the function pointer table.

Figure 40-3. Core Component Repository Structure



The following operations are performed within the execution engine `init` function:

1. Determine the current execution engines ID from the execution engine handle.
2. Query the registry to find the execution engine entry matching the ID.
3. Read the value of the `CC_LIST` property under the correct execution engine.
4. For each entry in the `CC_LIST` create a core component using the `init` and `fini` functions stored in the function pointer table. The `CC_LIST` entry is used as an index into the function pointer table.

40.3.8 Scratch Rings

A set of scratch rings must be allocated for the microcode. Depending on how the microblocks are organized, various rings need to be placed between them.

The number and configuration of these rings cannot be inferred from the core component configuration. Therefore a separate set of configuration items are specified in the registry (or header file).

The configuration information for each scratch ring is as follows:

- Ring ID
- Element Size
- Number of elements.

For each configured scratch ring, the System Application allocates the correct amount of scratch memory from the Resource Manager. This memory is then passed to `ix_rm_ring_create` to create the ring.

40.3.9 Property Master

All properties are mastered by the Stack Driver. Refer to [Chapter 64, “Stack Driver”](#).

40.3.9.1 Mastered Properties

In this release, no properties are mastered.

40.3.10 Support Facilities

No support facilities are initialized.

Receive Components

The Receive Core Components include the following:

- [Chapter 41, “POS RX Core Component”](#)
POS RX Core Component corresponds to the POS RX microblock. This component configures the POS RX microblock.
- [Chapter 42, “CSIX RX Core Component”](#)
CSIX RX Core Component performs initialization and configuration for the CSIX RX microblock through patching imported symbols and memory block into the microblock.
- [Chapter 43, “Ethernet RX Core Component”](#)
Ethernet RX Core Component handles the ARP exception packets received from the Ethernet RX microblock and also takes care of configuring the Packet RX microblock. In addition, Ethernet RX Core Component maintains the Ethernet MAC filtering table for MAC address filtering used by the microblock.

41.1 Overview

The POS RX core component performs the following functions:

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

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

For more information on the POS RX microblock, see [Chapter 4, “Packet RX Microblock.”](#) and for external APIs see [Chapter 4, “POS RX API”](#) of the *IXA Software Building Blocks Reference Manual*.

41.2 Assumptions and Dependencies

41.2.1 Assumptions

The following assumptions have been made based on the POS RX microblock design (see [Chapter 4, “Packet RX Microblock”](#)).

- The only exception packets that are passed are PPP LCP/PCP packets. These packets are sent to the output of this core component. The Core Components Infrastructure is used to deliver packets from the Packet RX microblock to the POS RX core component (see the *Intel® Internet Exchange Architecture Portability Framework Developer's Manual*).
- The POS RX core component needs to patch in symbols and memory for use by the Packet RX microblock.
- The POS RX core component uses the Resource Manager APIs for memory allocation, patching symbols etc. (see the *Intel® Internet Exchange Architecture Portability Framework Reference Manual*).

41.2.2 Dependencies

The System Application and system registry initialize and provide configuration information to the POS RX core component. For more information on the System Application, see [Chapter 40, “System Application.”](#)

41.3 Data flow

The POS RX core component defines a single packet input for receiving exception packets from the Packet RX microblock. It does not receive any packets from neighboring core components (see [Section 41.2, “Assumptions and Dependencies” on page 699](#)). The input id is defined in the system file, `bindings.h` as follows:

```
#define IX_CC_POS_RX_PKT_INPUT
```

The POS RX core component defines a single output for sending data packets to other core components. The output is bound to `IX_CC_PKT_DROP` by default, resulting in all exception packets being dropped. If any application or core component needs to capture these packets, then it can be bound to this output. The output ID for outgoing packets is defined in system's `bindings.h` as

```
#define IX_CC_POS_RX_PKT_OUTPUT
```

The POS RX core component uses API functions exposed by the Core Component Infrastructure to send packets (see the *Intel® Internet Exchange Architecture Portability Framework Reference Manual*).

All PPP control packets are sent to the POS RX core component by the Packet RX microblock. The PPP control packets are then sent to the assigned output.

41.4 Configuration and Initialization

41.4.1 Static Configuration Data

[Table 41-1](#) lists the static configuration data defined by the POS RX core component. This data is obtained from the system registry during the core component's initialization. If the system registry is not present, then the default values defined in the component own header file are used. This data includes configuration parameters to be patched into the microblocks.

Table 41-1. POS RX Core Component Static Configuration Data

Data	Default	Description
POS_RX_MEV2_NUMBER	0x00000001 (ME0)	Microengine numbers to be allocated to POS RX microblock. Only bit [0:7] are valid for ME0 - ME7. This data is required only when POS is included.

41.4.2 Patching Symbols and Memory Allocation

Table 41-2 lists the symbols which the POS RX core component must patch for use by the Packet RX microblock.

Table 41-2. POS RX Core Component Patch Symbols

Data	Description
PACKET_COUNTERS_SRAM_BASE	This is the base for the various counters used and maintained by the Packet RX microblock for management and debugging. Various counters used by the POS RX microblock for debugging are allocated by using the Resource Manager's 64-bit counter support.

No memory needs to be patched by the POS RX core component. The Receive Reassembly Context (RXC) is stored entirely in local memory.

41.5 External API

This section lists the external API for POS RX core component. For complete details, see Chapter 4, “POS RX API” in the *IXA Software Building Blocks Reference Manual*.

41.5.1 Core Component Infrastructure API

Table 41-3 lists the functions in the Core Component Infrastructure API for the POS RX core component.

Table 41-3. POS RX Core Component Infrastructure API

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

41.5.2 Messaging API

Table 41-4 lists the functions in the Messaging API for the POS RX core component.

Table 41-4. POS RX Core Component Messaging API

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

41.5.3 Library API

Table 41-5 lists the functions in the Library API for the POS RX core component.

Table 41-5. POS RX Core Component Library API

API	Description
<code>ix_cc_pos_rx_get_statistics_info()</code>	Returns the requested statistics information.
<code>ix_cc_pos_rx_get_interface_state()</code>	Returns the current state of the POS RX interface for a specified physical port.

The CSIX RX Core Component performs the following functions:

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

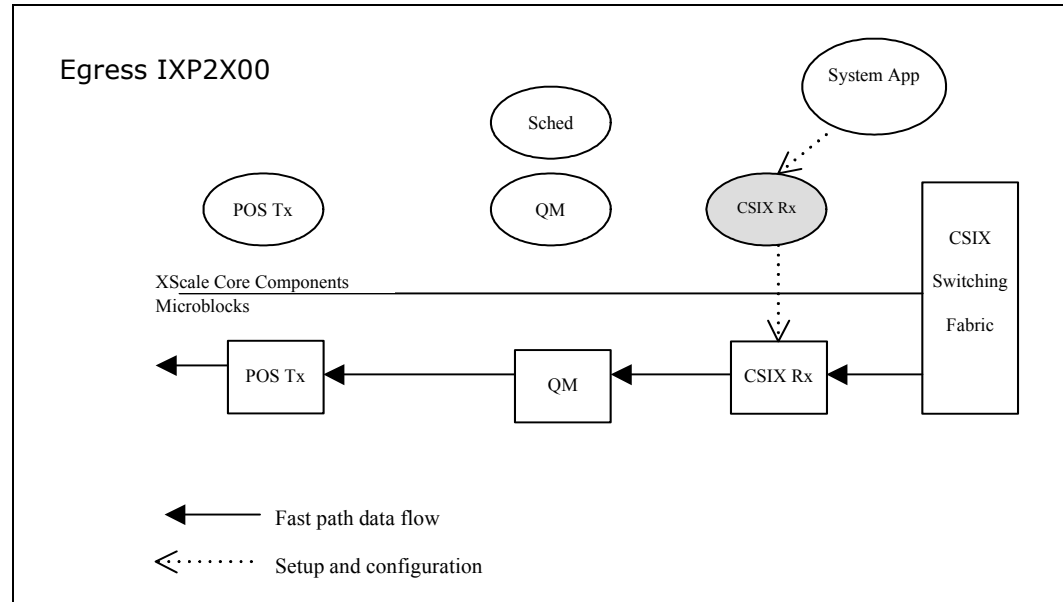
The CSIX RX Core Component does not process data packets since it has no exception packets from the CSIX RX microblock or neighboring core components.

For more information on the CSIX RX microblock, see [Chapter 5, “CSIX RX Microblock”](#) and for external APIs see [Chapter 5, “CSIX RX”](#) of the *IXA Software Building Blocks Reference Manual*.

42.1 Data flow

CSIX RX core component does not process data packets since it has no exception packets from CSIX RX microblock or neighboring core components (see also assumptions section). Figure 20. shows the logical location of CSIX RX CC (in grey) in relation to other core components and microblocks in an egress IXP2X00 for an Ipv4 forwarding reference application for OC-48 POS.

Figure 42-1. CSIX RX Core Component



42.2 Assumptions and Dependencies

42.2.1 Assumptions

- The CSIX RX core component does not receive any exception packets or messages from the CSIX RX microblock.
- The CSIX RX core component does not receive any packets for other core components. If other core components need to transmit packets to the CSIX fabric, they must send the packets to the Queue Manager.

Because it neither sends or receives packets, the CSIX RX Core Component does not define any input and output for data packets in the system file `bindings.h`.

- The CSIX loop-back switch fabric card does not require any initialization and configuration from CSIX RX Core Component.

42.2.2 Dependencies

The CSIX RX Core Component uses the services of the IXA Portability Framework for:

- allocating memory
- freeing memory
- patching symbols
- 64-bit counter support
- accessing the system registry to retrieve static parameters.

The CSIX RX Core Component uses the Core Components Infrastructure services for:

- message handling

For more information on the IXA Portability Framework and the Core Component Infrastructure, refer to the *Intel[®] Internet Exchange Architecture Portability Framework Developer's Manual*.

42.3 Configuration and Initialization

At initialization time, the CSIX RX Core Component allocates and patches memory and symbols into the microblock. These symbols and memory include:

Table 42-1. CSIX RX Core Component

Symbol	Description
CSIX_RX_CTX_SRAM_BASE (Receive Context in SRAM)	There are 1024 entries and each entry needs 64 bytes so the total memory required from SRAM is 1024 * 64 bytes. The CSIX RX Core Component patches the base address of this memory chunk into the microblock.
CSIX_RX_COUNTERS_SRAM_BASE	This is the base for the counters maintained by the CSIX RX microblock. For management and debugging purposes, there are 64-bit statistics counters allocated using the Resource Manager's 64-bit counter support. Since the CSIX RX microblock is getting c-frames from 1024 VoQs, there is a total of 4*1024 64-bit counters. Memory for these are allocated and only the base address for these counters are patched into the microblock (see the <i>Intel® IXA Building Blocks: Developer's Manual</i>).

42.4 External API

This section summarizes the CSIX RX core component external API. For complete details, see Chapter 5, “CSIX RX” in the *IXA Software Building Blocks Reference Manual*.

42.4.1 Core Component Infrastructure API

Table 42-2 summarizes the Core Component Infrastructure API provided by the CSIX RX Core Component.

Table 42-2. CSIX RX Core Component Infrastructure API

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

42.4.2 Messaging API

Table 42-3 summarizes the Messaging API provided by the CSIX RX Core Component.

Table 42-3. CSIX RX Core Component Messaging API

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

42.4.3 Library API

Table 42-4 summarizes the Library API provided by the CSIX TX core component.

Table 42-4. CSIX RX Core Component Library API

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

43.1 Overview

The Ethernet RX Core Component performs the following functions:

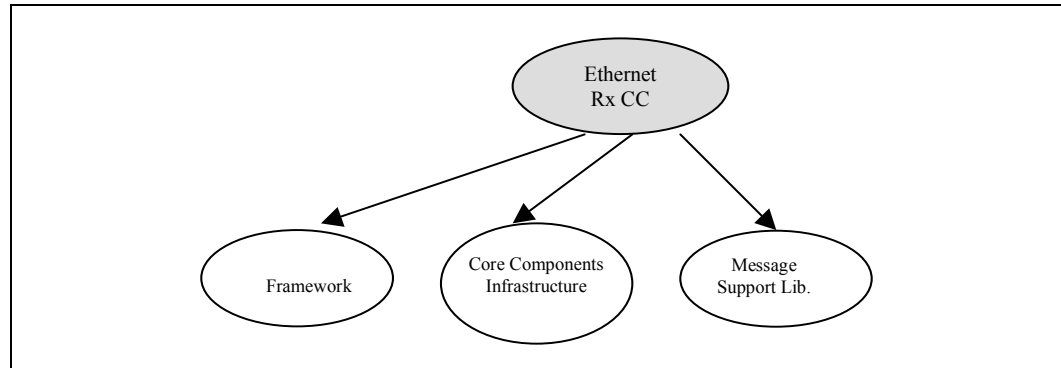
- Configures and initializes the Packet RX microblock
- Receives ARP and other exception packets from the Packet RX microblock
- Sends ARP packets over the PCI Bus to the Ethernet TX Core Component
- Sends other exception packets to another user-defined output core component

For more information on the Packet RX microblock, see [Chapter 4, “Packet RX Microblock”](#) and for external APIs see [Chapter 6, “Ethernet RX”](#) of the *IXA Software Building Blocks Reference Manual*.

43.2 Assumptions and Dependencies

Figure 43-1 illustrates the Ethernet RX Component Dependencies.

Figure 43-1. Ethernet RX Component Dependencies



43.2.1 Assumptions

All Ethernet interface media-related configuration API, such as enabling and disabling the Ethernet interface, must be provided by Ethernet TX Core Component.

43.2.2 Dependencies

The Ethernet RX Core Component uses the services of the IXA Portability Framework for:

- allocating memory
- freeing memory

- patching symbols
- 64-bit counter support
- accessing the system registry to retrieve static parameters.

The Ethernet RX Core Component uses the Core Component Infrastructure services to register its message handler. The Ethernet RX Core Component passes messages to its message handler using the Message Support Library (see [Chapter 63, “Message Helper and Support Library”](#)).

The Ethernet RX Core Component also uses the IXA Portability Framework's system registry to retrieve static parameters. If the system registry is not present, then the default values defined in the component's own header file is used.

43.3 Data flow

43.3.1 Data Input and Output

The Ethernet RX core component defines two packet inputs for receiving exception packets from the Packet RX microblock. The input IDs are defined in the system file, `bindings.h` as follows:

```
#define IX_CC_ETH_MICROBLOCK_HIGH_PRIORITY_PKT_INPUT
#define IX_CC_ETH_MICROBLOCK_LOW_PRIORITY_PKT_INPUT
```

The Ethernet RX core component defines two outputs for sending packets to the Ethernet TX Core Component and other core component. The output IDs for outgoing packets are defined in the system file, `bindings.h`, as follows:

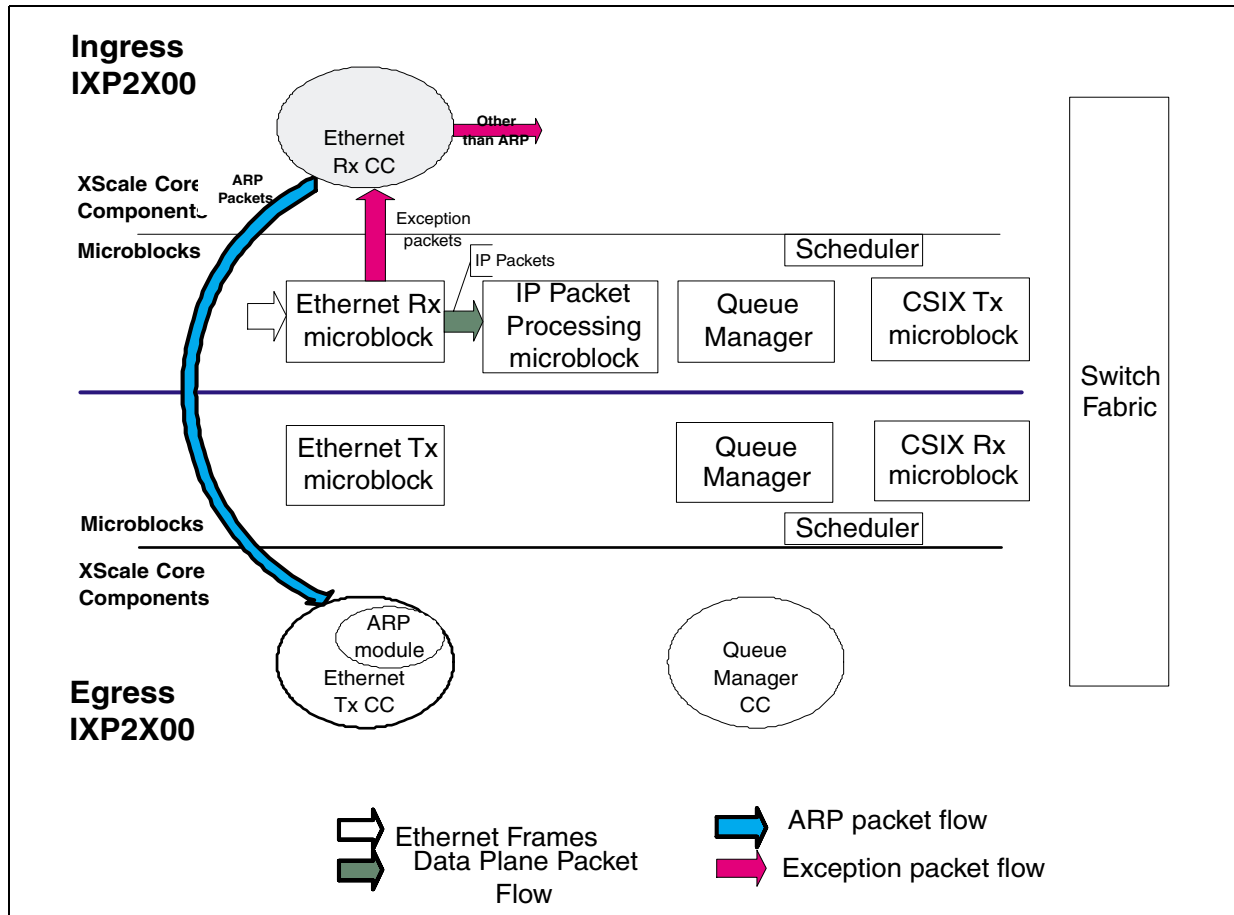
```
#define IX_CC_ETH_RX_ARP_PKT_OUTPUT
#define IX_CC_ETH_RX_COMMON_PKT_OUTPUT
```

The Ethernet RX core component uses API function exposed by Core Component Infrastructure to send out packets (see the *IXA Portability Framework Reference Manual*).

43.3.2 Packet Data Flow

Figure 43-2 illustrates the Ethernet RX Core Component data flow.

Figure 43-2. Ethernet RX Core Component Data Flow



Ethernet RX microblock receives Ethernet frames from Ethernet media. Based on the packet type in the Ethernet header, the microblock forwards the packet to the appropriate component.

Certain types of packets are always treated as an exception by the RX microblock and are sent directly to the Ethernet RX Core Component. For Ethernet, the packets treated as exceptions are:

- ARP
- PPP signaling

If the packet is not an exception packet, Ethernet RX microblock forwards the packet to IP packet processing microblock, IPv4/IPv6.

When the Ethernet RX Core Component receives the exception packet, it checks if the packet is an ARP packet and sends it to the output, `IX_CC_ETH_RX_ARP_PKT_OUTPUT`, which is by default bound to the Ethernet TX Core Component on the Egress side. The packet is forwarded to the Ethernet TX Core Component through PCI bus.

If the packet is another type of exception packet, such as a PPP LCP or IPCP packet, it is forwarded to the other output, `IX_ETH_RX_PKT_OUTPUT`, of the core component. Any application/core component that needs to capture these packets can bind to this output in the system file, `bindings.h`.

43.4 Configuration and Initialization

The Ethernet RX Core Component defines dynamic and static data to be used during initialization. This data must be set properly by the property master and system registry or a component's own header file if the system registry is not present in the system. Some of this data are configuration parameters to be patched into the microblock.

43.4.1 Dynamic Configuration Data

Table 43-1 lists the dynamic properties that are set by the core component's property master at run time.

Table 43-1. Ethernet RX Core Component Dynamic Configuration Data

Data	Default	Property Master	Description
Interface State	UP	Stack Driver	Interface Up or Down

The Ethernet media interface state is set to UP by the Ethernet RX Core Component during initialization. The interface state can be changed at run time by the property master of this data.

43.4.2 Static Configuration Data

The Ethernet RX Core Component defines the following static configuration data:

Table 43-2. Ethernet RX Core Component Static Configuration Data

Data	Default	Description
ETH_RX_MEV2_NUM_MASK	0x00000001 (ME0)	Microengine number to be allocated to Ethernet RX microblock. Only bit [0:7] are valid for ME0 - ME7.
L2_DECAP_MEV2_NUM_MASK	0x66	Microengine number for L2 Decap Microblock
ETH_RX_FLT_TABLE_SIZE	4 Kbytes	SRAM Filter table size
L2_DECAP_PORT_MODE_BIT_MAP	0xffffffff	Port mode bit map. Each bit represents one Ethernet port mode.

The static configuration data is retrieved either from the system registry, if it is present, or from one of the component's header files.

43.4.3 Patching Symbols

Table 43-3 shows the symbols and memory base addresses patched into the Packet RX microblock by Ethernet RX Core Component.

Table 43-3. Symbols and Memory Base Addresses to Patch Packet RX Microblock

Symbols	Default	Description
PACKET_COUNTERS_SRAM_BASE	-	This is the base for the various counters used and maintained by Ethernet RX Microblock for debugging purposes. Various counters used by the Ethernet RX Microblock for debugging are allocated by using the Resource Manager 64-bit counter support.

Note: The microblock counters by default are disabled. To enable the microblock counters for debugging purposes, define the compilation flag MICROBLOCK_COUNTERS.

Table 43-4 shows the symbols and memory base addresses patched into L2 Decap microblock by Ethernet RX Core Component..

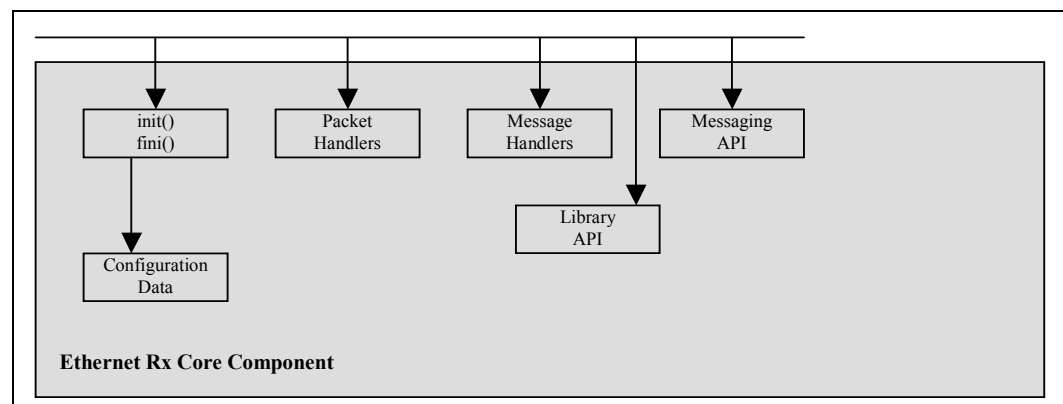
Table 43-4. Symbols and Memory Base Addresses L2 Decap Microblock

Symbols	Default	Description
SRAM_PORT_BIT_MAP_BASE	-	This is the base address for the port mode bit map in SRAM. Each bit represents one Ethernet port mode. Bit definitions are :0 - the port is defined as unicast mode1 - the port is defined as promiscuous mode
SRAM_FLT_TABLE_BASE		MAC filtering table base address

43.5 Modularity

Figure 43-3 illustrates the logical sub-modules in Ethernet RX core component.

Figure 43-3. Modularity of Ethernet RX Core Component



The init function of the component initializes all dynamic and static configuration data. The component defines packet inputs that associate with the packet handler functions for receiving exception packet from the Ethernet RX microblock. The Messaging API is called by level-1 based system application. It translates calls into messages and passes them to level-1 Framework. The

level-1 Framework sends these messages to the message inputs of the component. The message inputs defined by the component are for handling statistics and interface status related messages. The message handler functions call appropriate library API to process the message.

43.6 External API

This section summarizes the Ethernet RX core component external API. For complete details, see Chapter 6, “Ethernet RX” in the *IXA Software Building Blocks Reference Manual*.

43.6.1 Core Component Infrastructure API

Table 43-5 summarizes the Ethernet RX core component provided Core Component Infrastructure API.

Table 43-5. Ethernet RX Core Component Functions

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

43.6.2 Messaging API

Table 43-6 summarizes the Messaging API provided by the Ethernet RX core component.

Table 43-6. Ethernet Interface RX Messaging Functions

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

43.6.3 Library API

Table 43-7 summarizes the Library API provided by the Ethernet RX core component.

Table 43-7. Ethernet Interface RX Library Functions

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

Transmit Components

The Transmit core components include the following:

- [Chapter 44, “CSIX TX Core Component”](#)
CSIX TX Core Component corresponds to CSIX TX microblock and performs the initialization and configuration for the CSIX TX microblock through patching of the imported symbols and memory block into the microblock.
- [Chapter 45, “ATM/POS TX Core Component”](#)
POS TX Core Component performs the initialization and configuration for POS TX microblock through patching imported symbols and memory blocks into the microblocks.
- [Chapter 46, “Ethernet TX Core Component”](#)
Ethernet TX Core Component performs the following functions:
 - Initialization and Configuration of Ethernet TX microblock
 - Configuration of multi-port Gigabit Ethernet media card
 - Handling exception packets for ARP processing and resolution of layer2 address
- [Chapter 47, “Ethernet ARP Module”](#)
Ethernet ARP module - This module implements the basic ARP protocol outlined in RFC826 and additional mandatory rules specified in RFC1122:
 - ARP cache
 - Handling of ARP packet generation and reception
 - ARP cache time-out
 - Prevention of ARP flooding
 - ARP packet queue
 - Generation and handling of gratuitous ARP request.
 - Detection of IP address duplication

44.1 Overview

The CSIX TX core component performs the following functions:

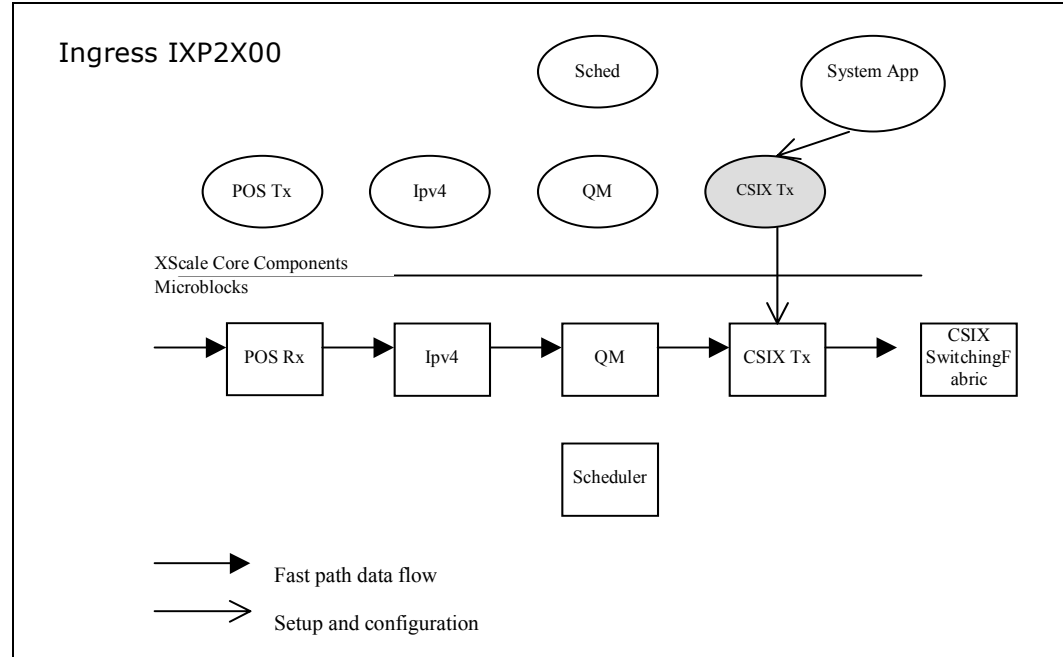
- Initializes and configures of the CSIX TX microblock
- Provides an interface to a system application for retrieving statistical parameters

For more information on the CSIX TX microblock, see [Chapter 7, “CSIX TX Microblock”](#) and for external APIs see [Chapter 7, “CSIX TX”](#) of the *IXA Software Building Blocks Reference Manual*.

44.2 Data flow

CSIX Tx core component does not process data packets since it has no exception packets from CSIX Tx microblock or neighboring core components (see also assumptions section). The following figure shows the logical location of CSIX Tx CC (in grey) in relation to other core components and microblocks in an ingress IXP2X00 for an Ipv4 forwarding application for OC-48 POS.

Figure 44-1. CSIX TX Core Component



44.3 Assumptions and Dependencies

44.3.1 Assumptions

- The CSIX TX core component does not receive any exception packets or messages from the CSIX TX microblock.
- The CSIX TX core component does not receive any packets for other core components. If other core components need to transmit packets to the CSIX fabric, they must send the packets to the Queue Manager.
Because it neither sends or receives packets, the CSIX TX core component does not define any input and output for data packets in the system file `bindings.h`.
- Unlike the POS TX core component, the CSIX loop-back switch fabric card does not require any initialization and configuration from CSIX TX core component.

44.3.2 Dependencies

The CSIX TX core component uses the services of the IXA Portability Framework for:

- allocating memory
- freeing memory
- patching symbols
- 64-bit counter support
- accessing the system registry to retrieve static parameters
- message handling

The CSIX TX core component uses the Core Components Infrastructure services for:

- message handling

For more information on the IXA Portability Framework and the Core Component Infrastructure, refer to the *Intel[®] Internet Exchange Architecture Portability Framework Developer's Manual*.

44.4 Configuration and Initialization

The CSIX TX core component requires the following configuration parameters from the system registry at initialization time:

- Ingress blade id (default 0x00000001, blade 1)
- MEv2 number (default 0x00000080, ME7)

If the system registry is used, the CSIX TX core component must define this parameter in one its header files.

During initialization, the CSIX TX core component allocates and patches memory and symbols to be used by the CSIX TX microblock.

Table 44-1. Patch Symbols for CSIX TX Core Component

Symbol	Description
THIS_BLADE_ID (Ingress Blade id information (32 bits))	A total of 64 blades are assumed so the valid bits are bits 0 - 5.
CSIX_TX_CTX_SRAM_BASE (Transmit Context in SRAM)	There are 1024 entries and each entry needs 64 bytes, so the total memory required from SRAM is 1024 * 64 bytes. The CSIX TX core component patches the base address of this memory chunk into the microblock.
CSIX_TX_COUNTERS_SRAM_BASE	This is the base for the counters maintained by the CSIX TX microblock. For management and debugging purposes, there are 64-bit statistics counters allocated using the Resource Manager's 64 bit counter support. Since the CSIX TX microblock gets c-frames from 1024 VoQ, there is a total of 1024 64-bit TX counters and 1024 64-bit drop counters be allocated (see <i>Intel® IXA Building Blocks: Developer's Manual</i>).

44.5 External API

This section summarizes the CSIX TC core component external APIs. For complete details, see [Chapter 7, “CSIX TX”](#) in the *IXA Software Building Blocks Reference Manual*.

44.5.1 Core Component Infrastructure API

Table 44-2 summarizes the Core Component Infrastructure API for the CSIX TX core component.

Table 44-2. CSIX TX Core Component Infrastructure API

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

44.5.2 Messaging API

Table 44-3 summarizes the Messaging API for the CSIX TX core component.

Table 44-3. CSIX TX Core Component Messaging API

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

44.5.3 Library API

Table 44-4 summarizes the Library API for the CSIX TX core component.

Table 44-4. CSIX TX Core Component Library API

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

45.1 Overview

The ATM/POS TX Core Component combines the configurations of ATM TX and POS TX.

The ATM/POS TX Core Component performs the following functions:

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

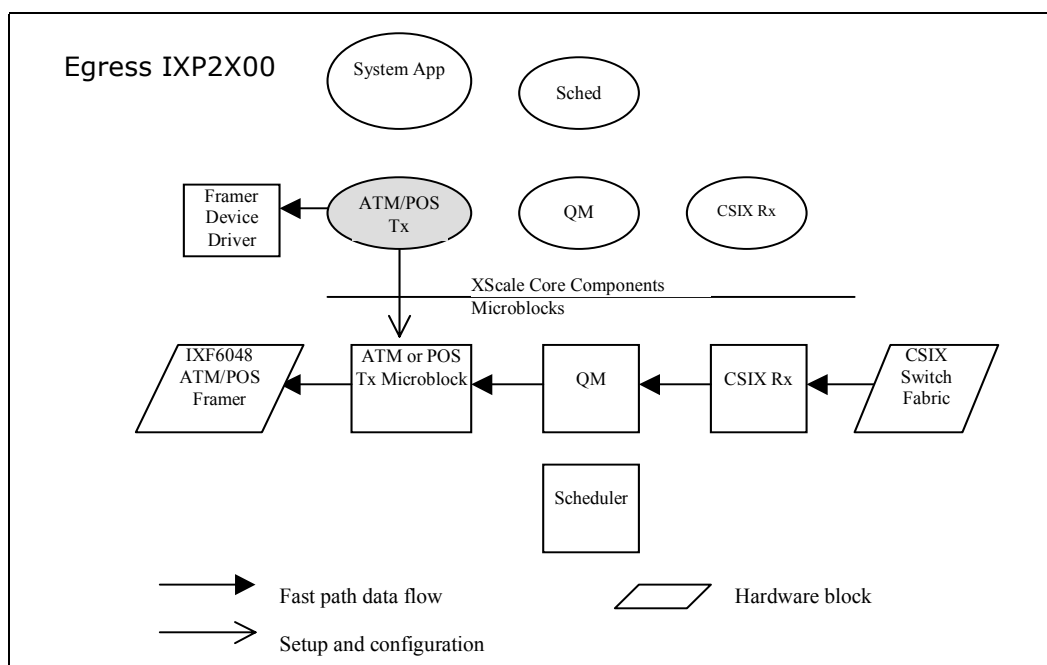
There are distinct microblocks for MPHY-16 and for SPHY and MPHY-4. For information on these microblocks, as well as the ATM TX microblock, see [Chapter 9, “Packet TX for MPHY-16,”](#) [Chapter 8, “Packet TX for SPHY and MPHY-4,”](#) and [Chapter 11, “ATM AAL5 TX Microblock.”](#) For external APIs see [Chapter 8, “ATM/POS TX”](#) of the *IXA Software Building Blocks Reference Manual*.

Note: The support for the ATM TX part of the core component is described in , [Section 65.10, “ATM TX Core Component”](#) on page 948 of [Chapter 65, “SoftSAR Core Components”](#) of this document.

45.2 Data flow

ATM/POS TX core component does not process data packets since it has no exception packets from POS TX microblocks and neighboring core components. [Figure 45-1](#) illustrates the logical location of ATM/POS TX core component (in grey) in relation to other core components and microblocks in an egress IXP2X00 for a forwarding application for OC-48.

Figure 45-1. ATM/POS TX Core Components



45.3 Assumptions and Dependencies

45.3.1 Assumptions

The following assumptions are made:

- The ATM/POS TX Core Component assumes that neither the ATM TX microblock nor the POS TX microblock send any exception packets and messages to the core.
- The neighboring core components cannot send any packets to the ATM/POS Core Component for transmitting out to the ATM/POS TX framer. If the neighboring core components need to transmit packets to the framer, they must be sent to the Queue Manager. Because it neither receives nor sends packets, the ATM/POS Core Component does not define any data input and output in the system file `bindings.h`.
- The ATM/POS TX Core Component provides a configuration API for ATM/POS TX interface media-related functions, such as enabling and disabling the ATM or POS interface. The initialization of the ATM and POS framer are also done by ATM/POS TX core component.

45.3.2 Dependencies

The ATM/POS TX Core Component uses the services of the IXA Portability Framework for:

- allocating memory
- freeing memory

- patching symbols
- 64-bit counter support
- accessing the system registry to retrieve static parameters.
- registering the message handler

The ATM/POS TX Core Component uses the Core Components Infrastructure services for to register its message handler. The ATM/POS TX Core Component passes messages to its message handler using the Message Support Library (see [Chapter 63, “Message Helper and Support Library”](#)).

The ATM/POS TX Core Component uses IXA Portability Framework's system repository to retrieve static parameters. Use of the system repository is implemented by using global definitions (such as `#define IX_INCLUDE_REGISTRY 1`) so that when the repository is not included in user's application, the ATM/POS TX Core Component can obtain the static data from compile time defined values, such as a header file.

The ATM/POS TX Core Component utilizes the API provided by the framer device driver to initialize and configure the IXF6048 ATM/POS TX framer device based on the static configuration data obtained from either the system repository or the component's global header file, `ix_cc_atm_pos_tx.h`.

45.4 Configuration and Initialization

The ATM/POS TX Core Component defines dynamic and static data to be during initialization. For dynamic properties where the ATM/POS TX Core Component is the master, the data must be set by the System Application. For other data, the data must be set using the system repository or if the system registry is not present then, for static data, by one of the core component's header files. This data includes configuration parameters to be patched into the microblock.

45.4.1 Dynamic Configuration Data

The dynamic property shown in [Table 45-1](#) can be set at run time:

Table 45-1. ATM/POS TX Core Component Dynamic Configuration Data

Data	Default	Property Master	Description
Interface State	UP	ATM/POS TX	Interface Up or Down

The ATM or POS interface state can be changed at run time by the Stack Driver.

45.4.2 Static Configuration Data

Table 45-2 shows the The ATM/POS TX core component defined static configuration data.. This data is obtained from the system repository during the core component's initialization. If the system repository is not present, then the default values defined in the component's global header file, `ix_cc_atm_pos_tx.h`, are used. This data includes configuration parameters to be patched into the microblocks.

Table 45-2. ATM/POS Interface TX Configuration Parameters

Data	Default	Description
ATM_TX_MEV2_NUMBER	0x0000060 (ME5, ME6)	Microengine numbers to be allocated to ATM TX microblock. Only bit [0:7] are valid for ME0 - ME7. This data is required only when ATM mode is selected.
POS_TX_MEV2_NUMBER	0x0000010 (ME4)	Microengine numbers to be allocated to POS TX microblock. Only bit [0:7] are valid for ME0 - ME7. This data is required only when POS mode is selected.

45.4.3 ATM/POS Media Card Operation Mode

The ATM/POS TX Core Component configures the framer media card based on the operation mode selected. The operation mode is selected at compile time through the use of the following compilation flags and `#define` codes in the component global header file, `ix_cc_atm_pos_tx.h`:

```
#if defined(IX_CONFIGURATION_oc48_sphy_pos_egress)
#define ATM_POS_OPERATION_MODES      0x000000C0 /* POS mode 0 */
#elif defined(IX_CONFIGURATION_oc12_sphy4_pos_egress)
#define ATM_POS_OPERATION_MODES      0x000010C1 /* POS mode 1 */
#elif defined(IX_CONFIGURATION_oc12_mphy4_pos_egress)
#define ATM_POS_OPERATION_MODES      0x000021C1 /* POS mode 2 */
```


The definition of each bit in the operation mode value is described in [Table 45-3](#).

Table 45-3. ATM/POS Media Card Operation Mode

Data	Default	Description
ATM_POS_OPERATION_MODE	0x000010C1 (Default is for OC-48 mode, 1 x 16-bit line side interface, POS mode with 4 x 8-bit POS-PHY interface)	<p>This is a 32-bit data that contains various options for operation modes to be configured in the ATM/POS framer device. Refer to Table 45-4 for the supported line side interface and ATM/POS Utopia interface for each OC mode supported by the IXD2448 Single OC-48 I/O Option Card and the IXD2412 Quad OC-12 I/O Option Card.</p> <p>This version of the software does not support combining of ATM and POS modes. All interfaces—single or quad—must use the same mode.</p> <p>Bit [31:16]: Reserved (must be all 0s)</p> <ul style="list-style-type: none"> Bit [15:12] and OC Mode <ul style="list-style-type: none"> 0000—OC-48c 0001—OC-48 0010—OC-12c 0011—OC-12 All others—Reserved Bit [11:8] and Line Side Interface <ul style="list-style-type: none"> 0000—1 x 16-bit PECL 0001—4 x 8-bit LVTTTL 0010—1 x 8-bit LVTTTL All others—Reserved Bit [7:6] and Channel Mode <ul style="list-style-type: none"> 00—Reserved 01—Reserved 10—ATM Mode 11—POS Mode Bit [5:0] and ATM Utopia or POS-PHY interface <ul style="list-style-type: none"> 000000—1 x 32-bit (Proprietary L2) 000001—4 x 8-bit (L1) 000010—1 x 32-bit (L3) 000011—4 x 8-bit (L3) All others—Reserved

Table 45-4 lists the allowed line side interface and the ATM/POS TX media interfaces supported by the IXD2448 Single OC-48 I/O Option Card and the IXD2412 Quad OC-12 I/O Option Card media cards for each OC mode.

Table 45-4. Supported Line Side Interface and Media Interface for each OC Mode

Media Card	Optical Carrier Level	Line Side Interface	ATM/POS TX Utopia Interface	
			ATM-Utopia	POS-PHY
IXP2400	OC-48c (Single)	1 x 16-bit PECL	1 x 32-bit (L3)	1 x 32-bit (Proprietary L2)
	OC-48 (Single)	1 x 16-bit PECL	4 x 8-bit (L3)	4 x 8-bit (L1)
IXP2400	OC-12c (Quad)	4 x 8-bit LVTTTL	1 x 32-bit (L3) or 4 x 8-bit (L1 or L3)	1 x 32-bit (Proprietary L2) ¹ or 4 x 8-bit (L1)
	OC-12 (Single)	1 x 8-bit LVTTTL	1 x 32-bit (L3) or 4 x 8-bit (L1 or L3)	1 x 32-bit (Proprietary L2) ¹ or 4 x 8-bit (L1)

1. Not recommended for IXP2400/IXF6048 comb in at on

45.4.4 Patching Symbols

The ATM/POS TX Core Component patches symbols and memory base addresses into its microblocks during initialization time. The following two tables show this data. ATM related symbols are patched into the ATM TX microblock only when ATM mode is selected. Similarly, POS related symbols are patched into the POS TX microblock only when POS mode is selected.

Table 45-5. Symbols and Memory Base Addresses Patched into ATM TX Microblock

Symbols Patched into ATM TX Microblock	Description
ATM_TX_COUNTERS_BASE	<p>For management and debugging purpose, ATM/POS TX Core Component maintains four types of 64-bit counters on either a per VCQ basis (total 64K VCQs reserved) or a per port basis (total 16 ports reserved).</p> <p>These include:</p> <ul style="list-style-type: none"> counters for packets transmitted per VCQ, cells transmitted per VCQ, packets transmitted per port, cells transmitted per port. <p>ATM/POS TX uses Resource Manager 64-bit counter API to allocate memories for both these 64-bit counters and their 32-bit version of the counters. ATM/POS TX patches the base address of the memory of 32-bit counters, with total size of:</p> $(64K * 2 * 4) + (2048 * 2 * 4) = 528 \text{ Kbytes, into the ATM TX microblock.}$ <p>Counter and Offset from Base for Port:</p> <ul style="list-style-type: none"> VCQ Counters <ul style="list-style-type: none"> Packets Transmitted—(VCQ 0)—0x0 Cells Transmitted—(VCQ 0)—0x4 Port Counters Packets <ul style="list-style-type: none"> Transmitted—(port 0)—(64K * 2 * 4) Cells Transmitted—(port 0)—(64K * 2 * 4) + 4

Table 45-5. Symbols and Memory Base Addresses Patched into ATM TX Microblock

Symbols Patched into ATM TX Microblock	Description
ATM_TX_TXC	Transmit Context - There is 8 LWs allocated for each VCQ used by ATM microblock. Since the ATM microblock supports 64K VCQs, the total memory required from SRAM is $(64K * 8) = 512$ K LW. ATM/POS TX Core Component patches the base address of this memory.

Table 45-6. Symbols and Memory Base Addresses Patched into POS TX Microblock

Symbols Patched into POS TX Microblock	Description
PACKET_TX_COUNTERS_BASE	<p>For management and debugging purpose, ATM/POS TX Core Component maintains four types of 64-bit counters on a per port basis. These include counters for the following:</p> <ul style="list-style-type: none"> packets transmitted packets dropped bytes transmitted bytes dropped. <p>ATM/POS TX uses Resource Manager 64-bit counter API to allocate memories for both these 64-bit counters and their 32-bit version of the counters. ATM/POS TX patches the base address of the memory of 32-bit counters, with total size of $(16 * 4 * 4) = 256$ bytes for 16 ports, into the POS TX microblock.</p> <p>Counter Offset from base for Port:</p> <ul style="list-style-type: none"> Packets Transmitted—0x0 Packets Dropped—0x4 Bytes Transmitted—0x8 Bytes Dropped—0xc
PACKET_TX_NEXT_AVAIL_TBUF_ELE_SCR_ADDR	The core component needs to allocate this memory space from scratch memory for being used internally by the POS TX microblock to manage its next available TBUF element

45.5 External API

This section summarizes the ATM/POS TX core component external API. For complete details, see [Chapter 8, “ATM/POS TX”](#) in the *IXA Software Building Blocks Reference Manual*.

45.5.1 Core Component Infrastructure API

[Table 45-7](#) summarizes the elements of the core component infrastructure API provided by the ATM/POS TX Core Component.

Table 45-7. ATM/POS TX Core Component Infrastructure API

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

45.5.2 Messaging API

[Table 45-8](#) summarizes the elements of the messaging API provided by ATM/POS TX Core Component. For more detailed information, refer to the *IXA Software Building Blocks Reference Manual*.

The messaging API is implemented in the core component message helper library, `ccmsg_hlp.lib`. It interfaces with the message handler defined in the infrastructure API indirectly through Message Support Library (see [Chapter 63, “Message Helper and Support Library”](#)).

Table 45-8. ATM/POS TX Core Component Messaging API

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

45.5.2.1 Library API

The library API provided by ATM/POS TX Core Component is summarized in [Table 45-9](#). For more detailed information, refer to the *IXA Software Building Blocks Reference Manual*.

Table 45-9. ATM/POS TX Core Component Library API

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

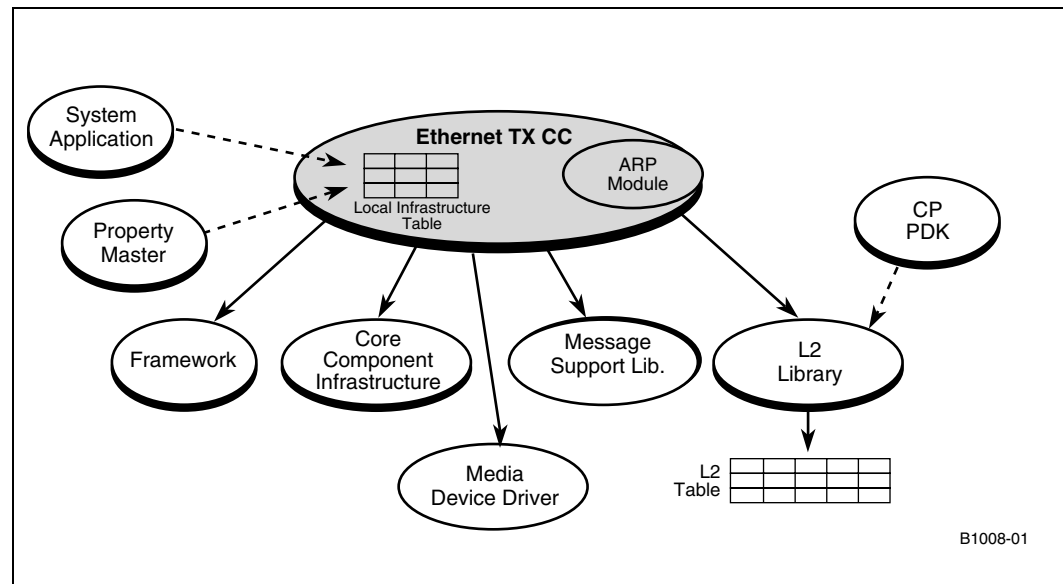
46.1 Overview

The Ethernet TX core component performs transmit functions for the Ethernet Interface. This chapter provides an overview of the Ethernet TX core component. For external APIs see [Chapter 9](#), “Ethernet TX” of the *IXA Software Building Blocks Reference Manual*.

46.2 Assumptions and Dependencies

Figure 46-1 illustrates Ethernet TX Component Dependencies.

Figure 46-1. Ethernet TX Component Dependencies



46.2.1 Assumptions

- The Ethernet TX core component assumes that all Ethernet interface media-related configuration APIs, such as enabling and disabling the Ethernet interface, are provided by the Ethernet RX core component.

The Ethernet RX core component is initialized first by the System Application prior to the initialization of the Ethernet TX core component. Initialization of the Ethernet media ports is also done by the Ethernet RX core component.

46.2.2 Component Dependencies

The Ethernet RX core component uses the services of the IXA Portability Framework for:

- allocating memory
- freeing memory
- patching symbols
- 64-bit counter support
- accessing the system registry to retrieve static parameters.

The Ethernet RX core component uses the Core Component Infrastructure services for to register its message handler. The Ethernet RX core component passes messages to its message handler using the Message Support Library (see [Chapter 63, “Message Helper and Support Library”](#)).

The Ethernet TX core component creates the Local Interface Table. The property master must set the port IP addresses in the table. The System Application must set the port layer-2 address in the table.

The Ethernet TX core component creates the L2 table by calling the `init` function of the L2 library (see [Chapter 62, “L2 Table Manager”](#)). The ARP module of the Ethernet TX core component requires the next hop ID information to be set by the Control Plane PDK in the L2 table. The detailed dependencies of the ARP module are described separately in [Chapter 47, “Ethernet ARP Module.”](#)

46.3 Data flow

46.3.1 Packet Input and Output

Ethernet TX core component defines two packet inputs for receiving exception packets from Ethernet TX microblock and Ethernet RX core component. The input IDs are defined in system file, `bindings.h` as follows:

```
#define IX_CC_ETH_TX_ARP_PKT_INPUT
#define IX_CC_ETH_TX_UBLK_PKT_INPUT
```

Ethernet TX core component sends packets to the outputs it defines. The output id for outgoing packets is defined in system's `bindings.h` as follows:

```
#define IX_CC_ETH_TX_ARP_PKT_OUTPUT
#define IX_CC_ETH_TX_COMMON_PKT_OUTPUT
```

Ethernet TX core component uses API function exposed by Resource Manager to send packets.

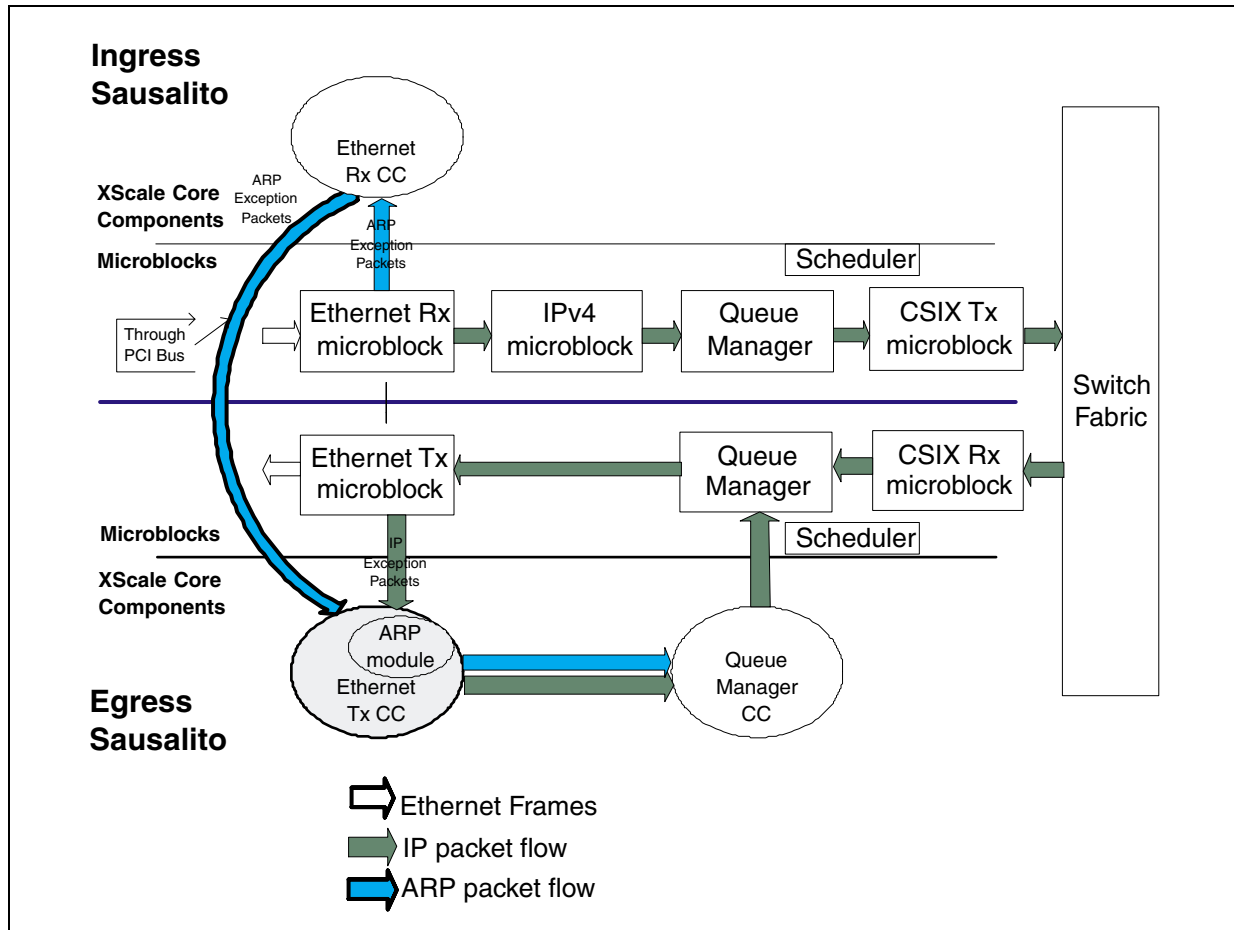
Ethernet TX core component defines the following message output for sending:

```
#define IX_CC_ETH_TX_FP_MODULE_MSG_OUTPUT
```

46.3.2 Packet Data Flow

Figure 46-2 illustrates Ethernet TX Core Component data flow.

Figure 46-2. Ethernet TX Core Component Data Flow



The Ethernet TX core component receives packets from one of two sources listed in Table 46-1.

Table 46-1. Ethernet TX Core Component Input Sources

Source	Reason
Ethernet RX core component	ARP Packets received from the Ethernet RX microblock are sent as exceptions to the Ethernet TX core component.
Ethernet TX microblock	Packets where the Layer-2 address cannot be found by the Ethernet TX microblock are sent as exceptions to the Ethernet TX core component.

When an exception packet reaches the Ethernet TX core component, it is processed as follows:

1. The packet is handed over by the Ethernet TX core component to its internal ARP module.
2. If the exception packet is an ARP packet, the result of this processing is that either an ARP reply packet or a previously held IP packet is sent to the Queue Manager core component.

3. If the exception packet is an IP packet, the result is an ARP request packet being sent to the Queue Manager core component.
4. Regardless of its type, the output packet is eventually sent to Ethernet TX microblock for transmission to the media.

All packets to be sent out to the Ethernet media are encapsulated with Ethernet header [RFC 894 and 1042]. For held IP packets, the encapsulation is performed by the Ethernet TX microblock. For ARP packets, the encapsulation is done by the ARP module of the Ethernet TX core component before the packets are sent to the Queue Manager core component. The next hop id field in the metadata of ARP packets is set to indicate that the packet is a non-IP packet. This is done so that the Ethernet TX microblock can easily identify these packets.

46.4 Configuration and Initialization

The Ethernet TX core component defines certain dynamic and static data to be used during its initialization. This data is either configuration parameters to be used in the ARP handling by its ARP module or symbols to be patched into the microblock. The data must be set properly by the property master, or the system registry, or be defined in the component own header file if the system registry is not present in the system.

46.4.1 Dynamic Configuration Data

Table 46-2 shows the dynamic property set by their property master at run time.

Table 46-2. Dynamic Configuration Data

Data	Default	Property Master	Description
IP address	-	Stack Driver	Interface port IP address
MAC address	-	Stack Driver	Interface port MAC address
Interface State	UP	Stack Driver	Interface Up or Down
Link Speed	-	Stack Driver	Interface speed (10M/100M or Gigabit)

The port IP address is an essential data required in the ARP handling. When the port IP address is set or changed, the Ethernet TX core component is notified by the IP port property master of this shared property (see [Chapter 64, “Stack Driver”](#)). The Ethernet TX core component stores the new IP addresses in the Local Interface Table created by the component initialization routine and, implements a message handler to receive and process the property updates (see [Chapter 63, “Message Helper and Support Library”](#)).

The Ethernet media interface state is set to UP by Ethernet TX core component at the initialization time. The interface state can be changed at run time by the property master of this data.

Ethernet TX core component receives notice of an update to these property from the Stack driver, and implements a message handler and API functions (see also [Section 46.5, “External API” on page 736](#)) to receive and process this property updates.

46.4.2 Static Configuration Data

The Ethernet TX core component defines the following static configuration data:

Table 46-3. Ethernet TX Core Component Static Configuration Data

Data	Default	Description
ETH_TX_MEV2_NUM_MASK	0x00000070 (ME4,ME5 and ME6)	Microengine number to be allocated to the Ethernet TX microblock. Only bit [0:7] are valid for ME0 - ME7.

The static configuration data is retrieved from either the system registry, if it is present, or from one of the core component's header files.

46.4.3 Table Creations

During initialization, the Ethernet TX core component is responsible for creating the following tables:

- Local Interface Table
- L2 Table
- ARP Cache

46.4.3.1 Local Interface Table

The Local Interface Table contains all IP addresses and layer-2 addresses for network ports on the blade. These are dynamic properties and need to be set and updated by the property master and system application through the API exposed by the Ethernet TX core component. The table is used internally by the Ethernet TX core component and it serves two purposes:

- It is looked-up by the internal ARP module for matching the target IP address of incoming ARP packets
- Any change in the layer-2 address in the table by the property master triggers a gratuitous ARP packet to be sent out by the ARP module through the Ethernet TX core component.

46.4.3.2 L2 Table

The L2 Table is a shared table between the Ethernet TX microblock and the Ethernet TX core component. The Ethernet TX core component creates the table by calling the `create` routine of the L2 library. The table contains layer-2 address information and is indexed by next hop id. It is updated by both the Control Plane PDK and the ARP module. The Control Plane PDK creates the next hop id and the ARP module fills in the L2 header info fields of the table. The microblock looks in the table for the layer-2 address for composing the layer-2 header when transmitting IP packet to the Ethernet media.

46.4.3.3 ARP Cache

The ARP cache is created by the ARP module of the Ethernet TX core component at initialization time. It contains next hop IP and layer-2 address information. It is used internally by the ARP module.

46.4.4 Patching Symbols

The Ethernet TX core component patches the following symbols and memory base addresses into Ethernet TX microblock:

Table 46-4. Symbols and Memory Base Addresses to be Patched

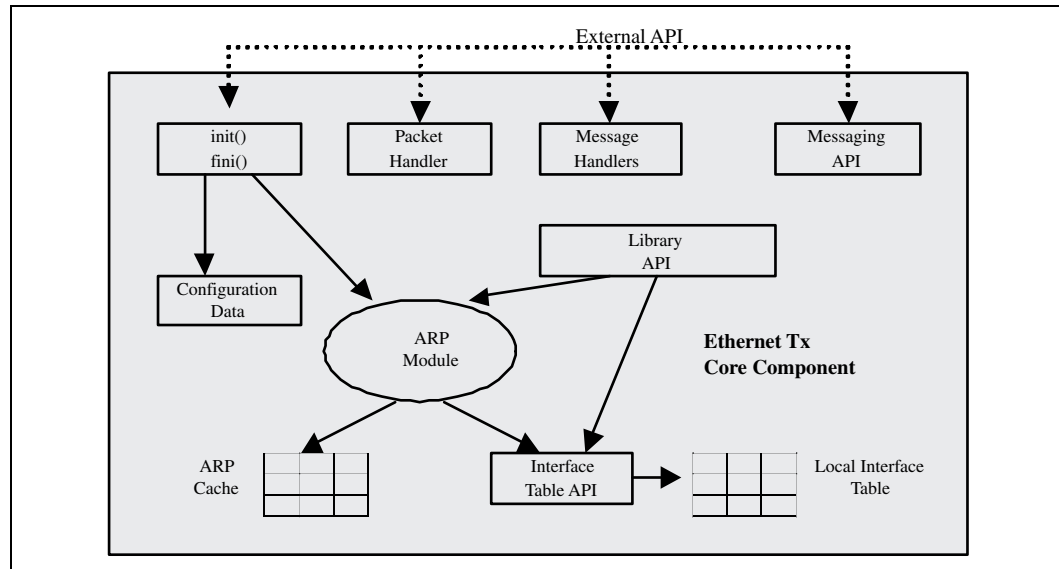
Symbols	Description
L2_TABLE_SRAM_BASE	Base address of the L2 table. The L2 library returns this base address to Ethernet TX core component in the invocation of the <code>init</code> routine of L2 library.
ETH_TX_COUNTERS_BASE	<p>For management and debugging purpose, the Ethernet TX core component maintains a set of four 64-bit counters for each of the ports—where the total number of ports is <code>ETH_TX_NUM_PORTS</code>. Each set of four counters includes counters for packets transmitted, packets dropped, bytes transmitted, and bytes dropped. Ethernet TX uses the Resource Manager 64-bit counter API to allocate memory for both 64-bit counters and for 32-bit versions of them. Ethernet TX patches into the microblock the base address of the memory of 32-bit counters. The total size allocated is <code>ETH_TX_NUM_PORTS</code> times four bytes per counter times four kinds of counters.</p> <p>Counter Offset from Base to Port</p> <ul style="list-style-type: none"> • Packets Transmitted—<code>0x0</code> • Packets Dropped—<code>0x4</code> • Bytes Transmitted—<code>0x8</code> • Bytes Dropped—<code>0xC</code>

Note: The microblock counters by default are disabled. To enable the microblock counters for debugging purposes, define the compilation flag `MICROBLOCK_COUNTERS`.

46.4.5 Modularity

The following figure shows the logical sub-modules in Ethernet TX core component.

Figure 46-3. Modularity of Ethernet TX Core Component



The `init` function of the component initializes all the dynamic and static configuration data. This also includes the initialization of the ARP module and creation of Local Interface Table and L2 Table. The component defines a packet input for receiving exception packets. The packet handler function associated with this input receives exception packet from clients and passes them to the ARP module for processing. The Messaging API is called by the system application. It translates calls into messages and passes them to Core Component Infrastructure. The Core Component Infrastructure sends these messages to the message inputs of the component. There are two message inputs defined by the component. One is to handle ARP and statistics related messages and the other is to handle port property related messages. These message handler functions call appropriate library API to process the message. The Library API accesses the ARP module for ARP related processing or Interface Table API for port property related operations. The ARP module updates its ARP cache accordingly based on the type of the operation. The Interface Table API is an internal module that provides the interface for accessing the Local Interface Table.

46.5 External API

This section summarizes the Ether TX core component. For complete details, see [Chapter 9](#), “Ethernet TX” in the *IXA Software Building Blocks Reference Manual*.

46.5.1 Data Structures

The data structures listed in this section and summarized in [Table 46-5](#) are defined by the core component

Table 46-5. Ethernet TX Core Component Data Structures

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

46.5.2 Core Component Infrastructure API

[Table 46-6](#) summarizes the Core Component Infrastructure API provided by Ethernet TX core component.

Table 46-6. Ethernet TX Core Component Infrastructure API

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

46.5.3 Messaging API

Table 46-7 summarizes the Messaging API provided by the Ethernet TX core component..

Table 46-7. Ethernet TX Messaging Functions

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

46.5.4 Library API

Table 46-8 summarizes the Library API provided by the Ethernet TX core component..

Table 46-8. Ethernet TX Library API

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

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

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

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

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

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

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

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

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

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

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

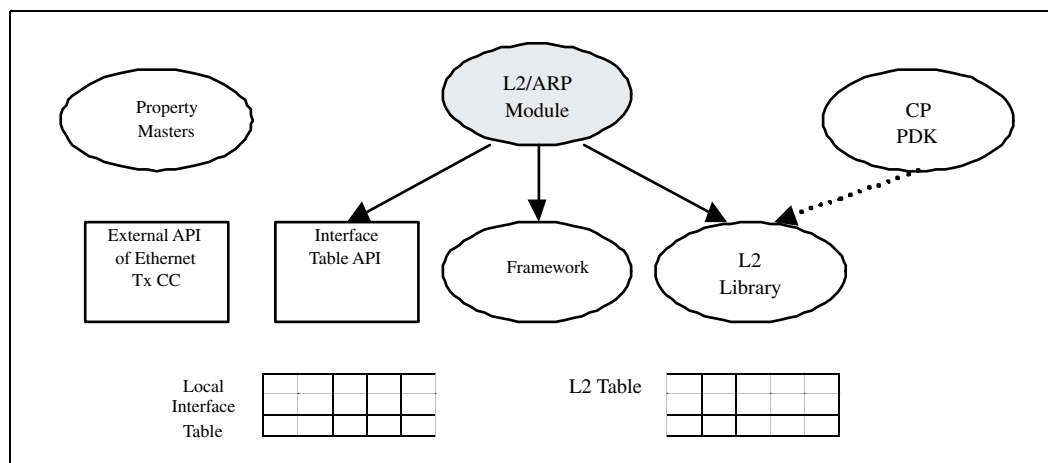
- Creation of static ARP entries
- Deletion of ARP entries
- ARP cache purging
- Printing ARP cache

For more information on the Ethernet TX Core Component, see [Chapter 46, “Ethernet TX Core Component”](#) and for external APIs see [Chapter 10, “Ethernet ARP Module”](#) of the *IXA Software Building Blocks Reference Manual*.

47.1 Assumptions and Dependencies

Figure 47-1 illustrates the Ethernet ARP Component Dependencies.

Figure 47-1. Ethernet ARP Component Dependencies



47.1.1 Assumptions

- The Ethernet TX Core Component is assumed to be the only component that can directly access the APIs exposed by this library. Other clients wishing to access the library should call the associated messaging API or library API provided by the Ethernet TX Core Component.
- The Ethernet ARP module assumes Ethernet II/RFC894 environment
- The Ethernet ARP module relies on the next hop identifier for performing lookups into the L2 table and its ARP cache. It assumes the next hop ID is present in all IP packets passed into the ARP module. Packets without a valid next hop ID embedded in the packet meta-data is dropped by the ARP module.

47.1.2 Dependencies

The Ethernet ARP module utilizes the following components and properties:

- Resource Manager
The ARP module accesses the services of the IXA Portability Framework for operations such as allocation of `ix_buffer` for generation of ARP packets.
- Local Interface Table
This table contains the IP address and layer-2 address information for network ports on the blade. It is created by the Ethernet TX Core Component during initialization and is updated at run-time by the property master and System Application, which use the external API of the Ethernet TX Core Component.
The ARP module performs lookups in the table through the Interface Table API, an internal module of the Ethernet TX Core Component, to determine the target IP address for received ARP packets. In addition, any change of the layer-2 address in the table by the System Application triggers a gratuitous ARP packet to be sent out by the ARP module through the Ethernet TX core component.

For more information on the Ethernet TX Core Component, see [Chapter 46, “Ethernet TX Core Component.”](#)

- L2 library

The L2 library provides an interface for accessing the L2 table. The L2 table is indexed by next hop ID and contains the layer-2 address information for each next hop IP discovered by the system.

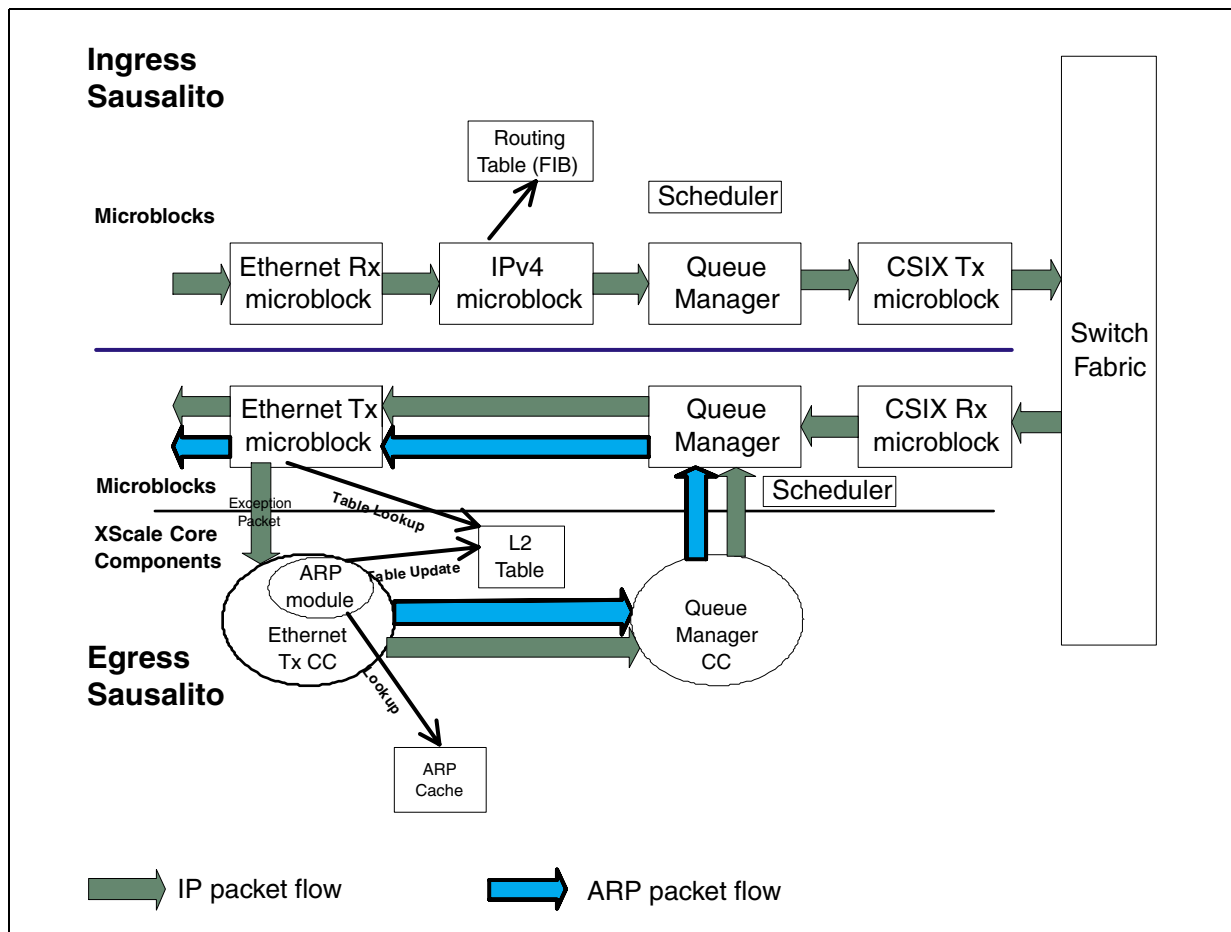
The table is created by the Ethernet TX Core Component and is updated by both the Control Plane PDK and the ARP module. The Control Plane PDK creates the next hop ID. The ARP module then fills in the L2 header info fields of the L2 table so that the Ethernet TX microblock can perform a lookup for outgoing IP packets. Therefore, the Control Plane PDK must be present in order to use ARP handling.

47.2 Data flow

47.2.1 ARP Generation Data Flow

Figure 47-2 illustrates Ethernet ARP generation data flow.

Figure 47-2. Ethernet ARP Generation Data Flow



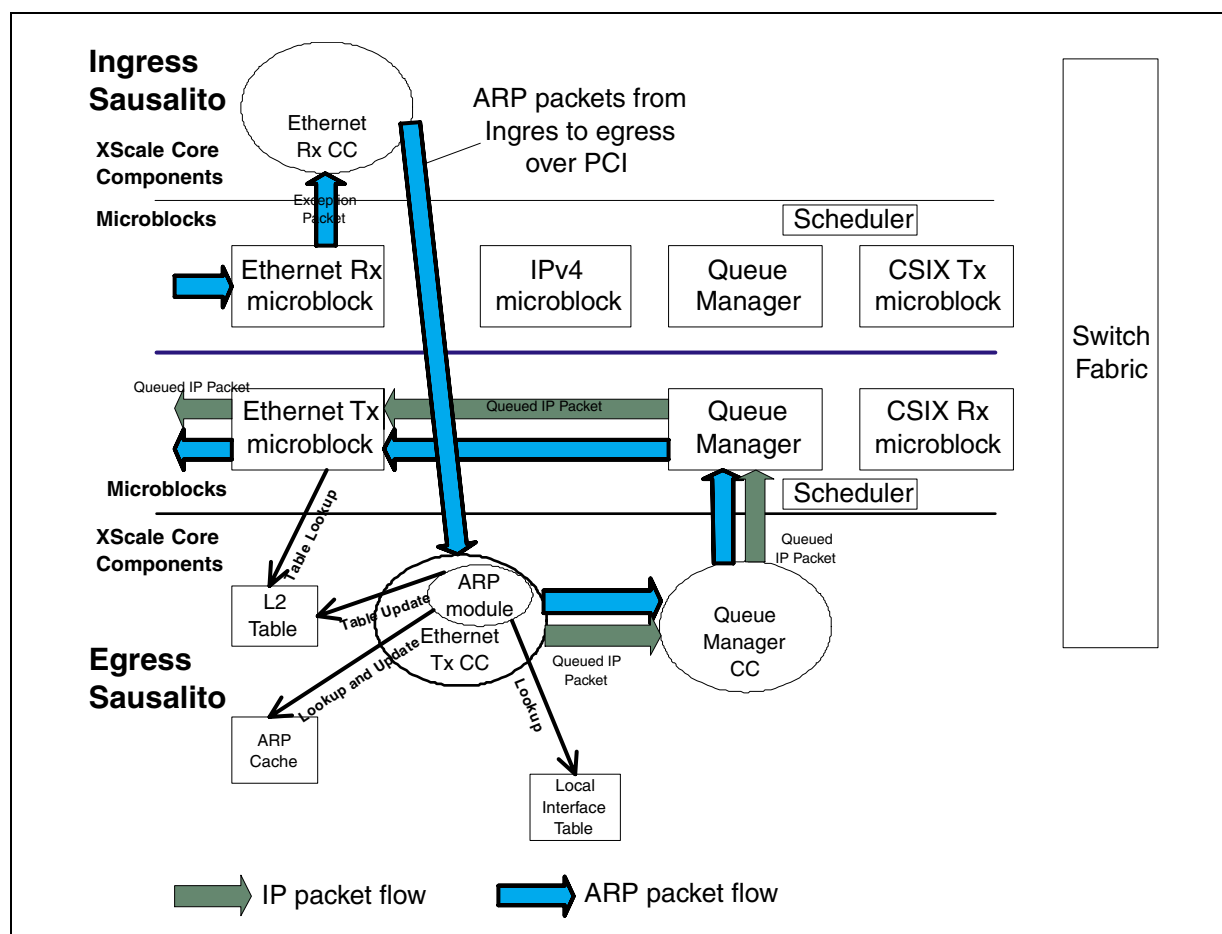
After receiving a packet, the Ethernet TX microblock performs a table lookup in the L2 table based on the next hop ID in the packet meta-data. If the layer-2 address cannot be found, the microblock sends this packet to Ethernet TX Core Component as an exception packet.

The Ethernet TX Core Component accesses its ARP module for the resolution of the L2 info. The ARP module either performs an ARP cache lookup and synchronizes the L2 table or it generates an ARP request packet for soliciting the L2 info. In the former case, once the L2 table is synchronized, the IP packet is sent back to the microblock through the Queue Manager Core Component for transmission. In the latter case, the ARP packet is handed over to the Queue Manager Core Component. The Queue Manager Core Component sends the ARP packet to the Queue Manager microblock for transmitting out to the media. The exception IP packet is held by the ARP module to be sent out later when a corresponding ARP reply is received.

47.2.2 ARP Reception Data Flow

Figure 47-3 illustrates Ethernet ARP reception data flow.

Figure 47-3. Ethernet ARP Reception Data Flow



The Ethernet TX Core Component also receives ARP packets sent through the PCI bus from the Ethernet RX Core Component.

In this case, the Ethernet TX Core Component hands the ARP packet to its ARP module for processing. Processing continues as follows:

1. The ARP module performs a lookup in the Local Interface Table to try to match the target IP address.
2. If there is a match, the ARP module handles the packet based on the ARP packet type.
3. If the ARP packet is a broadcast ARP request packet, the ARP module sends out a unicast ARP reply packet. This ARP reply packet is sent to the Queue Manager Core Component, which transmits it to the network media through the microblocks.
4. If the ARP packet is an ARP reply packet, the ARP module updates both the L2 table and its ARP cache and sends any previously held IP packets to the Queue Manager Core Component, which transmits them to the network media through the microblocks.

47.3 External API

47.3.1 Error Codes

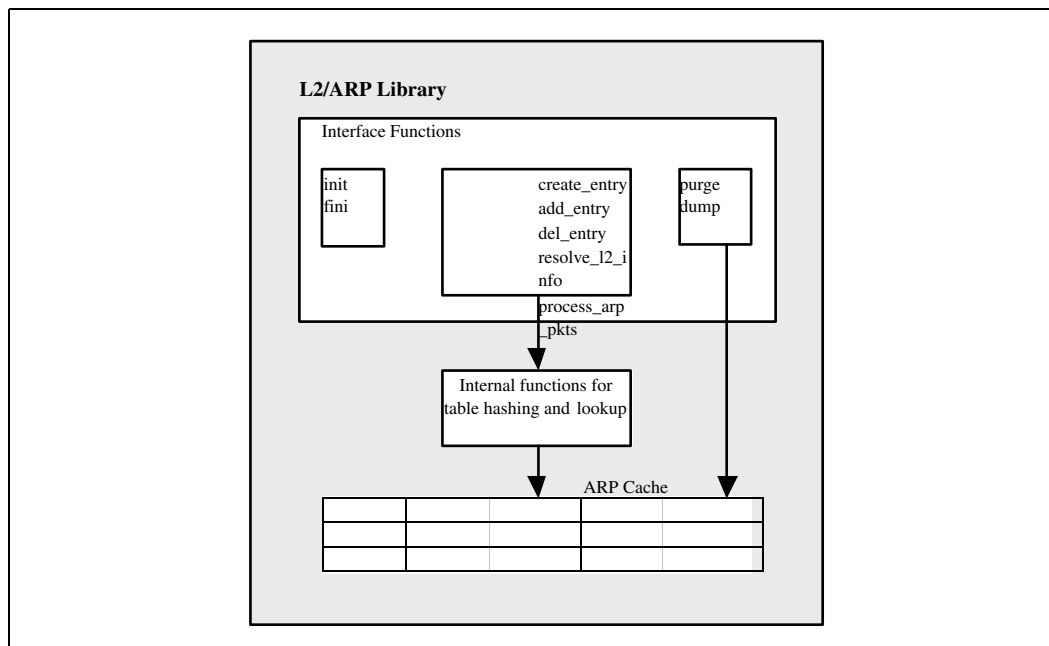
The Ethernet ARP module defines the following error codes for its external functions:

```
IX_CC_ARP_ERROR_INVALID_L2_INDEX
IX_CC_ARP_ERROR_INCORRECT_HEADER
IX_CC_ARP_ERROR_INVALID_PARAMETER
IX_CC_ARP_INVALID_HANDLE
IX_CC_ARP_ERROR_NHIP_NOT_FOUND
IX_CC_ARP_ERROR_NO_CACHE_ENTRY
IX_CC_ARP_ERROR_SIP_CONFLICT
IX_CC_ARP_ERROR_NO_LOCAL_MAC_ADDRESS
IX_CC_ARP_ERROR_ENTRY_EXIST
IX_CC_ARP_FAIL_UPDATE_L2TM
IX_CC_ARP_ERROR_EVENT_HANDLER_GET_TIME
IX_CC_ARP_ERROR_CCI
```

47.4 Modularity

Figure 47-4 illustrates the logical sub-modules in the Ethernet ARP library.

Figure 47-4. Modularity of the Ethernet ARP Module



The interface function block is accessed directly by the Ethernet TX core component. The ARP cache is allocated and de-allocated by the init and fini functions. ARP entries are added, deleted, and updated during run time by the interface functions, for example, `ix_cc_arp_create_entry`, and `ix_cc_arp_add_entry`. The internal function block supports the table look-up operation performed by these interface functions through some well-defined hashing mechanism. The purge and dump functions access the ARP cache directly for cleaning or retrieving the entire ARP cache information.

47.4.1 External API

The Ethernet ARP library provides the following external functions for use by the Ethernet TX Core Component. For complete details of this API, see [Chapter 10, “Ethernet ARP Module”](#) of the *IXA Software Building Blocks Reference Manual*.

The ARP cache is allocated and de-allocated by the `init` and `fini` functions respectively. ARP entries are added, deleted, and updated during run-time by the interface functions such as `ix_cc_arp_create_entry`, `ix_cc_arp_add_entry`, etc. Table lookups are performed by these interface functions through a well-defined hashing mechanism. The purge and dump functions access the ARP cache directly for cleaning or retrieving the entire ARP cache information.

Table 47-1 shows the Ethernet ARP library API.

Table 47-1. Ethernet ARP Library API

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

Queue Manager Components

The Queue Manager includes the following core components:

- [Chapter 48, “Queue Manager Core Component”](#)

Queue Manager Core Component – There are two Queue Manager Core Components that exist in the system – Ingress Queue Manager Core Component and Egress Queue Manager Core Component. Queue Manager Core Component provides the following functionality in IXA SDK 3.1, common to both Ingress and Egress Queue Manager:

- configuration module for Queue Manager microblock
- centralized packet en-queuing from core to the microblocks
- handling of packets en-queued for local output ports in case the switch fabric does not support loop back

- [Chapter 49, “Queue Manager \(DiffServ\) Core Component”](#)

The Egress Queue Manager Core Component for DiffServ is identical to the Queue Manager Core Component detailed in [Chapter 48, “Queue Manager Core Component.”](#)

Queue Manager Core Component 48

48.1 Overview

The Queue Manager Core Component performs the following functions:

- configuration for Queue Manager microblock
- centralized packet enqueueing from all core components to the microblocks
- handling of packets enqueued for local output ports in case the switch fabric does not support loopback.

There are generally two Queue Manager Core Components in an application—one for ingress and one for egress. In most of the cases, the Queue Manager Core Component on ingress side receives packets and enqueues them to be sent to the CSIX switch fabric media. The Queue Manager on the egress side enqueues packets to be sent to the ATM, POS, or Ethernet media. The differences between these two are mentioned explicitly in the remainder of this chapter.

There are two types of Queue Manager microblocks—Cell Based and Packet. In this chapter, the term “Queue Manager microblock” refers to both types. For more information on the Queue Manager microblocks, see [Chapter 13, “Queue Manager For OC-48 Microblock”](#) and [Chapter 15, “Packet Queue Manager Microblock.”](#) For external APIs see [Chapter 11, “Queue Manager”](#) of the *IXA Software Building Blocks Reference Manual*.

48.2 Configuration and Initialization

48.2.1 Configuration

Both the ingress and egress Queue Manager Core Components must set up and configure their respective Queue Manager microblocks. The configuration includes:

- Patching Queue Manager microblock symbols with Queue Manager specific values
- Allocating memory blocks for use by the Queue Manager microblock

The values to patch into the symbols are defined either by the Queue Manager Core Component or they are written to the system registry by the system application during initialization (see [Chapter 40, “System Application”](#)).

The number and size of memory blocks, as well as memory source (SRAM, DRAM, or scratch ring) are defined in the Queue Manager Core Component. During initialization, the Queue Manager Core Component allocates symbol/value pair data structures and uses the Resource Manager to patch the symbols.

In addition, it uses the Resource Manager to allocate memory for the Queue Manager microblocks, and patches the memory physical base addresses and data sizes to the microblock. The Queue Manager Core Component does not provide the ability to change configuration parameters dynamically.

48.2.2 Configuration Items

Table 48-1 lists the configuration items patched by the Queue Manager Core Component for use by the Queue Manager microblock.

Table 48-1. Queue Manager Core Component Configuration Items

Configuration Item	Description
QD_SRAM_BASE	Base Address in SRAM for Queue Descriptors
QD_TOTAL	Total number of Queue Descriptors - currently 1024
QM_DROP_QUEUE_ENTRY	Currently 20. This identifies which Q-Array is used as drop queue.

Note that this list of variables to be patched can change depending on the requirements of the Queue Manager microblock.

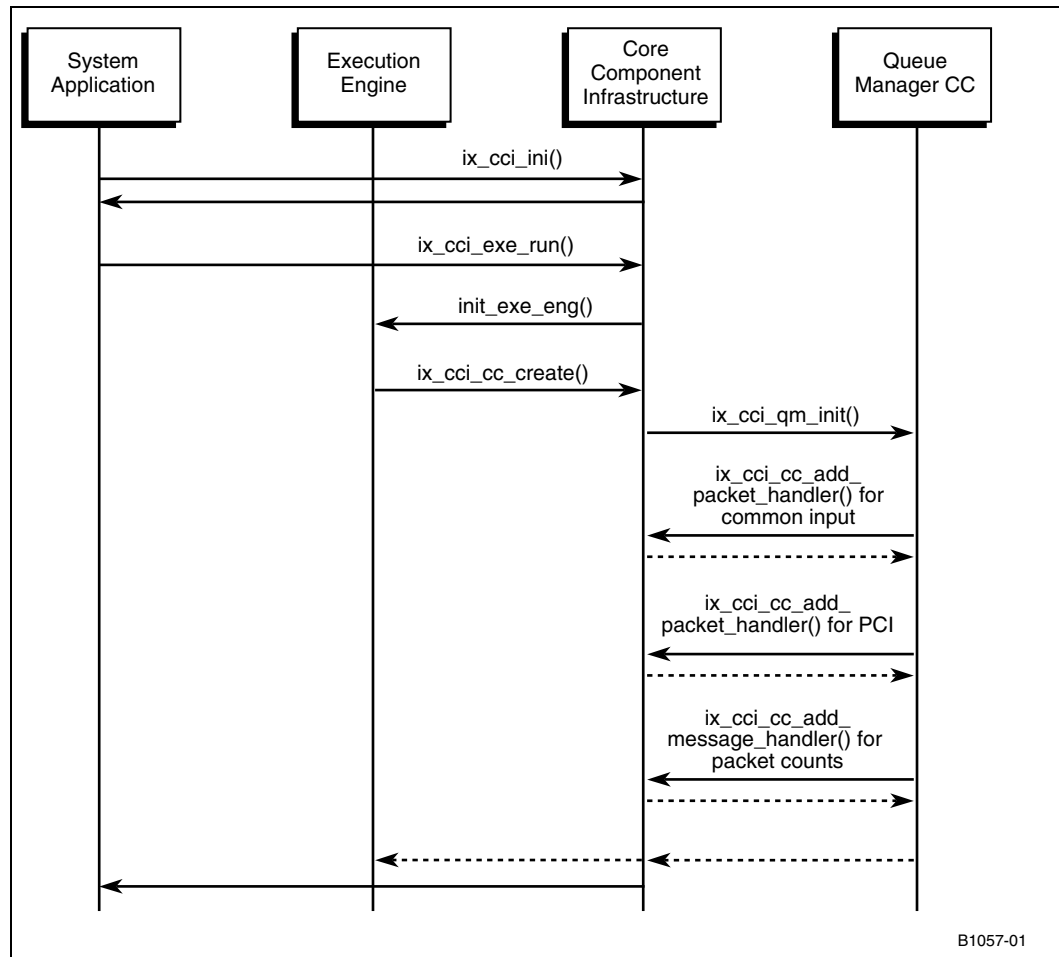
48.2.3 Initialization

Initialization involves the following steps:

1. The system application initializes the Core Component Infrastructure and creates an Execution Engine in which the Queue Manager core component runs.
2. The Execution Engine creates all core components assigned to the engine, including the Queue Manager Core Component.
3. The Core Component Infrastructure invokes the initialization function of the Queue Manager Core Component, `ix_cc_qm_init()`.
4. From inside the `init()` function, the Queue Manager Core Component registers packet and message handlers.
5. After successfully registering, the Queue Manager Core Component returns control to Core Component Infrastructure, which, in turn, returns control to the system application.

Figure 48-1 shows the sequence of calls between the System Application, the Core Component Infrastructure and the Queue Manager Core Component during initialization.

Figure 48-1. Initialization Flow of the Queue Manager Core Component



B1057-01

48.3 Data Flow

Other core components direct processed packets to the Queue Manager Core Component, which then enqueues packets to its corresponding Queue Manager microblock (either Cell Based or Packet Based, depending on the application). The Core Component Infrastructure API is used for communication between the core component and the microblock. (see the *IXA Portability Framework Reference Manual*). In general, the Queue Manager Core Component on the ingress side receives packets and enqueues them to be sent to the CSIX switch fabric media, and the Queue Manager Core Component on the egress side enqueues packets to be sent to the ATM, POS, or Ethernet media.

In the case where packets need to go from the ingress side to the egress side of the same blade and the switch fabric does not support loopback, the Queue Manager Core Component matches packets against the output blade id from the packet metadata and uses PCI to send packets to the egress Queue Manager Core Component.

48.4 External API

This section lists the external API for Queue Manager core component. For complete details, see Chapter 11, “Queue Manager” in the *IXA Software Building Blocks Reference Manual*.

48.4.1 Core Component Infrastructure API

Table 48-2 summarizes the Core Component Infrastructure API for the Queue Manager Core Component.

Table 48-2. Queue Manager Core Component Infrastructure API

Name	Description
<code>ix_cc_qm_init()</code>	Initializes a Queue Manager Core Component.
<code>ix_cc_qm_fini()</code>	Terminates a Queue Manager Core Component.
<code>ix_cc_qm_pkt_handler()</code>	Receives the packets sent to the Queue Manager by other core components.
<code>ix_cc_qm_msg_handler()</code>	Processes messages of type <code>IX_CC_QM_MSG_GETPKTCOUNT</code> .

48.4.2 Functional API

The Functional APIs for the Queue Manager (Messaging API and Library API) report the total number of packets processed by the Queue Manager Core Component. The main use of this API is to provide debugging support.

48.4.2.1 Messaging API

Table 48-3 summarizes the Queue Manager core component Messaging API.

Table 48-3. Queue Manager Core Component Messaging API

Name	Description
<code>ix_cc_qm_async_get_packet_count()</code>	Returns the number of packets processed by the Queue Manager core component.

48.4.2.2 Library API

Table 48-4 summarizes the Queue Manager core component Library API.

Table 48-4. Queue Manager Core Component Library API

Name	Description
<code>ix_cc_qm_get_packet_count()</code>	Reports the number of packets processed by a Queue Manager Core Component.

Queue Manager (DiffServ) Core Component

49

49.1 Overview

The Egress Queue Manager Core Component for DiffServ is identical to the Queue Manager Core Component detailed in [Chapter 48, “Queue Manager Core Component.”](#)

For a complete description of the Egress Queue Manager (DiffServ) Microblock, see [Chapter 21, “Egress Queue Manager \(DiffServ\) Microblock”](#) and for external APIs see [Chapter 12, “Egress Queue Manager \(DiffServ\)”](#) of the *IXA Software Building Blocks Reference Manual*.

Scheduler Components

The Scheduler includes the following core components:

- [Chapter 50, “Scheduler Core Component”](#)

Two scheduler core components exist in the system—Ingress Scheduler core component and Egress Scheduler core component. The Ingress Scheduler core component configures the CSIX Scheduler microblock, and is called as CSIX Scheduler core component; Egress Scheduler core component configures the Packet Scheduler microblock and is called the Packet Scheduler core component.

- [Chapter 51, “Scheduler \(DiffServ\) Core Component”](#)

The Scheduler Core Component for DiffServ is virtually identical to the Scheduler Core Component described in [Chapter 50, “Scheduler Core Component.”](#)

The only difference is that the Scheduler (DiffServ) Core Component has the additional new patching symbol

50.1 Overview

The Scheduler Core Component initializes and configures its corresponding microblock. In a typical application, there are two Scheduler core components—the CSIX Scheduler runs on the ingress side, while the Packet Scheduler runs on the egress side.

The Scheduler on the ingress side configures the scheduling algorithm for the CSIX Scheduler microblock and is called the CSIX Scheduler Core Component. The Scheduler on the egress side configures the scheduling algorithm for the Packet Scheduler microblock and is called Packet Scheduler Core Component.

Both the CSIX and Packet Scheduler Core Components have the following common functionality:

- Starts and configures itself as a standard core component
- Read configuration data on initialization from system registry or from the common header file in the absence of the registry

The CSIX Scheduler also has the following specific functionality:

- The CSIX Scheduler Core Component creates an array of entries in shared memory representing <port, class> maps for WRR algorithm in the microblock. The number of entries is application dependent, and is currently 1024. The number of entries can be changed based on the system design. See [Chapter 17, “Fabric Scheduler For OC-48”](#) for a description of array entries.
- Patches the CSIX Scheduler microblock with base address of the <port, class> map array

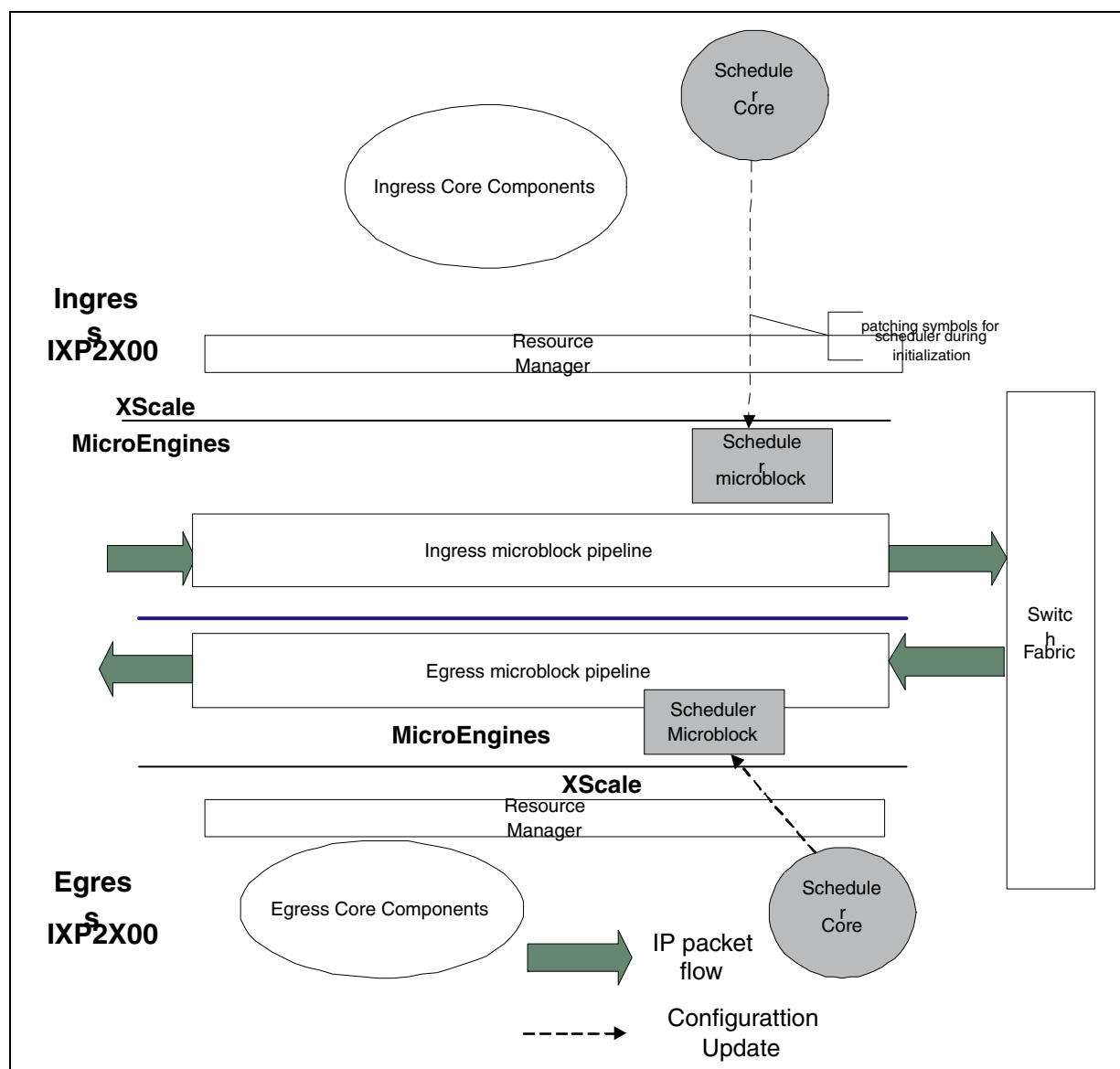
The Packet Scheduler also has the following specific functionality:

- The Packet Scheduler Core Component creates a region of shared memory combined from one array of 16 entries (representing the class weight for each output port and an array of 256 entries) representing credit for each class queue (16 class queues per each of the 16 ports) to support the DRR algorithm in the microblock. See [Chapter 19, “OC-48 WRR/DRR Packet Scheduler”](#) for a description of the array entries.
- Patches the Egress Packet Scheduler microblock with the base address of the combined weight array and the base address of the <port, queue> map array

The Scheduler Core Component does not receive packets from other microblocks or core components.

For more information on the Scheduler microblocks, refer to [Chapter 17, “Fabric Scheduler For OC-48”](#) and [Chapter 19, “OC-48 WRR/DRR Packet Scheduler”](#). For external APIs see [Chapter 13, “Scheduler”](#) of the *IXA Software Building Blocks Reference Manual*.

Figure 50-1. Scheduler Core Component Data Flow



50.2 Configuration and Initialization

50.2.1 Configuration

The configuration of the Scheduler Core Component depends upon whether the core component is running on the ingress or egress sides of the application.

50.2.2 CSIX Scheduler

The CSIX Scheduler runs on the ingress side of the IXDP2400. During initialization, the CSIX Scheduler Core Component creates an array of 1024 integer values for the Weighed Round Robin algorithm used by the Scheduler microblock. These values represent 16 QoS classes for each of the 64 switch fabric ports. This array is allocated from shared SRAM memory. The base address of the array and its size are patched into the microblock. The number of entries are application dependent and can be changed if the system design changes.

Entries in this array conform to the rule that the sum of 16 consecutive values should be equal between each other. For example, the sum of values between indices 0 and 15 equal the sum of values between indices 16-31, 32-47, etc. This requirement is necessary for supporting the algorithm in the microblock. The values for the array are provided by the System Application during initialization. These values are static and do not change during run-time. The System Application retrieves the values from the registry or from the common header file if the registry is not being used.

50.2.3 Packet Scheduler

The Packet Scheduler runs on the egress side of the IXDP2400. During initialization, the Packet Scheduler core component allocates two arrays for the Egress Scheduler microblock. One is an array of 16 weights where each entry corresponds to an individual output port. The other array is 256 entries in size, containing 16 classification queues for each of the 16 ports. Both of these arrays are allocated from shared SRAM memory. The base addresses of the arrays and their sizes are patched into the microblock.

50.2.4 Configuration Items

Table 25-1 by Scheduler core component in the control block to be patched into the Scheduler microblock.

Table 50-1. Scheduler Core Component Configuration Items

Configuration Item	Description
SCHED_VOQ_BASE (Ingress Scheduler)	Address in SRAM for base of array of virtual output queues - <port,class> map for the Ingress Scheduler microblock. Corresponds to the QD_SRAM_BASE for the Queue Manager microblock
SCHED_VOQ_SIZE (Ingress Scheduler)	Total number of <port,class> entries- currently 1024 (64 ports x 16 QoS classes). Currently this variable is hardcoded in the CSIX Scheduler microblock and is not being patched. This value is used to create entries in the <port,class> array.

Table 50-1. Scheduler Core Component Configuration Items

Configuration Item	Description
SCHED_WEIGHT_BASE (Egress Scheduler)	Address in SRAM for the base of an array of weights for the ports for Weighted Round Robin scheduling and the base of an array for queues per port <port, queue> for DRR scheduling.
SCHED_WEIGHT_SIZE (Egress Scheduler)	Size of weight array - currently 16 (16 output ports). Currently this variable is hard coded in the Packet Scheduler microblock and is not being patched. This value is used to create entries in the weight array.
SCHED_PORT_QUEUE_SIZE (Egress Scheduler)	Size of port/queue array - currently 256 (16 queue for each of the 16 ports). Currently this variable is hard coded in the Packet Scheduler microblock and is not being patched. This value is used to create entries in the port/queue array.

50.2.5 Initialization

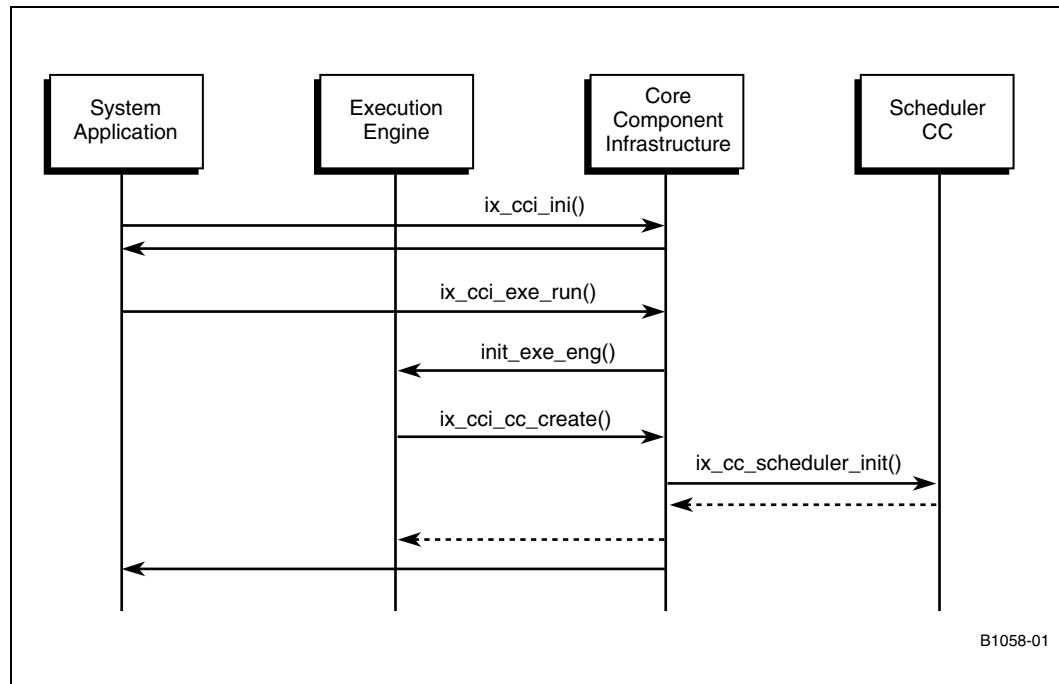
Initialization involves the following steps:

1. The system application initializes the Core Component Infrastructure and creates an Execution Engine in which the Queue Manager core component runs.
2. The Execution Engine creates all core components assigned to the engine, including the Scheduler core component.
3. The Core Component Infrastructure invokes the initialization function of the Scheduler core component, `ix_cc_qm_init()`.
4. The Scheduler core component returns control to Core Component Infrastructure, which, in turn, returns control to the system application.

For more information on the Core Component Infrastructure, refer to the *Intel® Internet Exchange Architecture Portability Framework Developer's Manual*.

Figure 50-2 shows the sequence of calls between the System Application, the Core Component Infrastructure and the Queue Manager core component during initialization.

Figure 50-2. Scheduler Core Component Initialization Data Flow



50.3 Core Component Infrastructure API

Table 25-2 summarizes the Scheduler Core Component Infrastructure API. For complete details, see Chapter 13, “Scheduler” in the *IXA Software Building Blocks Reference Manual*.

Table 50-2. Scheduler Core Component Infrastructure API

Name	Description
<code>ix_cc_scheduler_init()</code>	Initializes the scheduler core component.
<code>ix_cc_scheduler_fini()</code>	Terminates the scheduler component.

Scheduler (DiffServ) Core Component

51

51.1 Overview

The Scheduler Core Component for DiffServ is virtually identical to the Scheduler Core Component described in [Chapter 50, “Scheduler Core Component.”](#)

The only difference is that the Scheduler (DiffServ) Core Component has the additional new patching symbol listed in [Table 51-1](#).

Table 51-1. New Patching Symbols in Scheduler (DiffServ) Core Component

Variable	Default	Description
SCHED_HIGH_PRIORITY_MASK	0x00FF	Bit mask denoting high priority queues on a port.

For complete information on the Egress Scheduler Microblock for DiffServ, refer to [Chapter 22, “Egress Scheduler \(DiffServ\) Microblock”](#) for external APIs see [Chapter 14, “Egress Scheduler \(DiffServ\)”](#) of the *IXA Software Building Blocks Reference Manual*.

Forwarder Components

The Forwarder includes the following core components:

- [Chapter 52, “IPv4 Forwarder Core Component”](#)

IPv4 Forwarder - IPv4 Forwarder is a core component that assists IPv4 microblock to forward packet buffers. IPv4 Forwarder core component together with the microblock are implementation of near-RFC1812 compliant unicast capability on IXP2X00.

- [Chapter 53, “IPv6 Forwarder Core Component”](#)

IPv6 Forwarder - IPv6 Forwarder is a core component that assists IPv6 microblock to forward packets. IPv6 Forwarder core component together with the microblock is an implementation of RFC2460-compliant unicast capability on IXP2X00.

- [Chapter 54, “IPv6 To IPv4 Tunneling Core Component”](#)

IPv6 to IPv4 Tunneling - The IPv6-IPv4 Tunneling Core Component assists the IPv6-IPv4 Tunneling microblocks to provide IPv6 over IPv4 tunneling as specified in RFC 2893 and RFC 3056.

- [Chapter 55, “NAT-PT Translation Core Components”](#)

The translation core component assists the translation microblock to implement the IPv6 to IPv4 (and vice-versa) address and protocol translation as defined in RFC2766 Network Address Translation-Port Translation specification (NAT-PT).

A single core component supports the translation microblock. The translation core component can receive packets from the translation microblock only. Messages can be received from other core components and from other system components responsible for configuring the forwarding plane.

52.1 Overview

The IPv4 Forwarder Core Component performs the following functions:

- Configures the IPv4 microblock (static configuration)
- Provides message and packet handlers to receive messages and packets from other core components and from the IPv4 microblock.
- Generates ICMP error messages
- Performs RFC1812 and RFC2644 checks
- Validates the IP header
- Handles fragmentation
- Handles packets with IP options

The IPv4 Forwarder Core Component receives packets and messages from several components of the system, including the IPv4 microblock, the POS RX core component, and the Stack Driver.

For detailed information on the IPv4 microblock, see [Chapter 24, “IPv4 Forwarder Microblock.”](#) and for external APIs see [Chapter 15, “IPv4 Forwarder”](#) of the *IXA Software Building Blocks Reference Manual*.

52.2 Assumptions and Dependencies

52.2.1 Assumptions

These are some of the assumptions made about the implementation of the IPv4 Forwarder Core Component:

- SNMP is not included in the IPv4 Forwarder. The IPv4 Forwarder keeps counters but does not store these counters in any MIBs. In addition, configuration parameters may not be passed in via MIB. The IPv4 Forwarder exposes an interface for querying statistics and for configuration. Another component then does the translation to MIB format. In addition, the statistics kept are not the complete set specified by the MIBs.
- The IPv4 Forwarder does not support multicast or broadcast forwarding. Broadcast packets destined for local delivery are delivered to the Stack Driver but are not forwarded beyond the router. Multicast packets are also sent to the TCP/IP stack
- The TOS field in the IP header is ignored. No TOS handling is done in either the IPv4 microblock or the IPv4 Forwarder Core Component. The TOS field is treated as an unknown field and is not changed by the IPv4 Forwarder.
- The IPv4 Forwarder Core Component does not keep statistics for each port.

- The ICMP implementation only generates ICMP error messages; it only sends error messages. It is assumed that ICMP echo request and reply messages are sent to the TCP/IP stack for handling through the Stack Driver core component (see [Chapter 64, “Stack Driver”](#)).
- All input and output parameters must be represented in network byte order.

52.2.2 Dependencies

The IPv4 Forwarder Core Component uses the services of the IXA Portability Framework for:

- allocating memory
- freeing memory
- patching symbols
- 64-bit counter support
- accessing the system registry to retrieve static parameters.

The IPv4 Forwarder Core Component uses the Core Components Infrastructure services for:

- event handling
- message handling
- packet handling between core components and the IPv4 microblock

For more information on the IXA Portability Framework and the Core Component Infrastructure, refer to the *Intel[®] Internet Exchange Architecture Portability Framework Developer's Manual*.

Services of the Route Table Manager are used to add, delete and look up route information (see [Chapter 60, “Route Table Manager”](#)). The IPv4 Forwarder Core Component receives exception packets from the IPv4 microblock.

52.3 Configuration and Initialization

If the system supports registry services, the IPv4 Forwarder Core Component uses the registry for static configuration. If there is no registry support, the IPv4 Forwarder uses compile-time defined values for static configurations in one of its header files.

The registry is used for the following static configuration parameters:

- Route table size Hint—indicates approximately how many routes are supported
- Route table type—software or hardware (TCAM)
- Size of next hop table size
- RTM lookup Memory Type
- Next hop database memory type

The IPv4 Forwarder Core Component supports dynamic property updates as specified in [Section 39.2.1, “Dynamic Properties and Clients”](#) on page 666.

Note that properties of interfaces are updated dynamically by the Control Plane PDK. The registry is used to store initial property values for the start-up configuration and for system configurations without the Control Plane PDK. The messaging mechanism is used to update property values during run-time.

During initialization, the IPv4 Forwarder Core Component allocates memory and patches symbols. The IPv4 Forwarder Core Component is responsible for patching symbols used by the Route Table Manager and uses Route Table Manager services to get these values.

The IPv4 Forwarder Core Component must create 64-bit counters to swap 32-bit counter values from the IPv4 microblock. The IPv4 Forwarder Core Component uses the services of the IXA Portability Framework for allocating, patching the base of the 32-bit counter and managing the 64-bit counters (see [Chapter 24, “IPv4 Forwarder Microblock”](#)).

The client can get statistics from the IPv4 Forwarder Core Component, the Route Table Manager or the IPv4 microblock. The IPv4 microblock statistics are in a 64-bit counter and the IPv4 Forwarder Core Component statistics are stored in a 32-bit counter. The IPv4 microblock and the Route Table Manager statistics are obtained only by a client call. This allows the IPv4 Forwarder Core Component to avoid a periodic query to the IPv4 microblock and to the Route Table Manager.

[Table 52-1](#) lists IPv4 Forwarder Core Component statistics and their description

Table 52-1. IPv4 Forwarder Core Components Statistics

Statistics Variable	Description
IPv4MbRcvdPkts	Number of packets received by the IPv4 microblock
IPv4MbFwdPkts	Number of packets forwarded by the IPv4 microblock
IPv4MbDropPkts	Number of packets dropped by the IPv4 microblock
IPv4MbExcpPkts	Number of packets send to the IPv4 Forwarder Core Component as exception packets by the IPv4 microblock
IPv4MbBadHeaderPkts	Number of packets received with bad headers by the IPv4 microblock
IPv4MbBadTotalLengthPkts	Number of packets received with bad total length by the IPv4 microblock
IPv4MbBadTTLPkts	Number of packets received with bad TTL from the IPv4 microblock
IPv4MbBadNoRoutePkts	Number of packets dropped by the IPv4 microblock due to an invalid route.
IPv4MbLengthTooSmallPkts	Number of packets received with length too small by the IPv4 microblock
IPv4CoreInvalidHeaderPkts	Number of packets dropped by the IPv4 Forwarder Core Component due to invalid header.
IPv4CoreInvalidAddressPkts	Number of packets dropped by the IPv4 Forwarder Core Component due to invalid source/destination IP address.
IPv4CoreRcvdPkts	Number of packets received by the IPv4 Forwarder Core Component.

Table 52-1. IPv4 Forwarder Core Components Statistics

Statistics Variable	Description
IPv4CoreFwdPkts	Number of packets attempted to forward by the IPv4 core component.
IPv4CoreLocalDeliveryPkts	Number of packets delivered to the Stack Driver by the IPv4 Forwarder Core Component.
IPv4CoreNoRoutePkts	Number of packets dropped by the IPv4 Forwarder Core Component due to invalid route.
IPv4CoreInvalidFragPkts	Number of packets needed to be fragmented by the IPv4 Forwarder Core Component, but were dropped due to valid DF bit.
IPv4CoreCreatedFragmentedPkts	Number of fragment packets created by the IPv4 Forwarder Core Component.
IPv4CoreCreatedICMPMsgPkts	Number of created ICMP error messages packets by the IPv4 Forwarder Core Component.
IPv4CoreICMPSendFailed	Number of ICMP error message packets failed to send out by the IPv4 Forwarder Core Component due to problems like no buffer.
IPv4CoreCreatedICMPDestUnReachErrorPkts	Number of ICMP Destination Unreachable error message packets created by the IPv4 Forwarder Core Component.
IPv4CoreCreatedTimeExceedPErrorkts	Number of ICMP Time Exceeded error message packets created by the IPv4 Forwarder Core Component.
IPv4CoreCreatedParamProblemErrorPkts	Number of ICMP parameter problem error message packets created by the IPv4 Forwarder Core Component.
IPv4CoreCreatedRedirectErrorPkts	Number of ICMP Redirect error message packets created by the IPv4 Forwarder Core Component.
IPv4NumberOfRoutes	Number of routes in the Route Table Manager
IPv4NumberOfNextHops	Number of next hops in Route Table Manager

52.4 IPv4 Forwarder Core Component Modules

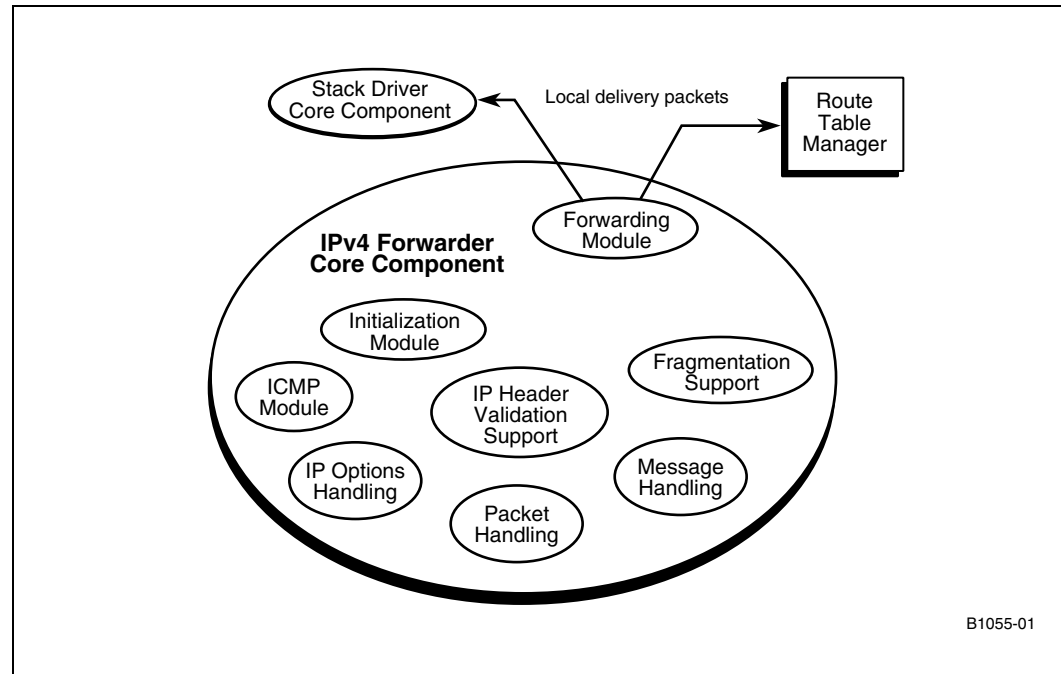
The IPv4 Forwarder Core Component provides compile time switches for RFC 1812 SHOULD features (RFC 1812 MUST features are supported by default) and RFC 2644 features.

IPv4 Forwarder is comprised of several modules:

- ICMP Module
- Forwarding and IP Header Validation Module
- IP Option handling Module
- Fragmentation Module
- Packet Handling Module
- Message Handling Module

Figure 52-1 shows the IPv4 Forwarder Core Component modules and their interaction with the Route Table Manager and the Stack Driver.

Figure 52-1. IPv4 Forwarder Core Component Modules



52.4.1 ICMP

The ICMP module builds and sends ICMP error messages for the IPv4 Forwarder Core Component as detailed in the Internet Control Message Protocol RFC 792. After determining that an ICMP error needs to be generated, the IPv4 Forwarder Core Component calls one of the appropriate functions of the ICMP module. These functions, in turn, verify that the packet can cause an ICMP error message to be sent (according to RFC1812), builds the ICMP error message and queues the ICMP error message to be sent.

52.4.1.1 Supported ICMP Error Messages

The ICMP error messages supported are:

- Parameter Problem

If the gateway or host processing a datagram finds a problem with the header parameters such that it cannot complete processing of the datagram, it must discard the datagram. One potential source of such a problem is incorrect arguments in an option. The gateway or host may also notify the source host via the parameter problem message. This message is only sent if the error caused the datagram to be discarded.

- Destination Unreachable

If, according to the information in the gateway's routing tables, the host specified in the internet destination field of a datagram is unreachable, the gateway may send a destination unreachable message to the internet source host of the datagram. In addition, in some networks, the gateway may be able to determine if the internet destination host is unreachable.

Gateways in these networks may send destination unreachable messages to the source host when the destination host is unreachable.

The following Destination Unreachable error messages are supported:

- Host unreachable
- Fragmentation needed and DF bit set
- Source route failed

- Redirect

The gateway sends a redirect message to a host in the following situation.

- A gateway, G1, receives an internet datagram from a host on a network to which the gateway is attached.
- The gateway, G1, checks its routing table and obtains the address of the next gateway, G2, on the route to the datagram's internet destination network, X.
- If G2 and the host identified by the internet source address of the datagram are on the same network, a redirect message is sent to the host. The redirect message advises the host to send its traffic for network X directly to gateway G2 as this is a shorter path to the destination. The gateway forwards the original datagram's data to its internet destination.

Being a router, only one Redirect error message is supported, which is the Redirect datagram for the host. Redirect datagrams for the TOS are not supported; the TOS field is ignored.

- Time Exceeded

If the gateway processing a datagram finds the Time To Live field is zero, it must discard the datagram. The gateway may also notify the source host via the Time Exceeded message. Only the Time to Live Exceeded in Transit error message is supported. Fragmentation reassembly time exceeded is not supported due to the fact that all fragments that are meant for local delivery are sent to the TCP/IP stack and the TCP/IP stack takes care of reassembly of fragments.

The following RFC1812 MUST features for ICMP error message generation are supported:

1. If a router cannot forward a packet because it has no routes at all (including no default route) to the destination specified in the packet, then the router MUST generate a Destination Unreachable, Code 0 (Network Unreachable) ICMP message.
2. A router SHOULD NOT originate ICMP Source Quench messages. A router that does originate Source Quench messages MUST be able to limit the rate at which they are generated. The IPv4 Forwarder Core Component does not generate ICMP Source Quench message.
3. A router MUST generate a Parameter Problem message for any error not specifically covered by another ICMP message.
4. A router MUST generate a Time Exceeded message Code 0 (In Transit) when it discards a packet due to an expired TTL field.

52.4.1.2 Rate Limiting

According to RFC1812, a router which sends ICMP Source Quench messages MUST be able to limit the rate at which the messages can be generated. A router SHOULD also be able to limit the rate at which it sends other sorts of ICMP error messages (Destination Unreachable, Redirect, Time Exceeded, Parameter Problem). The rate limit parameters SHOULD be settable as part of the configuration of the router. How the limits are applied (e.g., per router or per interface) can be decided depending on the implementation.

52.4.2 Forwarding and IP Header Validation Module

When attempting to forward packets, the IPv4 Forwarder Core Component must perform IP header validation (RFC1812 and RFC2644 checks). It must also check packets for local delivery.

52.4.2.1 Forwarding

The IPv4 Forwarder Core Component receives packets from the following sources:

- the Stack Driver
- the IPv4 microblock
- other core components

When forwarding packets coming from the Stack Driver, the IPv4 Forwarder Core Component does not do IP header validation for packets; it only checks for the directed broadcast source address and drops the packets which have the directed broadcast source address. The IPv4 Forwarder then performs a route lookup on the destination IP address. There is no Time To Live field decrement required for packets coming from the Stack Driver as the Stack Driver takes care of decrementing the Time To Live field. Note that when the IPv4 Forwarder Core Component sees problems with the packet (e.g. route lookup failed) and drops the packet, it does not send ICMP error messages back to the Stack Driver core component but instead simply drops the packet.

When the IPv4 Forwarder Core Component receives packets from the IPv4 microblock or from other core components for which it has to do forwarding, it does IP header validation for the packets and also takes care of Time to Live field decrement. For certain types of exception packets coming from the IPv4 microblock it is assumed that the IPv4 microblock fill in meta data in the buffer for which it performs route lookup before sending the packets to the IPv4 Forwarder Core Component. The IPv4 Forwarder Core Component performs a lookup in the shared route table for route information for those packets for which the IPv4 microblock does not fill in the meta data. These fields are inserted into the packet meta data and the packet is forwarded for output from the IPv4 core component. Packets meant for local delivery are sent to Stack Driver.

52.4.2.2 IP Header Validation

There are two kinds of RFC1812 checks that must be performed for IP header validation. In addition, a RFC2644 check must be performed.

52.4.2.2.1 RFC1812 Packet Header validation

Before a router can process an IP packet, it must perform the following basic validity checks on the packet's IP header to ensure that the header is meaningful (RFC 1812 Section 5.2.2). If the packet fails any of the following tests, are discarded and an ICMP error message is generated.

- The packet length reported by the link layer must be large enough to hold the minimum length legal IP datagram (20 bytes)
- The IP checksum must be correct.
- The IP version number must be 4
- The IP header length field must be large enough to hold the minimum length legal IP datagram (20 bytes=5 words)
- The IP total length field must be large enough to hold the IP datagram header, whose length is specified in the IP header length field.

The checksum field is the 16 bit one's complement of the one's complement sum of all 16 bit words in the header. For purposes of computing the checksum, the value of the checksum field is zero. The checksum is for the entire packet not for IP header, so the checksum for the packet is calculated again for validation.

52.4.2.2.2 RFC1812 checks for IP addresses

Figure 52-2 lists the RFC 1812 checks for IP addresses.

Table 52-2. RFC 1812 Checks for IP Addresses

Packet Description	Action
Incoming datagrams with a source address of {0, 0} which are received for local delivery	The packet must be accepted if the router implements the associated protocol and that protocol clearly defines the appropriate action to be taken. Otherwise, a router must silently discard any locally-delivered datagram whose source address is {0, 0}.
{-1, -1} Limited broadcast	This address must not be used as a source address. A datagram with this destination address is received by every host and router on the connected physical network, but is not forwarded outside that network.
{<Network-prefix>, -1} Directed Broadcast	This is a broadcast directed to the specified network prefix. It must not be used as a source address.
{127, <any>} Internal host loopback address	Addresses of this form must not appear outside a host.
Packet destination is an IP multicast address (28 bit prefix), and at least one of the logical interfaces is associated with the interface on which the packet arrived	The packet is delivered locally. This implementation only checks for multicast packets i.e. 224.0.0.* and send them to the TCP/IP stack. This was done as there is currently no infrastructure support for multicasting.

Additionally, RFC1812 sec 4.2.3.1 states that a router should silently discard on receipt any packet addresses to 0.0.0.0 or <net-prefix, 0>

52.4.2.2.3 RFC2644 check

RFC 2644 states that a router MAY have an option to enable receiving network-prefix-directed broadcasts on an interface and MAY have an option to enable forwarding network-prefix-directed broadcasts. These options MUST default to blocking receipt and blocking forwarding of network-prefix-directed broadcasts.

To check for a directed broadcast Source IP address, a directed broadcast table is maintained in shared memory between the microblock and the IPv4 Forwarder core component. To ensure that a directed broadcast address is not used as the source address, the hash table maintains the directed broadcast addresses for all interfaces. The IPv4 Forwarder core component creates the hash table.

The population of directed broadcast source address is the responsibility of the Control Plane PDK or any other configuration application that the developer might be using. The source IP address of incoming packet is compared against these addresses. If there is a match, the packet is dropped. In this way, the source address is checked for directed broadcast.

The IPv4 Forwarder Core Component adds directed broadcast addresses for all active interfaces in the Route table with IX_CC_IPV4_NH_DROP as the nexthop-identifier value. To check incoming packets for directed broadcast destination IP address, a route lookup is performed based on the destination IP address. If there is a match, the packet is dropped. In this way, destination IP address is checked for directed broadcast.

52.4.2.2.4 Microcode interaction(Exception handling)

If the IPv4 microblock is not able to find the route for a particular packet, it sends it to the IPv4 Forwarder Core Component with the exception tag `IPv4_EXCP_NO_ROUTE`. The IPv4 core component then sends an ICMP Destination Unreachable message for the packet.

52.4.2.3 Identifying Packets for Local Delivery

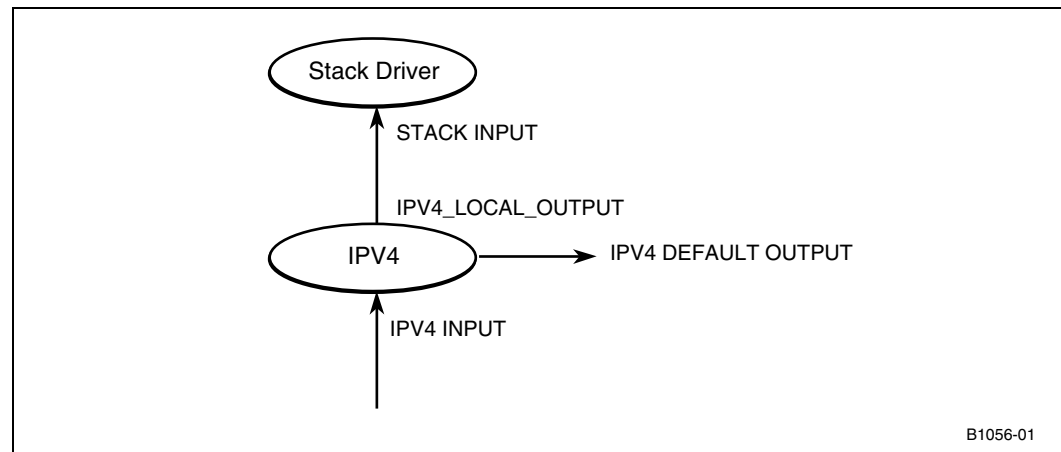
When a router receives an IP packet, it must determine whether the packet is addressed to the router for local delivery or whether the packet is addressed to another system and must be forwarded (RFC1812 section 5.2.3).

A packet is delivered locally and not forwarded in the following cases:

- the packet's destination address exactly matches one of the router's IP address
- the packet's destination address is a limited broadcast address $\{-1, -1\}$
- If packet destination is an IP multicast address (28 bit prefix), and at least one of the logical interfaces is associated with the interface on which the packet arrived, then the packet is delivered locally.

Packets meant for local delivery are sent by the IPv4 Forwarder Core Component to the TCP/IP stack through the Stack Driver. The bindings between the IPv4 Forwarder Core Component and the Stack Driver core component are shown in [Figure 52-2](#). These bindings are defined in the header file, `bindings.h`

Figure 52-2. IPv4 Forwarder Core Component Bindings



While forwarding packets, the IPv4 Forwarder Core Component does a route lookup. If the resulting route entry is marked for local delivery, it increments the count of local delivery packets and sends the packet to the Stack Driver.

52.4.3 IP Options Handling

If a packet has the IP Option field set, then the IPv4 Forwarder Core Component calls the IP Options processing routine. The supported IP Options are:

- IP Record Route Option

The Record Route option provides a means to record the route of an internet datagram. A recorded route is composed of a series of internet addresses. Each internet address is 32 bits or 4 octets. When an internet module routes a datagram it checks to see if the record route option is present. If it is, it inserts its own internet address (as known in the environment into which this datagram is being forwarded) into the recorded route beginning at the octet indicated by the pointer, and increments the pointer by four.

- Strict and Loose IP Source Route and Record Route Option

The Strict/Loose Source and Record Route (LSRR/SSRR) option provides a means for the source of an internet datagram to supply routing information to be used by the gateways in forwarding the datagram to the destination, and to record the route information.

- IP Timestamp option

The Timestamp option works like the record route option in that the timestamp option contains an initially empty list, and each router along the path from source to destination fills in one item in the list. Each entry in the list contains two 32-bit items: the IP address of the router that supplied the entry and a 32-bit integer timestamp.

RFC1812 states several rules regarding IP options handling:

- Source Route Option

A router **MUST** be able to act as the final destination of a source route. If a router receives a packet containing a completed source route, the packet has reached its final destination. In such an option, the pointer points beyond the last field and the destination address in the IP header addresses the router. The option as received (the recorded route) **MUST** be passed up to the transport layer (or to ICMP message processing).

Packets meant for local delivery are sent to Stack Driver.

- Timestamp option

If the router itself receives a datagram containing a Timestamp Option, the router **MUST** insert the current time into the Timestamp Option (if there is space in the option to do so) before passing the option to the transport layer or to ICMP for processing. If space is not present, the router **MUST** increment the Overflow Count in the option.

The IPv4 Forwarder core component uses the OSSL services, `ix_ossl_time_of_day_get()`, to get the current time from the system (see the *Intel IXA Building Blocks Reference Manual*).

Routers must insert their address into Record Route, Strict Source and Record Route, Loose Source and Record Route, or Timestamp Options. When a router inserts its address into such an option, it **MUST** use the IP address of the logical interface on which the packet is being sent.

52.4.4 Fragmentation support

The IP specification [RFC791] states that routers must accept datagrams up to the maximum of the maximum transmission units (MTUs) of the networks to which they attach. In addition, a router must always handle datagrams of up to 576 octets. The IPv4 Forwarder Core Component fragments the datagram into smaller pieces whenever the datagram size is greater than the Maximum Transmission Unit.

RFC1812 states that fragmentation **MUST** be supported by the router. When a router fragments an IP datagram, it **SHOULD** minimize the number of fragments. When a router fragments an IP datagram, it **SHOULD** send the fragments in order. A router **MUST** support reassembly of datagrams that it delivers to itself.

The IPv4 Forwarder Core Component forwards the fragments to its output endpoint (Queue Manager) for transmission. There is no guarantee that the fragments are transmitted in order.

Whenever it receives packets from the IPv4 microblock with the exception `IPv4_EXCP_FRAG_REQUIRED`, then the IPv4 Forwarder Core Component fragments the packet and forwards all of the fragments.

Fragmentation is supported to satisfy RFC1812 MUST and SHOULD features. However, reassembly is not supported for fragments of a datagram that are destined to the router. All packets meant for local delivery are sent to TCP/IP stack through the Stack Driver for reassembly.

52.4.5 Packet Handling Module

The packet handling module provides three different packet handlers for receiving packets and calling the appropriate low level functions in order to process packets received from the IPv4 microblock, the Stack Driver or other core components. The packet handling APIs are described in detail in the *Intel IXA Building Blocks Reference Manual*.

52.4.6 Message Handling Module

The message handling module is responsible for receiving messages from core components and calling the appropriate library API function to process the message. The message handler and corresponding library APIs are described in [Chapter 15, “IPv4 Forwarder”](#) of the *Intel IXA Building Blocks Reference Manual*.

52.5 External API

This section lists the external API for IPV4 Forwarder core component. For complete details, see [Chapter 15, “IPv4 Forwarder”](#) in the *IXA Software Building Blocks Reference Manual*.

52.5.1 Data Structures, Types and Macros

Table 53-3 lists the data structures, types, and macros for the IPv4 Forwarder Core Component.

Table 53-3. IPv4 Forwarder Core Component Data Structures, Types, and Macros

Data Structures, Types, and Macros	Description
<code>IX_CC_RTMV4_DUMP_ROUTE_SIZE</code>	Calculates the size of memory required to dump Route Table Manager routes
<code>IX_CC_RTMV4_DUMP_NEXT_HOP_SIZE</code>	Calculates the size of memory to dump Route Table Manager next hops
<code>ix_cc_rtmv4_nhidx</code>	Type definition for next hop identifier
<code>Reserved Next Hop Ids</code>	Adds special next hops during initialization
<code>IX_CC_RTMV4_NHID_NO_ROUTE</code>	Structure definition for next hop information
<code>ix_cc_rtmv4_next_hop_info</code>	Defines the reserved next hop ID
<code>ix_cc_ipv4_dump_data</code>	Defines the structure describing memory dump information
<code>ix_cc_ipv4_stats_data</code>	Defines the structure describing counter information

52.5.2 Core Component Infrastructure API

Table 53-4 lists the functions of the IPv4 Forwarder Core Component Infrastructure API.

Table 53-4. IPv4 Forwarder Core Component Infrastructure API

API	Description
<code>ix_cc_ipv4_init()</code>	Initializes the IPv4 Forwarder core component.
<code>ix_cc_ipv4_fini()</code>	Terminates services from the IPv4 Forwarder core component.
<code>ix_cc_ipv4_msg_handler()</code>	Passes messages to the IPv4 Forwarder Core Component.
<code>ix_cc_ipv4_microblock_high_priority_pkt_handler()</code>	Receives exception packets from IPv4 microblock.
<code>ix_cc_ipv4_microblock_low_priority_pkt_handler()</code>	Receive exception packets from IPv4 microblock
<code>ix_cc_ipv4_stackdrv_pkt_handler()</code>	Receives packets from the Stack Driver core component.
<code>ix_cc_ipv4_common_pkt_handler()</code>	Receives packets from any core component other than the Stack Driver.

52.5.3 Messaging API

Table 53-5 lists the functions of the IPv4 Forwarder Message Helper API.

Table 53-5. IPv4 Forwarder Message Helper API

API	Description
<code>ix_cc_ipv4_async_add_route()</code>	Adds a route in the Route Table Manager
<code>ix_cc_ipv4_async_delete_route()</code>	Deletes a route in the RTM
<code>ix_cc_ipv4_async_update_route()</code>	Update an existing route in the RTM
<code>ix_cc_ipv4_async_lookup_route()</code>	Looks up routing information for a given IP address
<code>ix_cc_ipv4_async_purge_routes()</code>	Removes all routes from the RTM
<code>ix_cc_ipv4_async_dump_routes()</code>	Dumps all routes of RTM in memory
<code>ix_cc_ipv4_async_add_next_hop()</code>	Adds the next hop information to the RTM database
<code>ix_cc_ipv4_async_delete_next_hop()</code>	Deletes the next hop information from the RTM database
<code>ix_cc_ipv4_async_update_next_hop()</code>	Updates the next hop information into the RTM database
<code>ix_cc_ipv4_async_get_next_hop()</code>	Retrieves the next hop information from the RTM database
<code>ix_cc_ipv4_async_dump_next_hops()</code>	Dumps all next hops of RTM in memory
<code>ix_cc_ipv4_async_purge_rtm()</code>	Removes all routes and next hops from the RTM database
<code>ix_cc_ipv4_async_set_mtu()</code>	Updates MTU for a given next hop
<code>ix_cc_ipv4_async_set_flags()</code>	Updates flags for a given next hop.

Table 53-5. IPv4 Forwarder Message Helper API (Continued)

API	Description
<code>ix_cc_ipv4_async_get_sleep_time()</code>	Retrieves the number of seconds allocated for calling the ICMP event
<code>ix_cc_ipv4_async_set_sleep_time()</code>	Sets the number of seconds for calling the ICMP event handler
<code>ix_cc_ipv4_async_get_queue_depth()</code>	Retrieves the depth of the ICMP error message queue
<code>ix_cc_ipv4_async_get_packets_to_drain()</code>	Retrieves the maximum number of ICMP error messages to send
<code>ix_cc_ipv4_async_set_packets_to_drain()</code>	Sets the maximum number of ICMP error messages to send
<code>ix_cc_ipv4_async_get_statistics()</code>	Retrieves the statistical report from the IPv4 Forwarder Core Component

52.5.4 Library API

Table 53-6 lists the elements of the IPv4 Forwarder Library API:

Table 53-6. IPv4 Forwarder Library API

API	Description
<code>ix_cc_ipv4_add_route()</code>	Adds a route in the Route Table Manager (RTM)
<code>ix_cc_ipv4_delete_route()</code>	Deletes a route from the RTM
<code>ix_cc_ipv4_update_route()</code>	Updates an existing route in the RTM
<code>ix_cc_ipv4_lookup_route()</code>	Looks up the routing information for a given IP address
<code>ix_cc_ipv4_purge_routes()</code>	Removes all routes from the RTM
<code>ix_cc_ipv4_dump_routes()</code>	Dumps all routes of memory in the RTM
<code>ix_cc_ipv4_add_next_hop()</code>	Adds the next hop information to the RTM
<code>ix_cc_ipv4_delete_next_hop()</code>	Deletes the next hop information from the RTM
<code>ix_cc_ipv4_update_next_hop()</code>	Updates the next hop information into the RTM database
<code>ix_cc_ipv4_get_next_hop()</code>	Retrieves the next hop information from the RTM
<code>ix_cc_ipv4_dump_next_hops()</code>	Dumps all next hops of the RTM
<code>ix_cc_ipv4_purge_rtm()</code>	Internally calls <code>ix_cc_rtmv4_purge()</code> API of the RTM
<code>ix_cc_ipv4_set_mtu()</code>	Calls <code>ix_cc_rtmv4_set_mtu()</code> API of the RTM
<code>ix_cc_ipv4_set_flags()</code>	Internally calls <code>ix_cc_rtmv4_set_flags()</code> API of the RTM
<code>ix_cc_ipv4_get_rtm_handle()</code>	Returns the handle to the RTMv4.
<code>ix_cc_ipv4_get_sleep_time()</code>	Retrieves the number of seconds allocated for calling the ICMP event handler, that is, messages dequeued and sent
<code>ix_cc_ipv4_set_sleep_time()</code>	Sets the number of seconds for calling the ICMP event handler

Table 53-6. IPv4 Forwarder Library API (Continued)

API	Description
<code>ix_cc_ipv4_get_queue_depth()</code>	Retrieves the maximum number of ICMP messages that can fit in the queue
<code>ix_cc_ipv4_get_packets_to_drain()</code>	Retrieves the maximum number of ICMP error messages to send
<code>ix_cc_ipv4_set_packets_to_drain()</code>	Sets the maximum number of ICMP error messages to send
<code>ix_cc_ipv4_set_property()</code>	Sets the dynamic properties of the IPv4 Forwarder Core Component

The IPv6 Core Component performs the following functions:

- Configures IPv6 microblock (static configuration)
- Provides message and packet handlers to receive messages and packets from other core components and IPv6 microblock.
- Generates ICMPv6 error messages
- Validates IP header
- Handles extension headers
- Supports Neighbor Discovery
- Supports Stateless Address Auto Configuration

The microblock performs fast-path forwarding of IPv6 packets. Exception packets (such as packets requiring options processing) are sent to the IPv6 Forwarder Core Component for special handling.

For complete details on the IPv6 Microblock, see [Chapter 25, “IPv6 Forwarder Microblock”](#) and for external APIs see [Chapter 16, “IPv6 Forwarder”](#) of the *IXA Software Building Blocks Reference Manual*.

53.1 Data Flow

The IPv6 Forwarder core component can receive packets and messages from several components of the system—from IPv6 microblock, Interface RX, and Stack Driver Core Component.

53.2 Assumptions and Dependencies

53.2.1 Assumptions

The following assumptions made about the implementation:

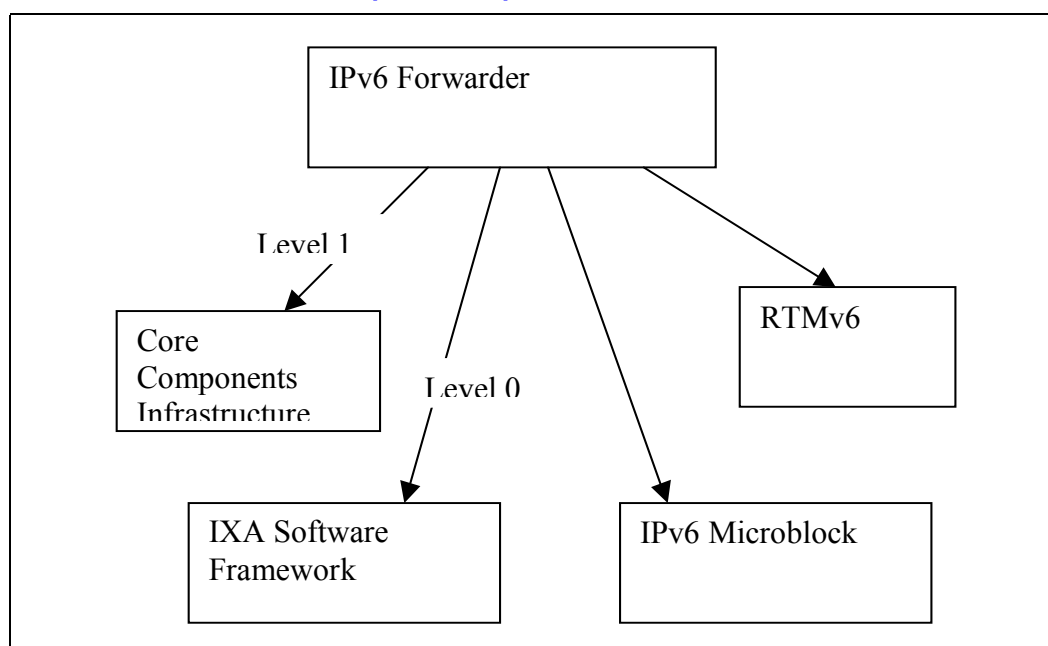
- SNMP is not included in the IPv6 Forwarder. The IPv6 Forwarder keeps the counters (to be detailed later in the document) but does not store these counters in any MIBs. In addition, configuration parameters may not be passed via a MIB. The IPv6 Forwarder exposes an interface for querying statistics and configuration, and it would be up to some other component to do the translation to MIB format. In addition, the statistics kept are not the complete set specified by the MIBs.
- The IPv6 Forwarder does not handle multicast or forwarding. Multicast packets destined for local delivery are delivered to the Stack Driver but is not forwarded beyond the router. However, the core component support multicast packets for the purpose of Neighbor Discovery and Address Auto configuration.

- Traffic Class field in the IP header is ignored. No Traffic Class handling is supported in either the ME component or the core component. The Traffic Class field is treated as an unknown field and is not changed by the IPv6 Forwarder.
- All input and output parameters must be represented in host byte order.
- The IPv6 Forwarder does not support scoped addresses.

53.2.2 Component Dependencies

IPv6 Forwarder Core Component uses the services of IXA software framework for allocating memory, freeing memory, patching symbols, 64-bit counter support and system registry to retrieve static parameters. IPv6 Forwarder Core Component uses core components infrastructure services for event handling, message handling and packet handling between core components and IPV6 microblock. Services of RTMv6 is used to add, delete and look up route information. IPv6 Forwarder receives exception packets from the IPv6 microblock for further processing. Figure 53-1 illustrates the IPv6 component dependencies.

Figure 53-1. IPV6 Forwarder Core Component Dependencies



53.3 Configuration and Initialization

IPv6 Forwarder uses the registry for static configuration if system supports registry services. If there is no registry support, then the IPv6 forwarder uses compile time defined values for static configurations by using one of its header files.

The registry is used for the following static configuration parameters:

- Route table size hint—indicates approximately how many routes are supported
- Route table type

- Size of the next hop table
- RTMv6 lookup Memory Type
- Next hop database memory type
- Number of IPv6 Microengines on which the microblocks run.

IPv6 Forwarder core component supports dynamic property updates as specified in [Section 39.2.1, “Dynamic Properties and Clients”](#) on page 666.

Note: The properties of interfaces are updated dynamically by the CP-PDK. The registry is used to keep initial property values for the start-up configuration and in system configurations without CP-PDK. The messaging mechanism is used to update property values during run-time.

IPv6 Forwarder Core Component is responsible for allocating memory and patching symbols during initialization. The core component uses RTMv6 and the lookup library to manage routing tables.

In addition, the IPv6 Forwarder maintains statistics for the forwarder (both core component and microblock) and Route Table Manager IPv6.

The IPv6 microblock statistics include both 32-bit and 64-bit counters. Counters that require 64-bit support are actually maintained within the core component (created during initialization by the calling the resource manager) and periodically updated.

[Table 53-1](#) and [Table 53-2](#) shows the counters for both incoming and outgoing interfaces maintained by the core component and microblocks. The incoming packet statistics are maintained for all ports present on the blade that the core component is running on. Outgoing packet statistics are maintained for all ports on all blades.

[Table 53-1](#) [Table 53-2](#) lists the statistics supported by the IPV6 Forwarder core component and their descriptions.

Table 53-1. Counter Offsets in Incoming Statistics Block

Counter	Meaning
IPv6CoreInNoRoutes	Count of packets discarded, as there was no route to the destination.
IPv6CoreInUnknownProtos	Count of local packets discarded because of unsupported protocol.
IPv6CoreInDiscards	Count of IP packets with no problems, but were still discarded (e.g.: out of buffer space)
IPv6CoreInDelivers	Count of IP packets delivered successfully to IP user protocols
IPv6CoreInMulticast	Number of multicast IP packets received
IPv6CoreInMulticastOctets	Number of bytes received in IP multicast packets

Table 53-2. Counter Offsets in Outgoing Statistics Block

Counter	Meaning
IPv6OutForwDatagrams	Count of IP packets for which a path to the destination was found(Increment count on outgoing interface)
IPv6OutTransmits	Count of forwarded packets supplied to lower layer
IPv6OutOctets	Number of bytes in IP packets delivered to lower layers for transmission
IPv6CoreOutRequests	Count of IP packets sent down from protocols running on this machine
IPv6CoreOutNoRoutes	Count of locally generated packets that were dropped as no route was found to the destination.
IPv6CoreOutDiscards	Count of packets with no problems that should have been transmitted, but were discarded.
IPv6CoreOutMulticast	Number of multicast IP packets transmitted
IPv6CoreOutMulticastOctets	Number of bytes transmitted in IP multicast packets

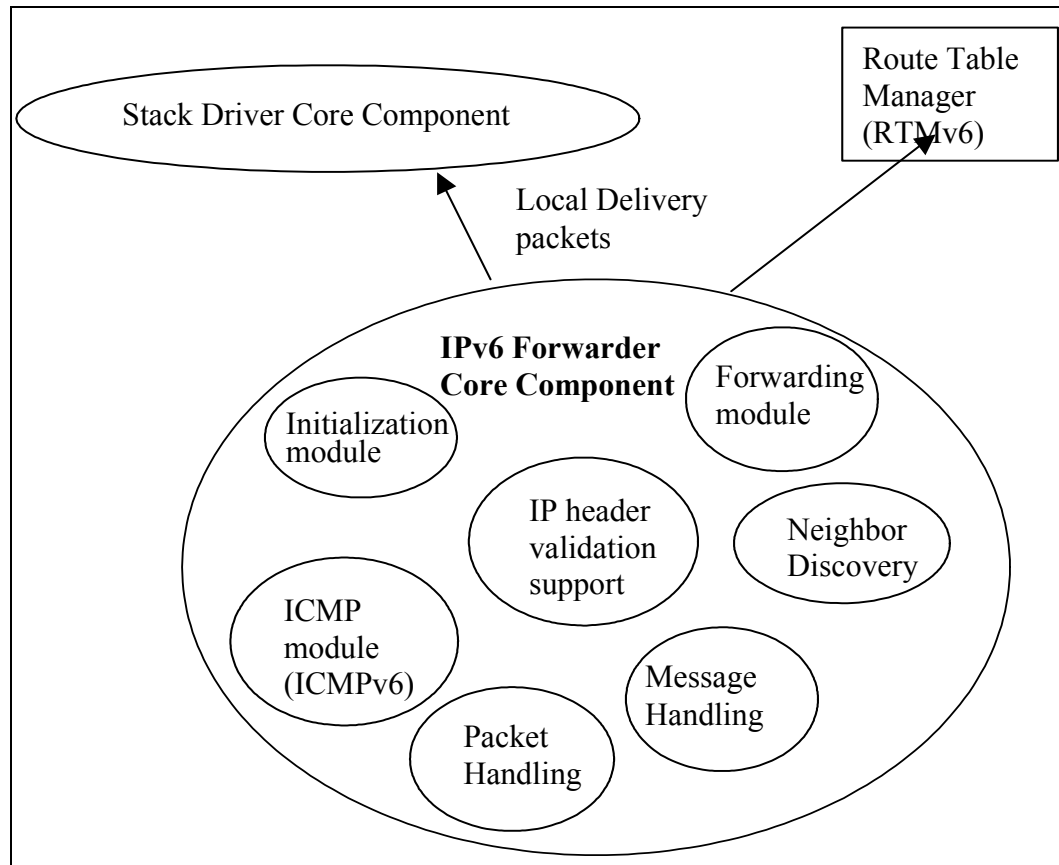
53.4 Modularity

IPv6 Forwarder Core Component provides compile time switches for RFC 2460 SHOULD features (RFC 2640 MUST features are supported by default) and RFC 2373 features.

IPv6 Forwarder is comprised of several modules:

- Forwarding and IP Header Validation Module
- Packet Handling Module
- Message Handling Module
- ICMPv6 Module
- Neighbor Discovery Module
- Address Auto-configuration Module

Figure 24 shows the different modules of IPv6 Forwarder core component and the interaction with RTMv6 and stack driver core component.

Figure 53-2. Modularity of IPv6 Core Component


53.4.1 Forwarding and IP Header validation Module

When trying to forward packets, IPv6 Forwarder Core Component needs to do IP header validation (RFC2460 and RFC2373 checks) and also check packets for local delivery.

53.4.1.1 Forwarding

IPv6 Forwarder Core Component can receive packets from Stack Driver Core Component to forward the packets and it can also receive packets from the microblock to forward the packets on Core side.

It performs a route lookup on the destination IP address. There is no HopLimit field decrement required for packets coming from stack Driver as Stack Driver Core Component takes care of decrementing the HopLimit Field. Another important point is that when IPv6 Forwarder Core Component sees a problem with the packet (For example, route lookup failed) and drops it.

On the other hand, when IPv6 Forwarder receives packets from microblock/other core components for which it has to do forwarding, it does IP header validation for the packets and also takes care of HopLimit field decrement. If the microblock has already performed lookup (for certain exception packets), it is assumed that the microblock fills up meta data before sending the packets to IPv6

Forwarder Core Component. If the meta data is not available, the IPv6 forwarder Core Component performs lookup using the lookup library for route information. The meta data is updated and the packet is sent to the next Core Component. Packets meant for local delivery are sent to Stack Driver Core Component.

53.4.1.2 IP Header Validation

There are two kinds of checks that need to be performed for IP header validation. As per RFC2460 and RFC2373 the following checks are performed:

- RFC2460 checks
 - Before a router can process an IP packet, it must perform the following basic validity checks on the packet's IP header to ensure that the header is meaningful. If the packet fails any of the following tests, it is discarded and ICMP error message is generated.
 - The IP version number must be 6
 - The packet size must be at least 40 bytes
 - Packets with HopLimit equal to zero (after a routing decision is made) are dropped
- RFC2373 checks
 - Packets with link-local source addresses are dropped
 - Packets with multicast source address (the MSB of the source address is 0xFF) are dropped
 - Packets with destination set to 0 are dropped
 - Packets with destination::1 are sent back to the stack driver (packet originated locally and needs to be looped back)
 - Packets with multicast destination address (the MSB of the destination address is 0xFF) -
 - If the destination address is an “All-routers” multicast address it is sent up to the Control Plane.
 - If the destination address is an “All-nodes” (FF02::1) or “solicited node” (link local scope) are sent to the Neighbor Discovery module.
 - All other multicast packets are sent to the Control Plane.
 - Packets with source address set to loopback (::1) are dropped.
 - Packets with a link local destination address are dropped.
 - Packets with a site local destination address are dropped.
 - Packets with next header set to Hop-by-Hop are sent to the Control Plane.

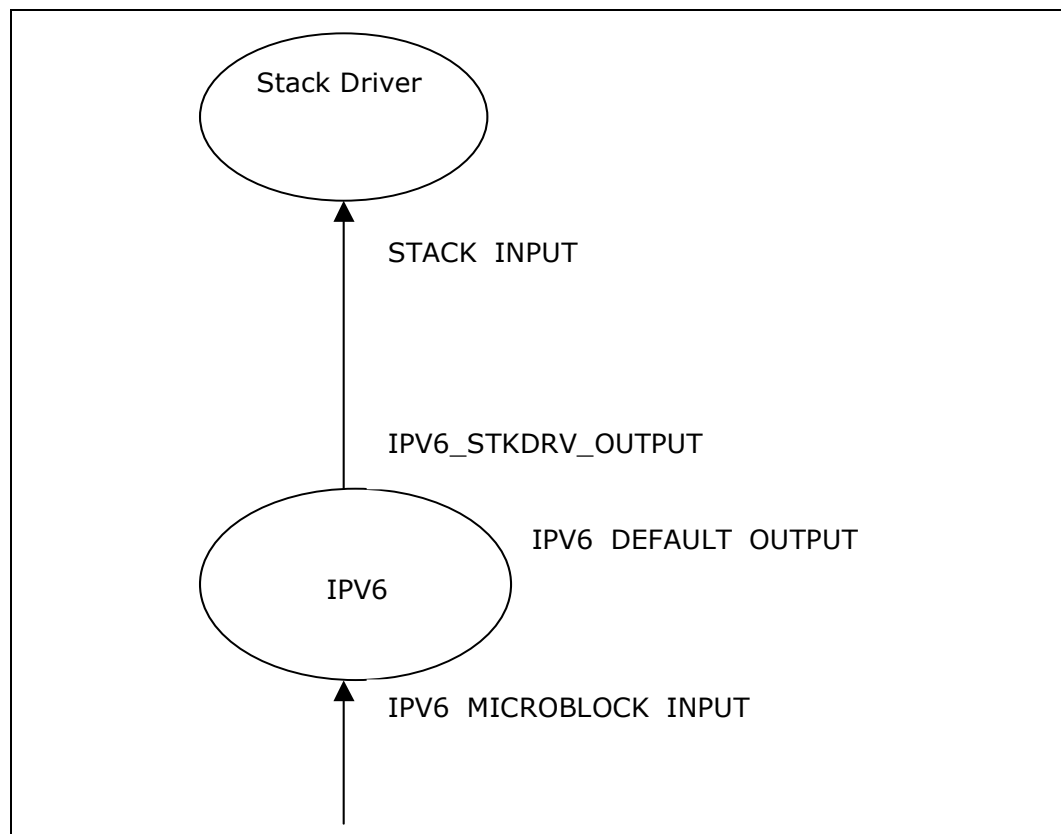
53.4.1.3 Identifying packets for local delivery

When a router receives an IP packet, it must consider whether the packet is addressed to itself (and should be delivered locally) or the packet is addressed to another system (and should be handled by the forwarder). The packet is delivered locally and not considered for forwarding in the following case:

- the next hop information has the NH_FLAGS_LOCAL_BIT set for packets destined for the local node.

The packets meant for local delivery are sent to the Stack Driver Core Component by IPv6 Forwarder Core Component. The bindings between IPv6 Forwarder Core Component and Stack Driver Core Component are described in bindings.h. Figure 53-3 illustrates this binding.

Figure 53-3. IPv6 Forwarder Core Component Bindings



IPv6 Forwarder Core Component does route lookup when trying to forward packets. If route entry is marked local, it increments count of local delivery packets and sends the packet to Stack Driver Core Component.

53.4.2 Packet Handling Module

Packet handling module is responsible to receive packet and call the appropriate low-level functions to process the packet from IPV6 Forwarder microblock, stack driver and other core components. Therefore, it provides three different packet handlers to receive packet from microblock, stack driver and other core components. We decided to have separate packet handler for stack driver due to special packet handling. Detailed descriptions of packet handler APIs are explained in `ix_cc_ipv4_microblock_high_priority_pkt_handler()`, `ix_cc_ipv4_stackdrv_pkt_handler()` and `ix_cc_ipv4_common_pkt_handler()`.

53.4.3 Message Handling Module

Message handling module is responsible for receiving messages from core components and calling the appropriate library API function to process the message. The message handler is described in `ix_cc_ipv4_msg_handler()` and corresponding library APIs are described in [Section 15.4](#), “Library API” on page 227.

53.4.4 ICMPv6

This module implements the building and sending of ICMPv6 messages for the IPv6 Forwarder Core Component. These messages include error messages defined in the ICMPv6 RFC [RFC2463] and messages for Neighbor Discovery, described in [RFC 2461]. Once the IPv6 Forwarder determines that an ICMP error needs to be generated, it calls one of the appropriate functions of the ICMPv6 module. These functions in turn verifies that the packet can cause an ICMP error message to be sent and build the ICMP error message and queue it to be sent.

The ICMPv6 messages supported are

- Parameter Problem

If an IPv6 node processing a packet finds a problem with a field in the IPv6 header or extension headers such that it cannot complete processing the packet, it must discard the packet and should send an ICMPv6 Parameter Problem message to the packet's source, indicating the type and location of the problem.

- Destination Unreachable

A Destination Unreachable message should be generated by a router, or by the IPv6 layer in the originating node, in response to a packet that cannot be delivered to its destination address for reasons other than congestion.

We support the following Destination Unreachable error messages:

- No route to destination
- Communication with destination administratively prohibited
- Address unreachable
- Port unreachable

- Time Exceeded

If a router receives a packet with a Hop Limit of zero, or a router decrements a packet's Hop Limit to zero, it MUST discard the packet and send an ICMPv6 Time Exceeded message with Code 0 to the source of the packet. This indicates either a routing loop or too small an initial Hop Limit value.

- Packet too big

This error is sent if the packet length exceeds the MTU of the outgoing link. The error includes the correct MTU value to be used for all future packets coming from the host.

In addition to the above error messages, ICMPv6 also generates the following messages defined by IPv6 Neighbor Discovery [RFC2461].

- Router Advertisement
- Neighbor Solicitation
- Neighbor Advertisement

53.4.4.1 RFC2463 MUST Features for ICMPv6 Error Message Processing

- If an ICMPv6 error message of unknown type is received, it MUST be passed to the upper layer.
- If an ICMPv6 informational message of unknown type is received, it MUST be silently discarded.
- Every ICMPv6 error message (type < 128) includes as much of the IPv6 offending (invoking) packet (the packet that caused the error) as will fit without making the error message packet exceed the minimum IPv6 MTU.
- In those cases where the Internet-layer protocol is required to pass an ICMPv6 error message to the upper-layer process, the upper-layer protocol type is extracted from the original packet (contained in the body of the ICMPv6 error message) and used to select the appropriate upper-layer process to handle the error. In case the upper layer protocol cannot be determined, the packet must be silently discarded.
- An ICMPv6 error message MUST NOT be sent as a result of receiving an ICMPv6 error message, or to a source whose address cannot be uniquely determined.
- The ICMPv6 packets must be Rate Limited. The rate limiting is timer-based.

Rate limiting—RFC2463 states

In order to limit the bandwidth and forwarding costs incurred sending ICMPv6 error messages, an IPv6 node MUST limit the rate of ICMPv6 error messages it sends. This situation may occur when a source sending a stream of erroneous packets fails to heed the resulting ICMPv6 error messages. There are two types of rate limiting—Timer based and bandwidth based.

53.4.5 Neighbor Discovery

This module provides Neighbor Discovery (ND) Services like Address Resolution, Neighbor Unreachability Detection and Next Hop Determination. It also provides the Duplicate Address Detection functionality for Address Autoconfiguration. Neighbor Discovery is described in [RFC2461].

53.4.5.1 Supported Neighbor Discovery Messages

The ND module implements the following messages defined in [RFC2461].

- Neighbor Solicitation
This message is sent to request the link-layer address of a target node. This message can be used for Address Resolution, Neighbor Unreachability Detection or Duplicate Address Detection. When used for Address Resolution, this message is multicast to the solicited-node multicast address. For the other two uses, the message is unicast to the specific target.
- Neighbor Advertisement
This message is typically sent in response to a Neighbor Solicitation message. It may also be sent unsolicited in order to quickly propagate address information. When sending unsolicited, the destination address is the all-node multicast address.

53.4.5.2 Address Resolution

The sub-section describes the Address Resolution portion of the Neighbor Discovery component.

Unlike the ARP library, which is located on the egress, with the Ethernet TX core component, the Neighbor Discovery resides on the ingress, and is integrated with the IPv6 Forwarder core component. This makes it easier for the Neighbor Discovery component to send and receive the ICMPv6 packets that it uses for Address Resolution and so on.

The ND component has two main functions regarding Address Resolution: Performing the actual Address Resolution to update the Neighbor Cache, and synchronizing the L2 table with information present in the Neighbor Cache.

All calls to update the L2 table go through the ND component, which updates its Neighbor Cache before actually updating the L2 table.

Also, the Ethernet TX core component can call the ND component to request address resolution to be performed on any packet which causes a miss in the L2 table.

53.4.6 Address Autoconfiguration

The IPv6 Core Component uses Address Autoconfiguration to determine its link local address as defined in [RFC2462]. According to the RFC, routers cannot use Autoconfiguration techniques to configure site local and global addresses. Hence, these need to be manually configured. The core component uses Duplicate Address Detection for ALL addresses to make sure they are unique.

There is no separate module for Address Autoconfiguration. Link-local address assignment is built into the initialization code of the core component, and the Duplicate Address Detection is implemented within the Neighbor Discovery module.

53.5 External API

This section lists the IPv6 Forwarder core components. For complete details, see [Chapter 16](#), “IPv6 Forwarder” of the *IXA Software Building Blocks Reference Manual*.

53.5.1 Data Structures

Table 53-1 lists the data structures, types and macros defined in IPv6 Forwarder Core Component.

Table 53-1. IPv6 Forwarder Data Structures, Types and Macros

Data Structures, Types and Macros	Description
<code>IX_CC_RTMOV6_DUMP_ROUTE_SIZE</code>	Defines a macro for calculating the size of memory to dump RTMv6 routes
<code>ix_cc_rtmv6_nhld</code>	Defines the data type for next hop identifier
<code>Reserved Next Hop Ids</code>	Defines the special next hops added during initialization
<code>ix_cc_rtmv6_next_hop_info</code>	Defines the data structure for next hop information
<code>ix_cc_ipv6_dump_data</code>	Describes memory dump information.
<code>ix_cc_in_ipv6_stats_data</code>	Defines data structure for statistics
<code>ix_cc_out_ipv6_stats_data</code>	Defines data structure for statistics
<code>ix_cc_ipv6_stats_data</code>	Defines counter information.
<code>ix_cc_ipv6_icmp_err_type</code>	Enumeration of the ICMPv6 error message types
<code>ix_cc_ipv6_icmp_err_code</code>	Enumeration of the ICMPv6 codes to accompany the error message types

53.5.2 Core Component Infrastructure API

Table 53-2 lists the IPV6 Forwarder Core Component Infrastructure API.

Table 53-2. IPV6 Forwarder Core Component Infrastructure API

API	Description
<code>ix_cc_ipv6_init()</code>	Initializes the IPv6 Forwarder component
<code>ix_cc_ipv6_fini()</code>	Terminates services from the IPv6 Forwarder
<code>ix_cc_ipv6_msg_handler()</code>	Message handler function for IPv6 Forwarder
<code>ix_cc_ipv6_microblock_high_priority_pkt_handler()</code>	Receives exception packets from high priority queue of IPv6 microblock
<code>ix_cc_ipv6_microblock_low_priority_pkt_handler()</code>	Receive exception packets from IPv6 microblock
<code>ix_cc_ipv6_stackdrv_pkt_handler()</code>	Receives packets from stack driver core component
<code>ix_cc_ipv6_common_pkt_handler()</code>	Receive packets from any core component other than stack driver—interface RX core component

53.5.3 Message Helper API

Table 53-3 lists the IPv6 Forwarder message helper API.

Table 53-3. IPv6 Forwarder Message Helper API

API	Description
<code>ix_cc_ipv6_async_add_prefix()</code>	Adds a prefix (route) in RTMv6
<code>ix_cc_ipv6_async_delete_prefix()</code>	Deletes a prefix (route) in the RTMv6 database
<code>ix_cc_ipv6_async_update_prefix()</code>	Updates an existing route in the RTMv6 database
<code>ix_cc_ipv6_async_lookup_prefix()</code>	Looks up routing information for a given IP address
<code>ix_cc_ipv6_async_purge_prefixes()</code>	Removes all prefixes from the RTMv6 database
<code>ix_cc_ipv6_async_dump_prefixes()</code>	Dumps all prefixes stored by the RTMv6 database in memory
<code>ix_cc_ipv6_async_add_next_hop()</code>	Adds next hop information to NH database
<code>ix_cc_ipv6_async_delete_next_hop()</code>	Deletes the next hop information from the RTMv6 database
<code>ix_cc_ipv6_async_update_next_hop()</code>	Updates the next hop information into the RTMv6 database
<code>ix_cc_ipv6_async_get_next_hop()</code>	Retrieves the next hop information from the RTMv6 database
<code>ix_cc_ipv6_async_dump_next_hops()</code>	Dumps all next hops of the RTMv6 in memory
<code>ix_cc_ipv6_async_purge_rtm()</code>	Removes all prefixes and next hops from the RTMv6 database
<code>ix_cc_ipv6_async_set_mtu()</code>	Updates MTU for a given next hop
<code>ix_cc_ipv6_async_set_flags()</code>	Updates flags for a given next hop
<code>ix_cc_ipv6_async_get_rate_limit_time()</code>	Retrieves the time interval for rate limiting in milliseconds
<code>ix_cc_ipv6_async_set_rate_limit_time()</code>	Sets the rate limit interval in milliseconds
<code>ix_cc_ipv6_async_get_queue_depth()</code>	Retrieves the depth of the ICMP error message queue
<code>ix_cc_ipv6_async_get_statistics()</code>	Retrieves statistics report from IPv6 Forwarder core component
<code>ix_cc_ipv6_async_perform_addr_resolution()</code>	Requests address resolution to be performed for the destination IP address contained in the packet
<code>ix_cc_ipv6_async_add_neighbor()</code>	Updates the neighbor cache and add an entry into the L2 table
<code>ix_cc_ipv6_async_del_neighbor()</code>	Deleted the contents located at the supplied L2Index in the L2 table
<code>ix_cc_ipv6_sync_add_neighbor()</code>	Updates the neighbor cache and adds an entry into the L2 table in a synchronous manner.
<code>ix_cc_ipv6_sync_del_neighbor()</code>	Delete the contents of the L2Index in the L2 table in a synchronous manner.
<code>ix_cc_ipv6_async_send_icmp_error()</code>	Sends an ICMP error message
<code>ix_cc_ipv6_async_send_icmp_info_message()</code>	Sends ICMP informational messages

53.5.4 Library API

Table 53-4 lists the IPV6 Forwarder library API.

Table 53-4. IPV6 Forwarder Library API

API	Description
<code>ix_cc_ipv6_add_prefix()</code>	Adds a prefix in RTMv6
<code>ix_cc_ipv6_delete_prefix()</code>	Deletes a prefix (route) from RTMv6
<code>ix_cc_ipv6_update_prefix()</code>	Updates a prefix in RTMv6
<code>ix_cc_ipv6_lookup_prefix()</code>	Looks up routing information for a given IP address
<code>ix_cc_ipv6_purge_prefixes()</code>	Removes all prefixes from the RTMv6
<code>ix_cc_ipv6_dump_prefixes()</code>	Dumps all prefixes stored by RTMv6 in memory
<code>ix_cc_ipv6_add_next_hop()</code>	Adds next hop information to NH database
<code>ix_cc_ipv6_delete_next_hop()</code>	Deletes next hop information from RTMv6
<code>ix_cc_ipv6_update_next_hop()</code>	Updates next hop information into RTMv6 database
<code>ix_cc_ipv6_get_next_hop()</code>	Retrieves next hop information from RTMv6
<code>ix_cc_ipv6_dump_next_hops()</code>	Dumps all next hops of RTM in memory
<code>ix_cc_ipv6_purge_rtm()</code>	Removes all routes and next hops from RTM database
<code>ix_cc_ipv6_set_mtu()</code>	Updates MTU for a given next hop
<code>ix_cc_ipv6_set_flags()</code>	Updates flags for a given next hop
<code>ix_cc_ipv6_get_rate_limit_time()</code>	Retrieves the time interval for rate limiting in milliseconds
<code>ix_cc_ipv6_set_rate_limit_time()</code>	Sets the rate limit interval in milliseconds
<code>ix_cc_ipv6_get_queue_depth()</code>	Retrieves the depth of the ICMP error message queue
<code>ix_cc_ipv6_get_statistics()</code>	Retrieves statistics report from IPV6 Forwarder core component
<code>ix_cc_ipv6_set_property()</code>	Sets dynamic properties of IPV6 Forwarder core component
<code>ix_cc_ipv6_perform_addr_resolution()</code>	Requests address resolution to be performed for the destination IP address contained in the packet
<code>ix_cc_ipv6_add_neighbor()</code>	Updates the neighbor cache and add an entry into the L2 table
<code>ix_cc_ipv6_del_neighbor()</code>	Deletes the contents located at the supplied L2Index in the L2 table
<code>ix_cc_ipv6_send_icmp_error()</code>	Sends an ICMP error message
<code>ix_cc_ipv6_send_icmp_info_message()</code>	Sends ICMP informational messages

IPv6 To IPv4 Tunneling Core Component

54

54.1 Overview

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

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

The following functionality is provided:

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

For complete details on the IPv6 Microblock, see [Chapter 26, “IPv6 To IPv4 Tunneling Microblock”](#) and for external APIs see [Chapter 17, “IPV6 to IPV4 Tunneling”](#) of the *IXA Software Building Blocks Reference Manual*.

54.2 Data Flow

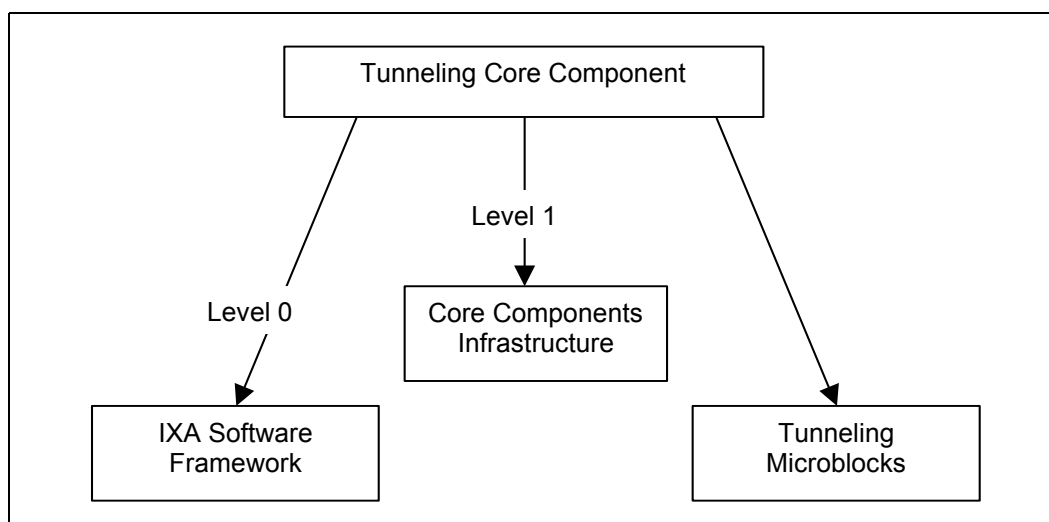
The Tunneling core component receives packets from the tunneling microblocks, IPv4, or IPv6 Forwarder core components.

54.3 Assumptions and Dependencies

54.3.1 Dependencies

The Tunneling Core Component uses the services of the IXA software framework for allocating memory, freeing memory, patching symbols, accessing shared resources such as packet data, and retrieving static parameters. The Core Component Infrastructure is used for event, message and packet handling, and for sending messages and packets to the IPv4 and IPv6 core components.

Figure 54-1. Tunneling Core Component Dependencies



54.4 Configuration and Initialization

54.4.1 Configuration Parameters

The tunneling core component uses the system registry to store its static configuration parameters. A description of each supported parameter follows.

54.4.1.1 Size of End Tunnel Next Hop Table

This parameter specifies the number of entries needed for the end tunnel next hop table.

54.4.1.2 Size of Start Tunnel Next Hop Table

This parameter specifies the number of entries needed for the start tunnel next hop table.

54.4.1.3 Format of Ingress Source Validation List

This parameter specifies the format used to store the acceptable prefixes for ending endpoints of configured tunnels. The following choices are available:

- Per-tunnel array with fixed (≤ 12) number of prefixes.
- Per-tunnel chained list of arrays (12 prefixes per array).
- Global trie table, where each prefix can apply to multiple tunnels.

54.4.1.4 Size Hint for Ingress Source Validation List

This parameter specifies how many entries are needed for the ingress source validation list. The interpretation of this parameter depends on the value specified for the format of the ingress source validation list. If the array format is used, then this parameter indicates how many array blocks are required. If the trie format is used, then this parameter indicates the number of prefixes supported.

54.4.2 Initialization

The tunneling core component initializes the memory that is shared with the tunneling microblocks. This includes the tunnel next hop tables, ingress source validation list and packet counters. The core component also patches symbols needed by the tunneling microblocks.

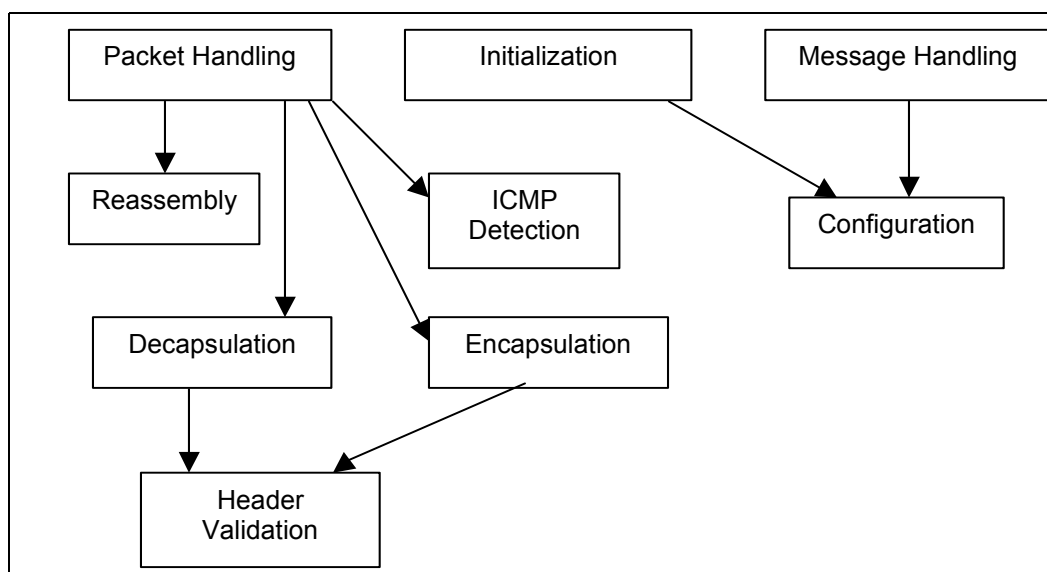
54.5 Modularity

The Tunneling Core Component is composed of the following modules:

- Initialization Module
- Message Handling Module
- Packet Handling Module
- Decapsulation Module
- Encapsulation Module
- Reassembly Support Module
- ICMP Error Message Support Module
- Tunneling Header Validation Support Module
- Configuration Support Module

Figure 54-2 illustrates the interdependencies among these modules.

Figure 54-2. Modularity of Tunneling Core Component



54.5.1 Initialization Module

The initialization module is responsible for performing the initialization tasks described in [Section 54.4, “Configuration and Initialization” on page 796](#).

54.5.2 Message Handling Module

The message handling module receives messages from the calling applications and uses the services of the appropriate support module to process the message.

54.5.3 Packet Handling Module

The packet handling module receives packets from the tunneling microblocks and other core components and calls the appropriate support modules to process the packet. Exception packets from the microblocks are processed with a separate packet handler from packets from other core components.

54.5.3.1 Microblock Packet Handler

This packet handler handles packets that were received from the decapsulation and encapsulation microblocks. These packets can be of the following types:

- Non-tunneled IPv4 packets from the decapsulation microblock that are destined for the local stack.

Non-Tunneled IPv4 Packets—The decapsulation microblock can encounter packets addressed to a local interface used for tunneling, but which are not themselves tunneled packets. Such packets are sent to the tunneling core component to be forwarded to the local stack. When the tunneling core component receives this type of packet, it forwards the packet to the stack driver as a local packet.

- Automatic tunneled packets IPv6 packets from the decapsulation microblock that are destined for the local stack.

Automatic Tunneled IPv6 Packets—When the decapsulation microblock detects a valid automatic tunneled packet addressed to a local interface, it forwards the packet to the tunneling core component to be delivered to the local stack. When the tunneling core component receives this type of packet, it forwards the packet to the stack driver as a local packet.

- Fragments of a tunneled packet that require reassembly.

Fragmented Packets—When the decapsulation microblock detects a fragmented tunneled packet addressed to a local interface, it forwards the packet to the tunneling core component. The tunneling core component uses the services of the reassembly support module to reassemble the packet. Once the packet has been reassembled, the decapsulation module processes the packet.

- Packets received from the encapsulation microblock for which the tunnel path MTU is smaller than the packet length.

Larger-than-MTU Packets—When the encapsulation microblock detects that an encapsulated packet's length exceeds both the minimum IPv6 MTU and the path MTU for the tunnel, it sends the packet to the tunneling core component. The tunneling core component invokes the services of the IPv6 Forwarder core component to send an ICMPv6 “Packet too Big” message.

54.5.3.2 IPv4 Packet Handler

This packet handler handles IPv4 packets that were processed by the IPv4 Forwarder core component and might need decapsulation. Packets that require decapsulation are handled by the Decapsulation module.

54.5.3.3 IPv6 Packet Handler

This packet handler handles IPv6 packets that were processed by the IPv6 Forwarder Core Component and might need encapsulation. Packets that require encapsulation are handled by the Encapsulation module.

54.5.4 Decapsulation Module

This module performs decapsulation in a manner similar to that used in the decapsulation microblock. The header is validated according to the compiler switches set and the configuration specified for the end tunnel endpoint. If the packet passes validation, it is either sent to the local stack (if an automatic tunneled packet) or sent to the IPv6 Forwarder Core Component for further processing.

54.5.5 Encapsulation Module

This module encapsulates packets in a manner similar to that used in the encapsulation microblock, and passes the packet to the IPv4 Forwarder Core Component for further processing.

54.5.6 Reassembly Support Module

This module handles the details involved in reassembling a fragmented packet.

54.5.7 ICMP Error Message Support Module

This module handles ICMP error messages received as a result of sending tunneled packets.

54.5.7.1 Packet Too Big Messages

“Packet too Big” messages that were received for tunneled packets previously sent on a local interface are not processed by the core component. They are simply forwarded to the local stack, where they can be used for path MTU discovery.

54.5.7.2 Other Error Messages

A compiler option can be set to control whether the core component performs ICMP error message reflection. If the option is set, then this module scans incoming ICMP messages for errors (other than “Packet too Big”) that were received for tunneled packets previously sent on a local interface. If messages of this type are detected and they contain sufficient information from the original packet, a corresponding ICMPv6 error message is sent to the original IPv6 sender of the packet. The ICMPv6 message sending facilities of the IPv6 forwarder core component is used to queue and send the ICMPv6 message.

54.5.8 Tunneling Header Validation Support Module

This module performs header validation for packets requiring encapsulation or decapsulation. The following types of validation might be performed, depending on the circumstances:

- RFC 2893 IPv4 Source Address Checks
- RFC 2893 IPv6 Source Address Checks
- RFC 2893 Automatic Tunnel Embedded Address Checks
- RFC 3056 6to4 Embedded Address Checks

54.5.8.1 RFC 2893 IPv4 Source Address Checks

This type of validation is performed on packets being decapsulated. The IPv4 source address is checked to ensure it is not a broadcast, multicast, loopback or unspecified address. This type of validation is performed only if a corresponding compiler option is set.

54.5.8.2 RFC 2893 IPv6 Source Address Checks

This type of validation is performed on packets being decapsulated. The IPv6 source address is checked to ensure it is not a multicast address, loopback address, unspecified address, or an IPv4 compatible address that embeds a broadcast, multicast, loopback or unspecified address. This type of validation is performed only if a corresponding compiler option is set.

54.5.8.3 RFC 2893 Automatic Tunnel Embedded Address Checks

This type of validation is performed on packets being encapsulated for an automatic tunnel. The IPv4 address embedded in the IPv6 destination address is checked to ensure it is not a broadcast, multicast, loopback or unspecified address.

54.5.8.4 RFC 3056 6to4 Embedded Address Checks

This type of validation is performed on packets being encapsulated or decapsulated for a 6to4 tunnel. The source and destination IPv6 addresses of the packet are checked for the 2002::/16 prefix, and if detected, the embedded IPv4 address is checked to ensure it is not a broadcast, multicast, loopback, unspecified or private address.

54.5.9 Configuration Support Module

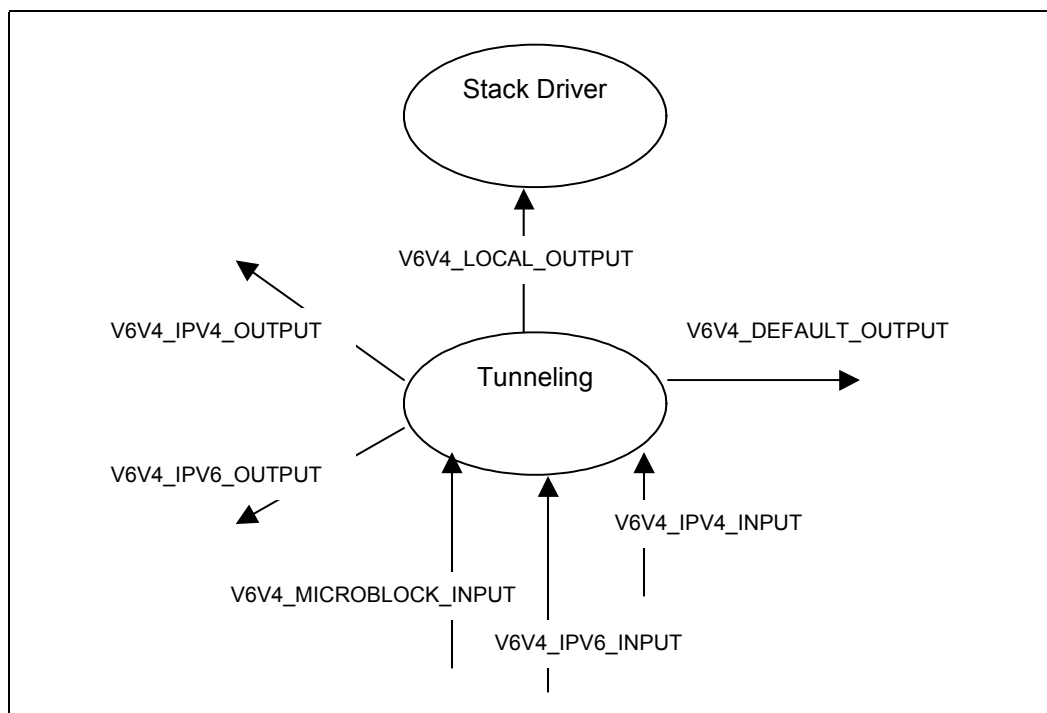
This module maintains the tunnel configuration information in memory. This includes the start and end tunnel next hop tables and the ingress source validation list for end tunnel endpoints.

The ingress source list can be maintained in one of two possible formats: a list of array blocks each containing a fixed number of entries and a trie-based format. Because the trie format used for the tunnel ingress source list is not the same as that supported by the lookup library, the lookup library is not used to manage the ingress source list.

54.6 Bindings

The Tunneling Core Component has three packet inputs, one for packets from the microblocks, one for IPv4 packets from other core components and one for IPv6 packets from other core components. There are four outputs, one for local packets, one for encapsulated IPv4 packets, one for decapsulated IPv6 packets, and a default output for packets that did not require handling by the Tunneling Core Component. [Figure 54-3](#) illustrates these bindings.

Figure 54-3. Tunneling Core Component Bindings



54.7 External API

This section lists the IPv6 Tunneling core components. For complete details, see [Chapter 17](#), “IPv6 to IPv4 Tunneling” of the *IXA Software Building Blocks Reference Manual*.

54.7.1 Data Structures, Types and Macros

Table 54-1 lists the IPv6-IPv4 tunneling data structures, types and macros.

Table 54-1. IPv6-IPv4 Tunneling Data Structures, Types and Macros

Data Structures, Types and Macros	
<code>ix_cc_v6v4_tunnel_handle</code>	Tunneling core component's handle to a start or end tunnel next hop information
<code>ix_cc_v6v4_end_tunnel_config</code>	Contains the configuration information for an end tunnel endpoint
<code>ix_cc_v6v4_end_tunnel_info</code>	Contains the information describing an end tunnel endpoint
End Tunnel Flags	Defined for end tunnel endpoints
<code>ix_cc_v6v4_ingress_source_entry</code>	Describes an entry in an ingress source validation list for an end tunnel endpoint
<code>ix_cc_v6v4_interface_id</code>	Defines an identifier for an interface used for tunneling

Table 54-1. IPv6-IPv4 Tunneling Data Structures, Types and Macros

Data Structures, Types and Macros	
<code>ix_cc_v6v4_start_tunnel_config</code>	Contains the configuration information for a start tunnel endpoint
<code>ix_cc_v6v4_start_tunnel_info</code>	Contains all information associated with a start tunnel endpoint
<code>Start Tunnel Flags</code>	defined for start tunnel endpoints
<code>ix_cc_v6v4_statistics</code>	Describes the packet counters maintained by the Tunneling core component
<code>Option Values</code>	Selects boolean options in the Tunneling core component API

54.7.2 Core Component Infrastructure API

Table 54-2 lists the Tunneling core component Infrastructure API.

Table 54-2. IPv6 to IPv4 Tunneling Core Component Infrastructure AP

API	Description
<code>ix_cc_v6v4_init()</code>	Initializes the Tunneling core component
<code>ix_cc_v6v4_fini()</code>	terminate the services of the Tunneling core component
<code>ix_cc_v6v4_msg_handler()</code>	Receives messages from the calling applications
<code>ix_cc_v6v4_microblock_pkt_handler()</code>	Handles packets received from the tunneling microblocks
<code>ix_cc_v6v4_ipv6_pkt_handler()</code>	Handles packets received from the IPv6 forwarder that might need encapsulation
<code>ix_cc_v6v4_ipv4_pkt_handler()</code>	Handles packets received from the IPv4 forwarder that might need decapsulation

54.7.3 Message Helper API

Table 54-3 lists the IPv6 to IPv4 Tunneling core component message helper API.

Table 54-3. IPv6 to IPv4 Tunneling Core Component Message Helper API

API	Description
<code>ix_cc_v6v4_async_add_end_tunnel()</code>	Sends a message of type <code>IX_CC_V6V4_MSG_ADD_END_TUNNEL</code> to the Tunneling core component
<code>ix_cc_v6v4_async_delete_end_tunnel()</code>	Sends a message of type <code>IX_CC_V6V4_MSG_DELETE_END_TUNNEL</code> to the Tunneling core component
<code>ix_cc_v6v4_async_set_decap_tos_option()</code>	Sends a message of type <code>IX_CC_V6V4_MSG_SET_DECAP_TOS_OPTION</code> to the Tunneling core component
<code>ix_cc_v6v4_async_set_src_validation()</code>	Sends a message of type <code>IX_CC_V6V4_MSG_SET_SRC_VALIDATION</code> to the Tunneling core component.

Table 54-3. IPv6 to IPv4 Tunneling Core Component Message Helper API (Continued)

API	Description
<code>ix_cc_v6v4_async_get_end_tunnel()</code>	Sends a message of type <code>IX_CC_V6V4_MSG_GET_END_TUNNEL</code> to the Tunneling core component.
<code>ix_cc_v6v4_async_add_allowed_source()</code>	Sends a message of type <code>IX_CC_V6V4_MSG_ADD_ALLOWED_SOURCE</code> to the Tunneling core component.
<code>ix_cc_v6v4_async_delete_allowed_source()</code>	Sends a message of type <code>IX_CC_V6V4_MSG_DELETE_ALLOWED_SOURCE</code> to the Tunneling core component.
<code>ix_cc_v6v4_async_get_allowed_sources()</code>	Sends a message of type <code>IX_CC_V6V4_MSG_GET_ALLOWED_SOURCES</code> to the Tunneling core component.
<code>ix_cc_v6v4_async_dump_end_tunnels()</code>	Sends a message of type <code>IX_CC_V6V4_MSG_DUMP_END_TUNNELS</code> to the Tunneling core component.
<code>ix_cc_v6v4_async_clear_allowed_sources()</code>	Sends a message of type <code>IX_CC_V6V4_MSG_CLEAR_ALLOWED_SOURCES</code> to the Tunneling core component.
<code>ix_cc_v6v4_async_add_start_tunnel()</code>	Sends a message of type <code>IX_CC_V6V4_MSG_ADD_START_TUNNEL</code> to the Tunneling core component.
<code>ix_cc_v6v4_async_delete_start_tunnel()</code>	Sends a message of type <code>IX_CC_V6V4_MSG_DELETE_START_TUNNEL</code> to the Tunneling core component.
<code>ix_cc_v6v4_async_set_ttl()</code>	Sends a message of type <code>IX_CC_V6V4_MSG_SET_TTL</code> to the Tunneling core component.
<code>ix_cc_v6v4_async_set_encap_tos_option()</code>	Sends a message of type <code>IX_CC_V6V4_MSG_SET_ENCAP_TOS_OPTION</code> to the Tunneling core component.
<code>ix_cc_v6v4_async_set_tos()</code>	Sends a message of type <code>IX_CC_V6V4_MSG_SET_TOS</code> to the Tunneling core component.
<code>ix_cc_v6v4_async_set_mtu()</code>	Sends a message of type <code>IX_CC_V6V4_MSG_SET_MTU</code> to the Tunneling core component.
<code>ix_cc_v6v4_async_set_subnet_broadcast()</code>	Sends a message of type <code>IX_CC_V6V4_MSG_SET_SUBNET_BROADCAST</code> to the Tunneling core component.
<code>ix_cc_v6v4_async_get_start_tunnel()</code>	Sends a message of type <code>IX_CC_V6V4_MSG_GET_START_TUNNEL</code> to the Tunneling core component.
<code>ix_cc_v6v4_async_dump_start_tunnels()</code>	Sends a message of type <code>IX_CC_V6V4_MSG_DUMP_START_TUNNELS</code> to the Tunneling core component.
<code>ix_cc_v6v4_async_get_statistics()</code>	Sends a message of type <code>IX_CC_V6V4_MSG_GET_STATISTICS</code> to the Tunneling core component.

54.8 Library API

Table 54-4 lists the IPv6 to IPv4 Tunneling library API.

Table 54-4. IPv6 to IPv4 Tunneling Library API

API	Description
<code>ix_cc_v6v4_add_end_tunnel()</code>	Adds an end tunnel endpoint
<code>ix_cc_v6v4_delete_end_tunnel()</code>	Deletes an end tunnel endpoint.
<code>ix_cc_v6v4_set_decap_tos_option()</code>	Specifies the manner in which the TOS byte is handled during decapsulation
<code>ix_cc_v6v4_set_src_validation()</code>	Specifies whether ingress source validation is performed during decapsulation
<code>ix_cc_v6v4_get_end_tunnel()</code>	Retrieves information about a specific end tunnel endpoint
<code>ix_cc_v6v4_add_allowed_source()</code>	Adds an entry to the ingress source validation list for an end tunnel endpoint
<code>ix_cc_v6v4_delete_allowed_source()</code>	Deletes an entry from the ingress source validation list for an end tunnel endpoint
<code>ix_cc_v6v4_clear_allowed_sources()</code>	Deletes all entries from the ingress source validation list
<code>ix_cc_v6v4_get_allowed_sources()</code>	Retrieves the list of allowed source addresses and associated prefix lengths associated with an end tunnel endpoint
<code>ix_cc_v6v4_dump_end_tunnels()</code>	Retrieves the list of all currently configured end tunnel endpoints
<code>ix_cc_v6v4_add_start_tunnel()</code>	Adds a start tunnel endpoint
<code>ix_cc_v6v4_delete_start_tunnel()</code>	Deletes a start tunnel endpoint
<code>ix_cc_v6v4_set_ttl()</code>	Sets the TTL value for a start tunnel endpoint
<code>ix_cc_v6v4_set_encap_tos_option()</code>	Specifies how the TOS byte is handled during the encapsulation process
<code>ix_cc_v6v4_set_tos()</code>	Sets the TOS value for the tunnel
<code>ix_cc_v6v4_set_mtu()</code>	Sets the path MTU for a start tunnel endpoint
<code>ix_cc_v6v4_set_subnet_broadcast()</code>	Sets the subnet broadcast address for a start tunnel endpoint
<code>ix_cc_v6v4_get_start_tunnel()</code>	Retrieves information about a specific start tunnel endpoint
<code>ix_cc_v6v4_dump_start_tunnels()</code>	Retrieves a list of all currently configured start tunnel endpoints
<code>ix_cc_v6v4_get_statistics()</code>	Retrieves tunneling packet counters
<code>ix_cc_v6v4_set_property()</code>	Updates the dynamic property

NAT-PT Translation Core Components**55**

The translation core component assists the translation microblock to implement the IPv6 to IPv4 (and vice-versa) address and protocol translation as defined in RFC2766 Network Address Translation-Port Translation specification (NAT-PT).

A single core component supports the translation microblock. The following functionality is provided:

- Configuration of the translation microblock.
- Setup and management of translation information.
- Packet handlers to receive packets from the translation microblock and from other core components.
- Message handlers to allow configuration of translation information.
- Translation of ICMP and ICMPv6 packets.
- Translation of FTP Control packets (FTP ALG).
- Translation of DNS packets (DNS ALG).
- Fragmentation and Reassembly of translated packets.

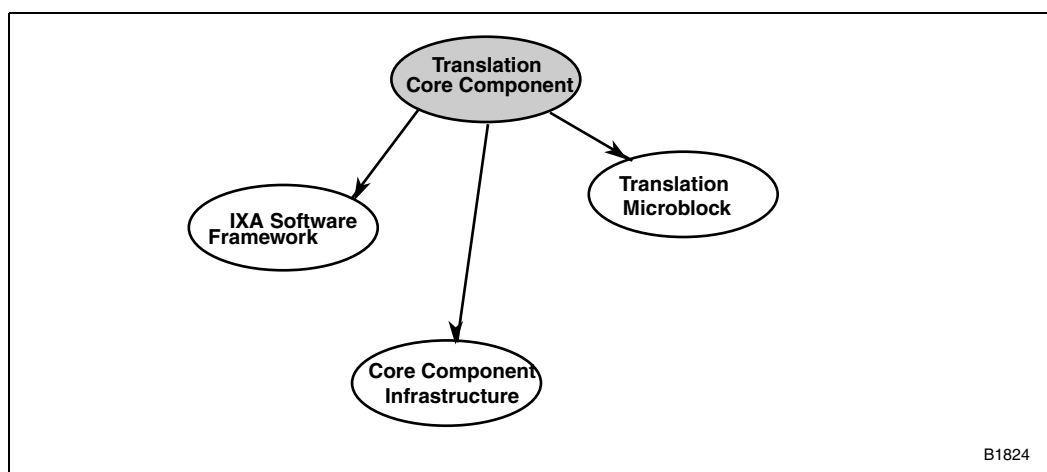
The translation core component can receive packets from the translation microblock only. Messages can be received from other core components and from other system components responsible for configuring the forwarding plane.

For complete details on the NAT-PT Microblock, see [Chapter 27, “IPv6 To IPv4 Translation Microblock”](#) and for external APIs see [Chapter 18, “NAT-PT Translation”](#) of the *IXA Software Building Blocks Reference Manual*.

55.1 Dependencies

The translation core component uses the services of the IXA software framework for allocating memory, freeing memory, patching symbols, accessing shared resources such as packet data, and retrieving static parameters. The Core Component Infrastructure (CCI) is used for event, message and packet handling, and for sending messages and packets to the IPv4 and IPv6 core components.

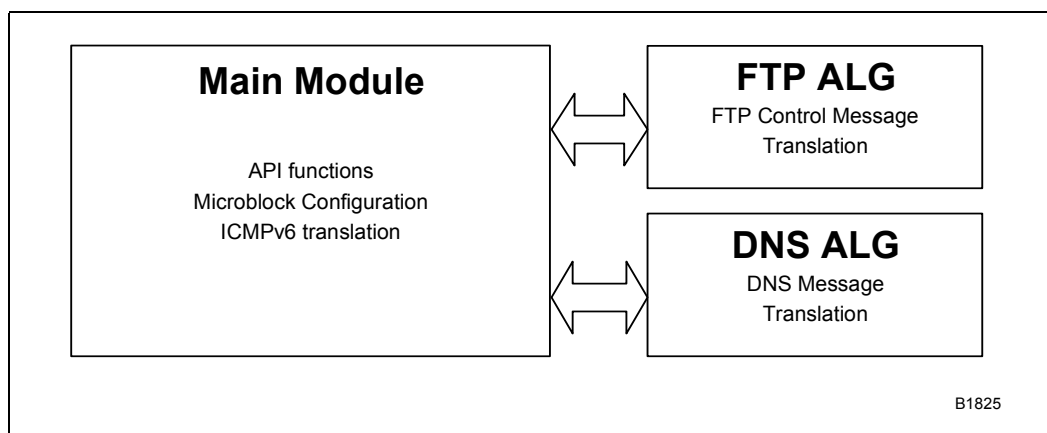
Figure 55-1. Translation Core Component Dependencies



55.2 High-Level Architecture

The following figure shows a high-level block diagram of the Translation Core Component. The main module provides most of the NAT-PT functionality. The Application Level Gateways such as FTP ALG and DNS ALG are implemented as separate modules so that only required ALGs are activated and more ALGs can be added easily.

Figure 55-2. Translation Core Component Architecture



55.2.1 Main Module

The main module is responsible for the following tasks:

- Exposes the API to other modules for configuration, translation and information.
- Creates and maintains the IPv6/IPv4 mapping tables that are used by the translation microblock.
- Accepts and processes packets given by the translation microblock.

- Keeps track of active sessions and, with the help of time-outs, terminates them when there is no activity.
- Generates and translates ICMP messages.

55.2.2 DNS Application Level Gateway

The DNS ALG module translates the contents of a DNS packet that is going from the IPv4 realm to the IPv6 realm or vice versa. DNS packets are recognized by the microblock and sent to the core component. The following sections show the important steps in translation.

55.2.2.1 DNS Name to Address Query from IPv4 Realm

- Query type “A” is changed to query type “AAAA” in the requests to IPv6 DNS Server. IPv4 packet header is translated to IPv6 packet header.
- Translate “AAAA” response from IPv6 DNS Server to “A” response. Replace the v6 address given by IPv6 DNS Server to the v4 address. If there is no v4 address to v6 address mapping, a new mapping is created. IPv6 packet header is translated to IPv4 packet header.

55.2.2.2 DNS Name to Address Query from IPv6 Realm

- Forward the “AAAA” query as it was received. Send an additional query for type “A” for the same node. IPv6 packet header is translated to IPv4 packet header.
- If there is an “A” record in the DNS Response from the IPv4 realm, translate the “A” record to “AAAA” record and add “IPv4-mapped IPv6 address” prefix to the IPv4 address. IPv4 packet header is translated to IPv6 packet header.

55.2.3 FTP Application Level Gateway

FTP ALG module translates the contents of an FTP control packet that is going from the IPv4 realm to the IPv6 realm or vice versa. FTP control packets are recognized by the microblock and sent to the core component. The following sections show the important steps in translation.

55.2.3.1 IPv4 FTP Client communicates with IPv6 FTP Server

IPv4 client may send PORT or PASV commands to IPv6 server. These commands are converted to EPRT and EPSV commands by the FTP ALG. Also, the response to EPSV command that originates from IPv6 FTP server is translated and sent to IPv4 client.

55.2.3.2 IPv6 FTP Client communicates with IPv4 FTP Server

IPv6 FTP client may send EPRT or EPSV commands to IPv4 FTP server. These commands are converted to PORT and PASV commands by the FTP ALG. Also, the response to PASV command that originates from IPv4 FTP server is translated and sent to IPv6 client. FTP ALG does not translate “EPSV all” command from IPv6 client, hence, an error message is sent back to IPv6.

55.2.3.3 Packet Header update

When the payload of FTP control packet changes as described in the previous two sections, the header needs to be updated for the new size of the payload and a new TCP checksum must be calculated. TCP sequence and acknowledgement numbers must also be changed in the TCP header.

Implementation Note: DNS ALG and FTP ALG components are not implemented in the current release.

55.2.4 Specific Design Details

55.2.5 NAT-PT Operation

The core component is provided with a pool of IPv4 addresses. When a new TCP or UDP session starts, an IPv4 address from the pool is mapped to the IPv6 address of the host in the IPv6 realm. The mapping tables are updated with the mapping, so that future packets are processed by the microblock. The number of simultaneous sessions supported in this NAT-PT mode is equal to the number of IPv4 address in the pool.

The core component's API has functions to provide and update the pool of IPv4 addresses.

55.2.6 NAPT-PT Operation

In NAPT-PT, the TCP/UDP port number is also translated and hence, with just one IPv4 address, it is possible to support more than 60,000 TCP and 60,000 UDP sessions.

Though more than one IPv4 address could be used for NAPT-PT, the current design of the core component uses only one IPv4 address for NAPT-PT for the following reasons:

- It simplifies configuration and implementation.
- Using multiple IPv4 addresses does not provide any significant benefit.

The core component's API has functions to enable the NAPT-PT mode of operation.

Implementation Note: NAPT-PT Operation is not implemented in the current release.

55.2.7 Static Mapping of Addresses

The core component creates the mappings of IP addresses dynamically, i.e., a mapping is created when a new session starts. However, static mapping of IPv4 addresses to some IPv6 hosts is necessary. For example, the IPv6 DNS server needs an IPv4 address that is given out to the IPv4 realm. There may be other services that require static mapping. Hence it is possible to add multiple static mappings. The IPv4 addresses used for the static mappings are not used for dynamic mapping.

The core component's API has functions to add and delete static mapping of IP addresses.

55.2.8 Static Port Mapping

In the NAT-PT mode of operation, since the whole IPv6 realm appears to be a single IPv4 host, it is not possible to have multiple servers that listen on same TCP/UDP port that could be accessed by IPv4 realm. For example, it is not possible to have two IPv6 web servers that are listening on TCP port 80 communicate simultaneously to IPv4 realm through the NAT-PT device. In order to designate the server in the IPv6 realm for communication with IPv4 realm, a static mapping of the IPv6 address of IPv6 server, the protocol/port number for the service and the IPv4 address used for NAT-PT must be created.

Other services such as SMTP (email) and FTP servers also may require such static mapping with the protocol/port number for those services.

The core component's API has functions to add and delete static mapping of IPv6 address and protocol/port number.

55.3 Configuration and Initialization

55.3.1 Configuration Parameters

The translation core component uses the system registry to store its static configuration parameters. A description of each supported parameter follows.

55.3.1.1 NAT-PT Table Size

This parameter specifies the number of entries in the NAT-PT mapping table, which is basically the maximum number of NAT-PT sessions supported. The current value is 256.

55.3.1.2 NAT-PT Hash Table Size

This parameter specifies the number of entries in the Hash table that is used for indexing into the NAT-PT table. The current value is 128.

55.3.1.3 NAT-PT Hash Collision Table Size

This parameter specifies the number of entries in the Hash Collision table that is used for indexing into the NAT-PT table. The current value is 128.

55.3.1.4 NAT-PT Prefix for IPv6 domain

This parameter specifies the NAT-PT prefix that is assigned to the IPv6 domain so the NAT-PT microblock can recognize the IPv6 packets that are to be translated. The current value is IPv6-Mapped-IPv4 address prefix (0:0:0:0:ffff:96).

55.3.2 Initialization

The translation core component initializes the memory that is shared with the translation microblock. This includes a number of tables that keep the mapping between IPv4 and IPv6 addresses and packet counters, if used. The core component patches symbols needed by the translation microblock at initialization.

A description of each patched symbol is given below.

55.3.2.1 NAT-PT Table Base

The core component allocates shared memory for the NAT-PT table and patches the starting memory address to the microblock.

55.3.2.2 NAT-PT V6V4 Hash Table Base

The core component allocates shared memory for the V6 to V4 Hash table and patches the starting memory address to the microblock.

55.3.2.3 NAT-PT V4V6 Hash Table Base

The core component allocates shared memory for the V4 to V6 table and patches the starting memory address to the microblock.

55.3.2.4 NAT-PT Hash Collision Table Base

The core component allocates shared memory for the collision hash table and patches the starting memory address to the microblock.

55.3.2.5 Blade ID

The core component patches the Blade Identification number to the microblock.

55.3.2.6 48-bit Hash Multiplier

This 48-bit hash multiplier is used as the seed to initialize the 48-bit hardware hash component. The 48-bit hash component is used to hash on IPv4 addresses. This is shared with the microblock.

55.3.2.7 128-bit Hash Multiplier

This 128-bit hash multiplier is used as the seed to initialize the 128-bit hardware hash component. The 128-bit hash component is used to hash on IPv6 addresses. This is shared with the microblock.

55.3.2.8 NAT-PT Prefix for IPv6 domain

The NAT-PT Prefix for the IPv6 domain is patched to the microblock

55.4 External API

This section summarizes the NAT-PT Translation core component external API. For complete details, see [Chapter 18, “NAT-PT Translation”](#) in the *IXA Software Building Blocks Reference Manual*.

55.4.1 Data Structures, Types and Macros

Chapter 55, “Data Structures, Types and Macros in Translation Core Components” lists the data structures, types and macros in translation core components.

Table 55-1. Data Structures, Types and Macros in Translation Core Components

Data Structures, Types and Macros	Description
<code>ix_cc_natpt_config_params</code>	Contains the configuration parameters
<code>ix_cc_natpt_v6addr</code>	Specifies an IPv6 address
<code>ix_cc_natpt_v4addr</code>	Specifies an IPv4 address
<code>ix_cc_natpt_static_v6v4map</code>	Specifies a static IPv6 address to IPv4 address mapping
<code>ix_cc_natpt_v4v6portmap</code>	Specifies a static mapping of IPv6 address, IPv4 address and TCP/UDP Port number
<code>ix_cc_natpt_naptmode</code>	Turns ON/OFF NAPT-PT mode of operation
<code>ix_cc_natpt_session</code>	Contains information on a NAT-PT session

55.4.2 Core Component Infrastructure API

Table 55-2 lists the translation Core Component Infrastructure API.

Table 55-2. Translation Core Component Infrastructure API

API	Description
<code>ix_cc_natpt_init()</code>	Initializes the translation core component
<code>ix_cc_natpt_fini()</code>	Terminate the services of the translation core component
<code>ix_cc_natpt_msg_handler()</code>	Message handler routine for the translation core component
<code>ix_cc_natpt_microblock_pkt_handler()</code>	Packet handler routine for packets received from the translation microblock.

55.4.3 Messaging API

The message helper API is used by the calling application to construct and send messages asynchronously to the translation core component. [Table 55-3](#) lists the message helper API in the translation core component.

Table 55-3. Message Helper API in the Translation Core Component

Message Helper API	Description
<code>ix_cc_natpt_async_set_config_parameters()</code>	Sends a message of type <code>IX_CC_NATPT_SET_CONFIG_PARAM</code> to the translation core component.
<code>ix_cc_natpt_async_get_config_parameters()</code>	Sends a message of type <code>IX_CC_NATPT_GET_CONFIG_PARAM</code> to the translation core component.
<code>ix_cc_natpt_async_add_static_mapping()</code>	Sends a message of type <code>IX_CC_NATPT_ADD_STATIC_MAP</code> to the translation core component.
<code>ix_cc_natpt_async_remove_static_mapping()</code>	Sends a message of type <code>IX_CC_NATPT_REM_STATIC_MAP</code> to the translation core component.
<code>ix_cc_natpt_async_get_static_mapping()</code>	Sends a message of type <code>IX_CC_NATPT_GET_STATIC_MAP</code> to the translation core component.
<code>ix_cc_natpt_async_add_v4addr_pool()</code>	Sends a message of type <code>IX_CC_NATPT_ADD_V4ADDR_POOL</code> to the translation core component.
<code>ix_cc_natpt_async_remove_v4addr_pool()</code>	Sends a message of type <code>IX_CC_NATPT_REM_V4ADDR_POOL</code> to the translation core component.
<code>ix_cc_natpt_async_get_v4addr_pool()</code>	Sends a message of type <code>IX_CC_NATPT_GET_V4ADDR_POOL</code> to the translation core component.
<code>ix_cc_natpt_async_set_natpt_mode()</code>	Sends a message of type <code>IX_CC_NATPT_SET_NAPT_MODE</code> to the translation core component.
<code>ix_cc_natpt_async_add_v4v6port_mapping()</code>	Sends a message of type <code>IX_CC_NATPT_ADD_V4V6_PORT_MAP</code> to the translation core component.
<code>ix_cc_natpt_async_remove_v4v6port_mapping()</code>	Sends a message of type <code>IX_CC_NATPT_REM_V4V6_PORT_MAP</code> to the translation core component.
<code>ix_cc_natpt_async_get_v4v6port_mapping()</code>	Sends a message of type <code>IX_CC_NATPT_GET_V6V4_PORT_MAP</code> to the translation core component.
<code>ix_cc_natpt_async_get_active_sessions()</code>	Sends a message of type <code>IX_CC_NATPT_GET_ACTIVE_SESSIONS</code> to the translation core component.

55.4.4 Library API

The library API consists of synchronous operations for the translation core component and is used by the asynchronous message support mechanism. [Table 55-4](#) lists the library API in the translation core component.

Table 55-4. Library API in the Translation Core Component

Message Helper API	Description
<code>ix_cc_natpt_set_config_parameters()</code>	Sets global translation core component configuration parameters
<code>ix_cc_natpt_get_config_parameters()</code>	Retrieves current global configuration parameters
<code>ix_cc_natpt_add_static_mapping()</code>	Adds a static IPv6 address to IPv4 address mapping
<code>ix_cc_natpt_remove_static_mapping()</code>	Removes an existing static IPv6 address to IPv4 address mapping
<code>ix_cc_natpt_get_static_mapping()</code>	Retrieves existing static IPv6 address to IPv4 address mappings
<code>ix_cc_natpt_add_v4addr_pool()</code>	Adds one or more IPv4 addresses used to map IPv6 addresses dynamically
<code>ix_cc_natpt_remove_v4addr_pool()</code>	Removes one or more existing IPv4 addresses from the IPv4 address pool
<code>ix_cc_natpt_get_v4addr_pool()</code>	Retrieves existing IPv4 addresses in the IPv4 address pool
<code>ix_cc_natpt_set_natpt_mode()</code>	Turns ON/OFF NATPT mode of operation
<code>ix_cc_natpt_add_v4v6port_mapping()</code>	Adds an IPv4 address, IPv6 address and Protocol/Port Number static mapping
<code>ix_cc_natpt_remove_v4v6port_mapping()</code>	Removes an existing static mapping of IPv4 address, IPv6 address and Port Number
<code>ix_cc_natpt_get_v4v6port_mapping()</code>	Retrieves an existing static IPv6 address, IPv4 address and Port Number mappings
<code>ix_cc_natpt_get_active_sessions()</code>	Retrieves the list of currently active NATPT sessions

55.5 Modularity

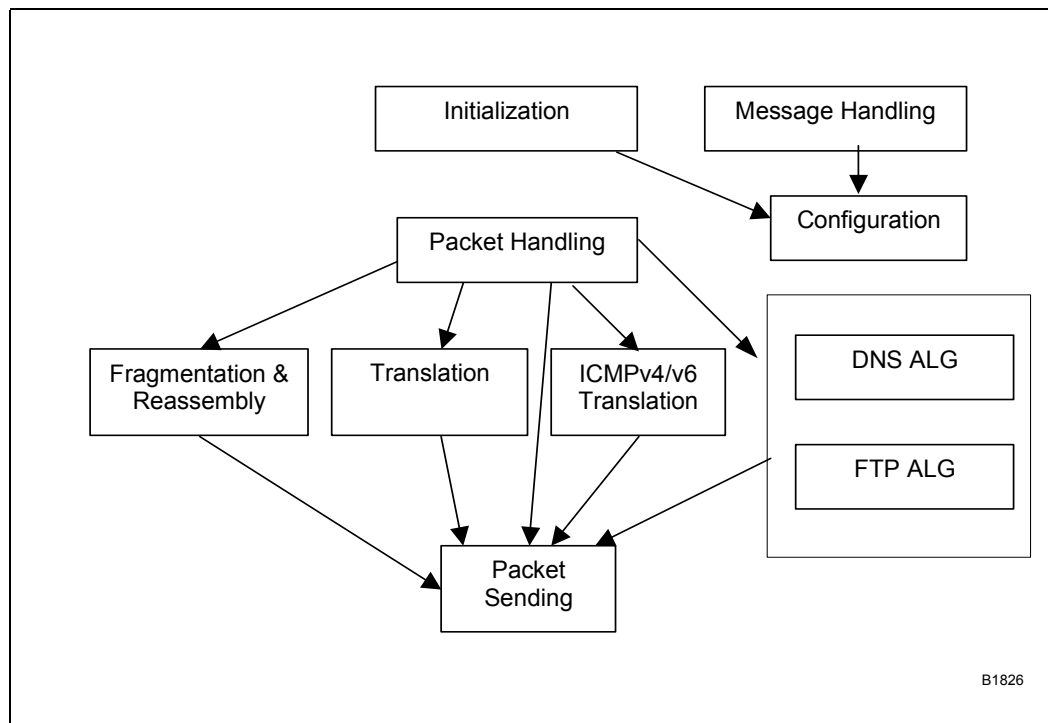
The translation core component is composed of the following modules:

- Initialization Module
- Message Handling Module
- Packet Handling Module
- Fragmentation & Reassembly Support Module
- ICMPv4/ICMPv6 Translation Module
- Translation Module
- DNS ALG Module
- FTP ALG Module
- Packet Sending Support Module

- Configuration Support Module

Figure 55-1 shows the interdependencies among these modules.

Figure 55-1. Modularity of Translation Core Component



55.5.1 Initialization Module

The initialization module is responsible for performing the initialization tasks.

55.5.2 Message Handling Module

The message handling module receives messages from the calling application and uses the services of the appropriate support module to process the message.

55.5.3 Packet Handling Module

The packet handling module receives packets from the translation microblock and calls the appropriate support modules to process the packet.

55.5.3.1 Microblock Packet Handler

This packet handler handles packets that were received from the translation microblock. These packets can be of the following types:

- ICMP packets that are to be forwarded from the v4 realm to v6 or vice-versa.
- Fragmented IPv4 packet that must be reassembled and translated to IPv6 packet.

- DNS packets that must be translated from IPv6 to IPv4 or vice-versa.
- FTP control packets that must be translated from IPv6 to IPv4 or vice-versa.
- Exception packets, i.e., packets for which the microblock could not find a mapping in the tables.

55.5.4 Fragmentation & Reassembly Support Module

This module handles the details involved in fragmenting a large packet and reassembling a fragmented packet.

When the IPv4 realm does not perform MTU discovery and sends a packet that is larger than the IPv6-side MTU, then that packet is fragmented before sending out to the IPv6 realm.

Reassembly is required in the case of a fragmented IPv4 UDP packet with zero checksum. When such a packet arrives, all the fragments of that packet must be collected in order to calculate the checksum over the entire packet.

55.5.5 ICMP Translation Module

All ICMP messages that must be forwarded from the IPv4 realm to the IPv6 realm or vice-versa are given to core component by the microblock. This module translates these ICMP packets and gives the packets to the appropriate forwarder core component.

55.5.6 Translation Module

The microblock gives the exception packets, i.e., the packets for which it could not find a mapping in the mapping tables, to the core component. This module then creates new mappings in the appropriate tables and translates the exception packets as well.

55.5.7 DNS ALG Module

This module translates the contents of a DNS packet that is going from the IPv4 realm to the IPv6 realm or vice versa. DNS packets are recognized by the microblock and sent to the core component.

55.5.8 FTP ALG Module

This module translates the contents of an FTP control packet that is going from the IPv4 realm to the IPv6 realm or vice versa. FTP control packets are recognized by the microblock and sent to the core component.

55.5.9 Packet Sending Support Module

This module provides support for calculating header checksum and passing the packet to the appropriate microblock to be sent out.

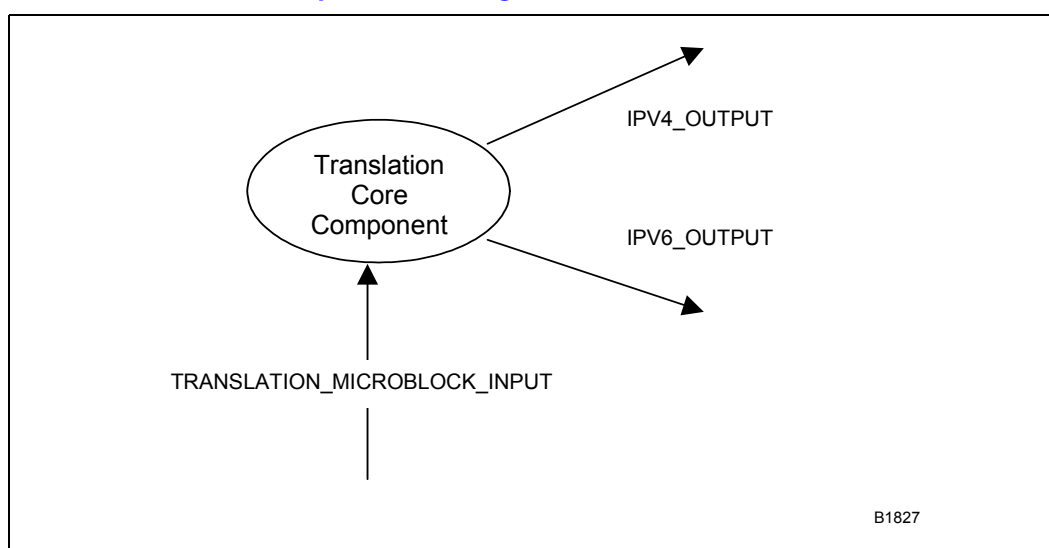
55.5.10 Configuration Support Module

This module maintains the configuration information in memory. This includes the static IPv6 and IPv4 mapping, pool of IPv4 addresses and static IPv6/IPv4/Port mappings.

55.6 Bindings

The translation core component has one packet input from the translation microblock. There are two outputs, one for translated IPv4 packets and one for translated IPv6 packets. [Figure 55-2](#) illustrates these bindings.

Figure 55-2. Translation Core Component Bindings



DiffServ Components

The DiffServ includes the following core components:

- [Chapter 56, “Six-Tuple Classifier Core Component”](#)
Provides multi-field range matching classification.
- [Chapter 57, “Three Color Meter Core Component”](#)
Three Color Marker based on RFC 2698.
- [Chapter 58, “Weighted Random Early Detection \(WRED\) Core Component”](#)
Weighted Random Early Detection microblock provides active queue congestion avoidance.
- [Chapter 59, “DSCP Classifier Core Component”](#)
Classifier based on 6-bit DSCP value in IP header. IPv4 and IPv6 packets are handled.

Six-Tuple Classifier Core Component 56

56.1 Overview

The 6-Tuple Exact Match Classifier Core Component provides the following functionalities:

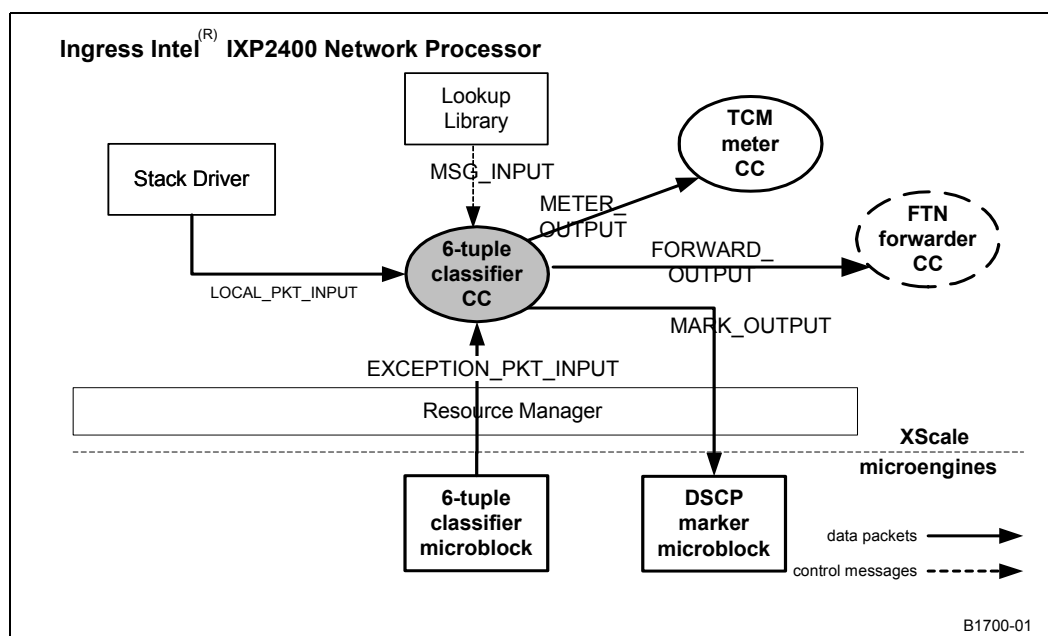
- Initializes and configures the 6-Tuple Classifier Microblock by patching symbols.
- Provides an API interface to add and remove exact-match rules. The rules are stored in a hash table shared between the core component and the microblock.
- Implements exact-matching capability. There is no range matching classification in a core component (slow path). Range matching is available with TCAM support.
- Implements the basic classification function of IP packets without header options in the same way as is done in the microblock. The core component classifies local IP packets generated by the Stack Driver.
- Implements an extended classification function for exception packets thrown by the 6-tuple microblock. The exception packets are the packets with IP header options.
- Marks DSCP value in exception packets and packets from local stack driver, if the QoS rule directs the packets to DSCP marking stage.

For complete details on the 6-Tuple Classifier Microblock, see [Chapter 30, “6-tuple Exact Match Classifier Microblock.”](#) and for external APIs see [Chapter 19, “Six-Tuple Exact Match Classifier”](#) of the *IXA Software Building Blocks Reference Manual*.

56.2 Data and Control Flow

Figure 56-1 illustrates data flow of 6-tuple classifier core component.

Figure 56-1. Data Flow of 6-tuple Classifier Core Component



56.2.1 Packet Inputs

The 6-tuple classifier CC defines two packet inputs, one per each data source. On one input, it receives exception packets from the 6-tuple classifier microblock. The second input is dedicated to local IP packets generated by the stack driver. Two separate inputs alleviate an overhead associated with mutual exclusion due to multiple sources on a single input. Both input identifiers are defined in a system file, `bindings.h` as follows:

```
#define IX_CC_CLASSIFIER_6T_EXEPTION_PKT_INPUT
#define IX_CC_CLASSIFIER_6T_LOCAL_PKT_INPUT
```

56.2.2 Packet Outputs

The core component defines four outputs: for packets requiring metering (for example, TCM meter), for packets marked with DSCP value, for packets matching a forwarding rule (for example, FTN forwarder), and for packets not matching any policy (for example, IPv4 forwarder or drop action). Output identifiers are defined in `bindings.h` as:

```
#define IX_CC_CLASSIFIER_6T_METER_OUTPUT
#define IX_CC_CLASSIFIER_6T_MARKER_OUTPUT
#define IX_CC_CLASSIFIER_6T_FORWARD_OUTPUT
#define IX_CC_CLASSIFIER_6T_DEFAULT_OUTPUT
```

A pure IP DiffServ application does not use the forwarding output. This output supports manually configured FECs in an MPLS traffic engineering scenario.

56.2.3 Message Inputs

The core component defines a single input for all configuration messages.

```
#define IX_CC_CLASSIFIER_6T_MSG_INPUT
```

The 6-tuple core component receives messages from a single logical source—the Lookup Library. However, the core component should not assume that the library serializes invocations from multiple upper-layer applications. Thus, the message input should be protected with mutual exclusion mechanism.

56.3 Assumptions, Dependencies and Risks

56.3.1 Assumptions

The following design assumptions are made:

- The control interface is compliant with exact-match Lookup Library API.
- The core component configures a default rule in microblock by a means of patched symbols.

56.3.2 Dependencies

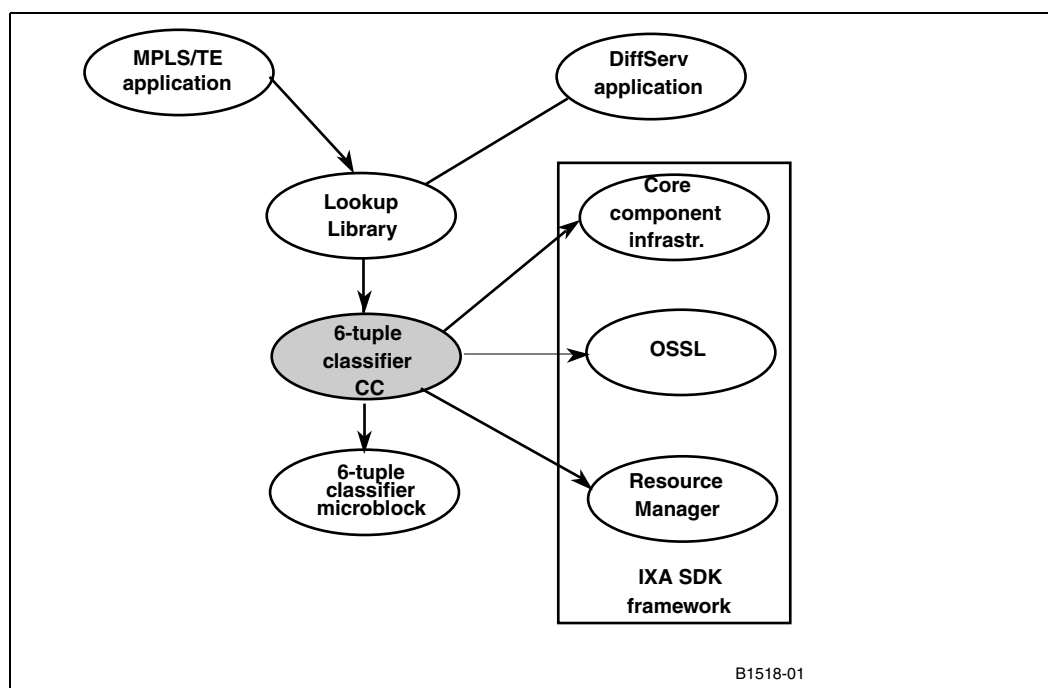
The 6-tuple classifier core component depends on the IXA SDK framework. It uses the services of Resource Manager for:

- Allocating and freeing DRAM/SRAM memory
- Patching symbols
- Accessing system registry to retrieve static parameters

The Core Components Infrastructure services are used for message and packet handling between other core components and 6-tuple microblock. The Operating System Service Layer (OSSL) makes the core component code independent of the underlying OS.

The core component configures a DRAM/SRAM hash table entry used by the 6-tuple classifier microblock. Execution of the microblock, however, does not impact the core component. [Figure 56-2](#) illustrates the 6-tuple Classifier core component dependencies.

Figure 56-2. 6-tuple Classifier Core Component Dependencies



The 6-tuple Classifier core component does not invoke itself recursively via the lookup library. In this way, it can stay independent of the lookup library design. Nevertheless, all applications should access the 6-tuple Classification table only through the Lookup Library API.

56.4 Configuration and Initialization

56.4.1 Static Configuration Data

The 6-tuple Classifier Core Component defines the following static configuration data listed in Table 56-1.

Table 56-1. Static Configuration Items in 6-tuple Classifier

Data (system property name)	Default	Description
CLASSIFIER_6T_HASH_ENTRY_DRAM_SIZE (\\CLASSIFIER_6T\\ HASH_ENTRY_DRAM_SIZE)CLASSIFIER_6T_HAS H_ENTRY_SRAM_SIZE (\\CLASSIFIER_6T\\HASH_ENTRY_SRAM_SIZE)	32	Number of bytes occupied by a single entry in a hash table. It is used to calculate the size of hash table.
CLASSIFIER_6T_HASH_KEY_MASK (\\CLASSIFIER_6T\\HASH_KEY_MASK)	16	Number of MSB bits from a hash key used as a hash table index. It is used to calculate the size of hash table and to patch imported variables.
CLASSIFIER_6T_STATS_ENTRY_SRAM_SIZE (\\CLASSIFIER_6T\\STATS_ENTRY_SRAM_SIZE)	16	Number of bytes occupied by a single entry in a statistics table. It is used to calculate the size of hash table.

Table 56-1. Static Configuration Items in 6-tuple Classifier (Continued)

Data (system property name)	Default	Description
CLASSIFIER_6T_DEFAULT_FLOW_ID (\\CLASSIFIER_6T\\DEFAULT_FLOW_ID)	0	Used to patch the imported variable in the micro-block.
CLASSIFIER_6T_DEFAULT_CLASS_ID (\\CLASSIFIER_6T\\DEFAULT_CLASS_ID)	8	Used to patch the imported variable in the micro-block.
CLASSIFIER_6T_DEFAULT_COLOR_ID (\\CLASSIFIER_6T\\DEFAULT_COLOR_ID)	0 (green)	Used to patch the imported variable in the micro-block.
CLASSIFIER_6T_DEFAULT_NEXTHOP_ID (\\CLASSIFIER_6T\\DEFAULT_NEXTHOP_ID)	0xFFFF (ID not known)	Used to patch the imported variable in the micro-block.
CLASSIFIER_6T_DEFAULT_NEXTHOP_ID_TYPE (\\CLASSIFIER_6T\\DEFAULT_NEXTHOP_ID_TYPE)	0 (IPv4 routing table)	Used to patch the imported variable in the micro-block.
CLASSIFIER_6T_DEFAULT_NEXT_BLOCK (\\CLASSIFIER_6T\\DEFAULT_NEXT_BLOCK)	CLASSIFIER_6T_NEXT4 (in system.h)	Used to patch the imported variable in the micro-block.
CLASSIFIER_6T_DEFAULT_STATS_ENABLE (\\CLASSIFIER_6T\\DEFAULT_STATS_ENABLE)	0	Indicates whether the classifier shall gather statistics for the default rule.
CLASSIFIER_6T_HASH_TABLE_SIZE (\\CLASSIFIER_6T\\HASH_TABLE_SIZE)	4096	The number of entries in the SRAM/DRAM hash table.
CLASSIFIER_6T_HASH_BY_CRC (\\CLASSIFIER_6T\\HASH_BY_CRC)	0	A flag indicating whether the microblock uses hash unit or CRC to calculate hash values.
CLASSIFIER_6T_HASH_MUL_LW0 (\\CLASSIFIER_6T\\HASH_MUL_LW0)	0xAAAAAAAA	Least significant 32 bits of 128-bit hash Multiplier.
CLASSIFIER_6T_HASH_MUL_LW1 (\\CLASSIFIER_6T\\HASH_MUL_LW1)	0xAAAAAAAA	Bits 32 to 63 of the 128-bit hash multiplier.
CLASSIFIER_6T_HASH_MUL_LW2 (\\CLASSIFIER_6T\\HASH_MUL_LW2)	0xAAAAAAAA	Bits 64 to 95 of the 128-bit hash multiplier.
CLASSIFIER_6T_HASH_MUL_LW3 (\\CLASSIFIER_6T\\HASH_MUL_LW3)	0xAAAAAAAA	Most significant 32 bits of 128-bit hash multiplier.
CLASSIFIER_6T_HASH_TABLE_IN_SRAM (\\CLASSIFIER_6T\\HASH_TABLE_IN_SRAM)	1	A flag indicating whether the microblock uses hash table in SRAM or DRAM.
CLASSIFIER_6T_PACKET_COUNTER_FEATURE (\\CLASSIFIER_6T\\PACKET_COUNTER_FEATURE)	0	A flag indicating whether the microblock supports statistics.
CLASSIFIER_6T_AUXILIARY_TABLE_SIZE (\\CLASSIFIER_6T\\AUXILIARY_TABLE_SIZE)	32 k	The size of the table storing collision chains in number of entries.

If the USE_HASH_UNIT is equal to 1, the core component configures a hardware hash unit with 128-bit multiplier on initialization.

The core component uses the CLASSIFIER_6T_PACKET_COUNTER_FEATURE flag to decide whether to allocate the statistics table in SRAM and patch CLASSIFIER_6T_DEFAULT_STATS_ADDRESS and CLASSIFIER_6T_STATS_SRAM_BASE symbols.

The static configuration data is obtained from the system repository during the component initialization time. If the system repository is not present, then the default values defined in the component's global header file, ix_cc_classifier_6t_defs.h, is used. The hash entry size and hash key mask values are configuration parameters patched into the microblock.

56.4.2 Dynamic Configuration Data

The dynamic configuration is pertinent to adding/removing entries from the hash table.

56.4.3 Patching Symbols

All patching symbols are enumerated in [Table 30.3.2.3](#). The core component patches all symbols during initialization.

56.4.4 Initialization and Shutdown Data Flow

[Figure 56-3](#) and [Figure 56-4](#) show a sequence of calls upon system initialization and shutdown.

Figure 56-3. Initialization of a 6-tuple Classifier Core Component

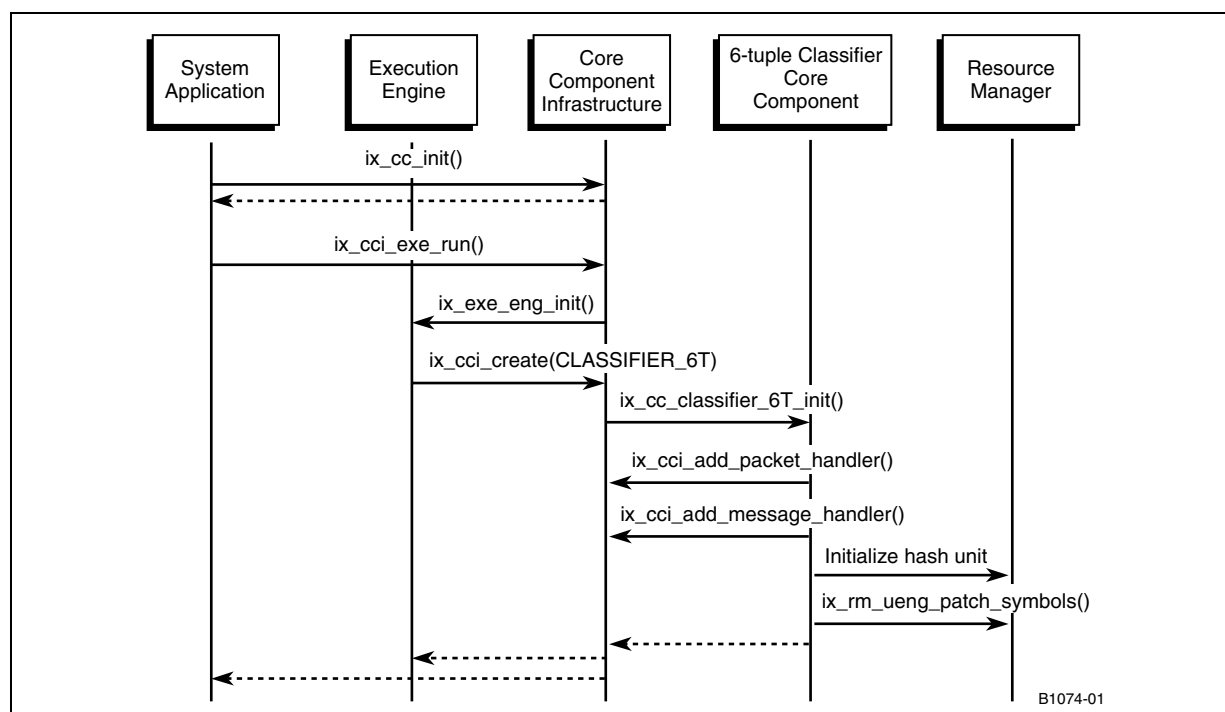
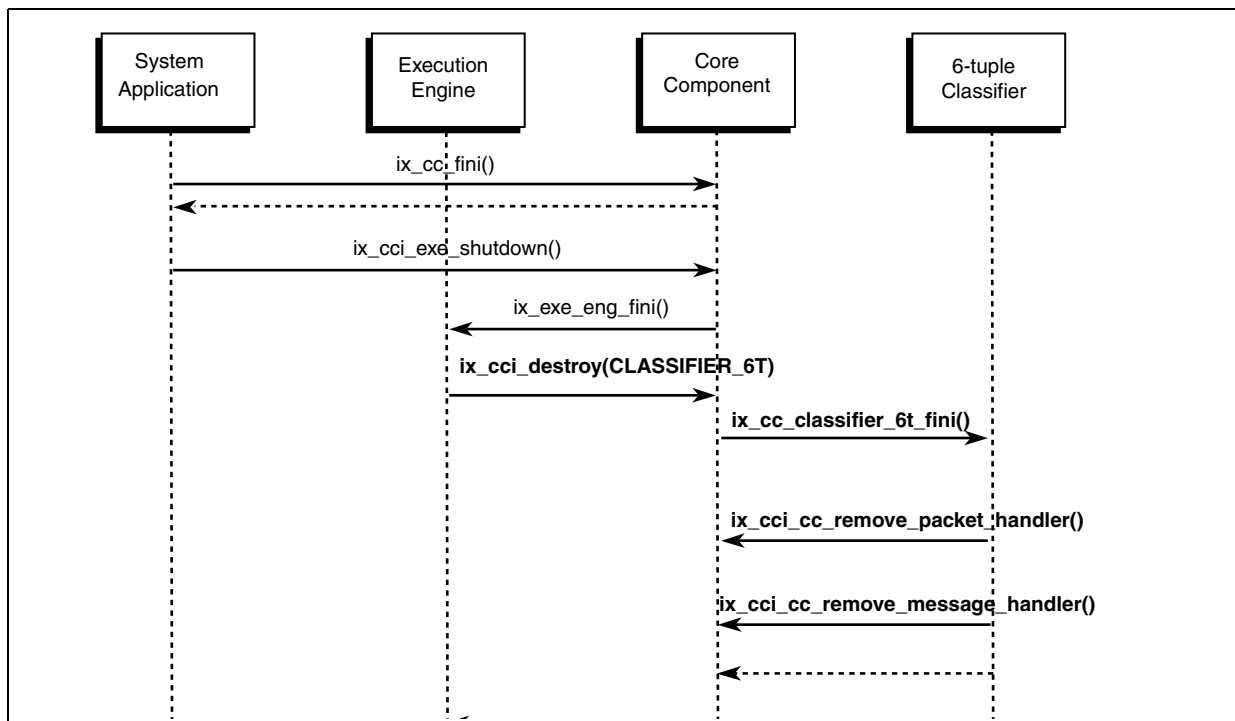


Figure 56-4. Shutdown of a 6-tuple Classifier Core Component



56.5 External API

This section lists the Six-Tuple Exact Match Classifier core components. For complete details, see [Chapter 19, “Six-Tuple Exact Match Classifier”](#) of the *IXA Software Building Blocks Reference Manual*.

56.5.1 Data Structures

The data structures are used in calls to functional APIs—Message Helper API and Library API. [Table 56-2](#) lists the three data structures for configuring exact-match rules defined by the classifier core component.

Table 56-2. Data Structures for Configuring Exact-match Rules

Data Structure	Description
ix_s_cc_classifier_6t_pattern	A bit string defining the exact-match classification rule.
ix_s_cc_classifier_6t_qos_result	A QoS classification result associated with the specified rule.
ix_s_cc_classifier_6t_fwd_result	A forwarding classification result associated with the specified rule.

56.5.2 Core Component Infrastructure API

Table 56-3 lists the Core Component Infrastructure API supported by the 6-tuple Classifier core component.

Table 56-3. 6-Tuple Classifier Core Component Infrastructure API

API Function	Description
<code>ix_cc_classifier_6t_init()</code>	Initializes the core component
<code>ix_cc_classifier_6t_fini()</code>	Terminates the core component
<code>ix_cc_classifier_6t_pkt_handler()</code>	Message handler for processing rule add/remove requests
<code>ix_cc_classifier_6t_msg_handler()</code>	Packet handler for processing exception packets

56.5.3 Message Helper API

Table 56-4 lists the Message Helper API functions.

Table 56-4. 6-Tuple Classifier Core Component Message Helper API

API Function	Description
<code>ix_cc_classifier_6t_async_add_entry()</code>	Adds a new classification pattern and the associated lookup results to the 6-tuple hash table
<code>ix_cc_classifier_6t_async_remove_entry()</code>	Removes a classification pattern and lookup result from the 6-tuple hash table
<code>ix_cc_classifier_6t_async_update_entry()</code>	Updates selected fields of the lookup result associated with a specified classification pattern.
<code>ix_cc_classifier_6t_async_search_entry()</code>	Returns a lookup result associated with a specified classification pattern.

56.5.4 Library API

Table 56-5 lists the functions provided by the Library API.

Table 56-5. 6-Tuple Classifier Core Component Library API

API Function	Description
<code>ix_cc_classifier_6t_add_entry()</code>	Adds a new classification pattern and the associated lookup results to the 6-tuple hash table
<code>ix_cc_classifier_6t_remove_entry()</code>	Removes a classification pattern and lookup result from the 6-tuple hash table
<code>ix_cc_classifier_6t_update_entry()</code>	Updates selected fields of the lookup result associated with a specified classification pattern.
<code>ix_cc_classifier_6t_search_entry()</code>	Returns a lookup result associated with a specified classification pattern.

56.6 Modularity

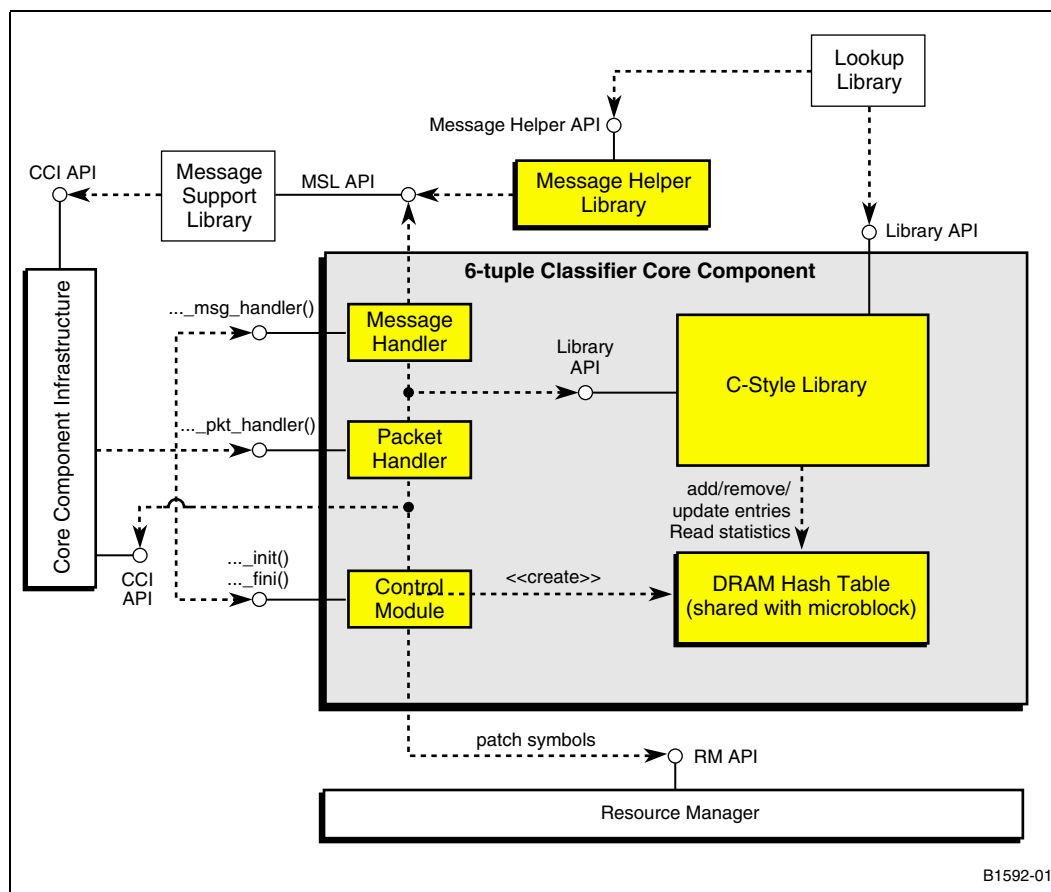
Table 56-6 lists the modules which comprise the 6-tuple Classifier Core Component.

Table 56-6. 6-Tuple Classifier Core Component Modules

Module	Description
Control Module	This module controls initialization and shutdown process. It implements two Core Component Infrastructure API functions: <code>ix_cc_classifier_6t_init()</code> and <code>ix_cc_classifier_6t_fini()</code> .
Message Handler	This module implements Core Component Infrastructure API function <code>ix_cc_classifier_6t_msg_handler()</code> , registered as a single handler for all message types. Whenever a message arrives, the module parses a message payload and converts it to the corresponding Library API call. The module uses Message Support Library to return operation results in a user-defined callback routine.
Packet Handler	This module implements Core Component Infrastructure API function <code>ix_cc_classifier_6t_pkt_handler()</code> , registered as a single handler for all packet sources. Internally, the module uses <code>ix_cc_classifier_6t_search_entry()</code> function of Library API to classify a packet. The classification results are written into a packet metadata. Finally, the module sends a packet to a selected output, using Core Component Infrastructure API.
C-style Library	This module implements all Library API functions, as enumerated in Section 56.5.4 . The module manages hash table entries. If a default rule is configured, the module patches respective symbols in microcode.
SRAM/DRAM Hash Table	This is a data structure shared between core component and microblock. Only the core component modifies hash table entries, while both CC and microblock can read this table. The table layout is defined in Section 30.3.3, "Data Structures" on page 534 .
Message Helper Library	This module implements all Message Helper API functions, as enumerated in Section 56.5.4 . The library constructs message payloads and sends them to a core component with a help of Message Support Library.

Figure 56-5 illustrates the architecture of the 6-Tuple Classifier Core Component.

Figure 56-5. 6-Tuple Classifier Core Component Architecture



Three Color Meter Core Component 57

57.1 Overview

The Three Color Meter (TCM) Core Component provides the following functionalities:

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

For complete details on the TCM Microblock, see [Chapter 31, “Three Color Meter Microblock”](#) and for external APIs see [Chapter 20, “Three Color Meter”](#) of the *IXA Software Building Blocks Reference Manual*.

57.2 Assumptions, Dependencies and Risks

57.2.1 Assumptions

The following design assumption is made:

- The core component does not meter packet flows, because a metering algorithm contains a critical section. The core component only configures meter instances.

57.2.2 Dependencies

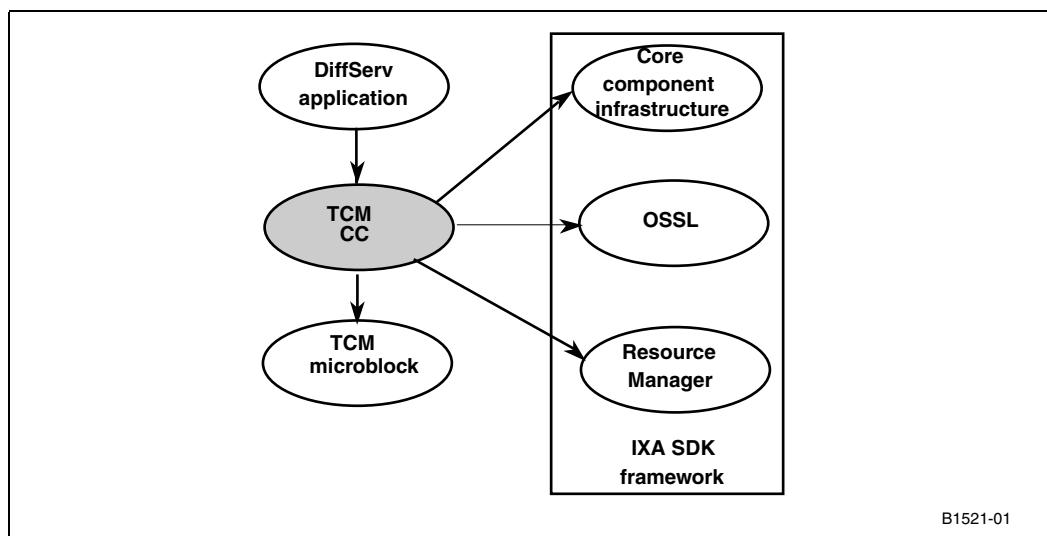
The TCM core component depends on the IXA SDK framework. It uses the services of Resource Manager for

- allocating and freeing SRAM memory
- patching symbols
- accessing system registry to retrieve static parameters

The Core Components Infrastructure services are used for message and packet handling and for sending packets to the TCM microblock. The Operating System Service Layer (OSSL) makes the core component code independent of the underlying OS.

The core component configures a SRAM meter table used by the microblock. Execution of the microblock does not impact the core component.

Figure 57-1. TCM Core Component Dependencies



57.3 Configuration and Initialization

57.3.1 Static Configuration Data

Table 57-1 lists the static configuration data defined by the TCM Core Component.

Table 57-1. Static Configuration Items in TCM

Data (system property name)	Default	Description
TCM_ENTRY_SRAM_SIZE (\\TCM\\ENTRY_SRAM_SIZE)	64	Number of bytes occupied by one entry in a meter table. Used to patch the imported variable in the micro-block and to calculate the amount of memory occupied by TCM meter table.
TCM_64BITSTAT_ENTRY_SRAM_SIZE (\\TCM\\64BITSTAT_ENTRY_SRAM_SIZE)	32	Number of bytes occupied by higher parts of 64-bit statistics counters associated with a meter entry. Used to patch the imported variable in the micro-block and to calculate the amount of memory occupied by TCM statistics table.
TCM_TABLE_SRAM_SIZE (\\TCM\\TABLE_SRAM_SIZE)	1024	Number of entries in a meter/statistics table. Used to calculate the amount of memory occupied by TCM meter and statistics tables.

These data are obtained from the system repository during the component initialization time. If the system repository is not present, then the default values defined in the component's global header file, `ix_cc_srtcm.h`, is used. The values are configuration parameters to be patched into the microblock.

57.3.2 Dynamic Configuration Data

The dynamic configuration is pertinent to adding/removing entries from the meter table.

57.3.3 Patching Symbols

The patching symbols correspond to setting an SRAM base address for meter and statistics tables. The patching symbols are defined in [Table 57-2](#). They have no default values. A core component patches these symbols only once, upon initialization.

Table 57-2. Variables Imported by this Block

Variable	Default	Description
TCM_TABLE_SRAM_BASE	-	Base address of the meter instance table maintained in SRAM.
TCM_64BIT_STAT_SRAM_BASE	-	Base address of the statistics table storing most significant parts of 64-bit long counters.

57.3.4 Initialization and Shutdown Data Flow

[Figure 57-2](#) and [Figure 57-3](#) show the sequence of calls upon system initialization and shutdown.

Figure 57-2. Initialization of TCM Core Component

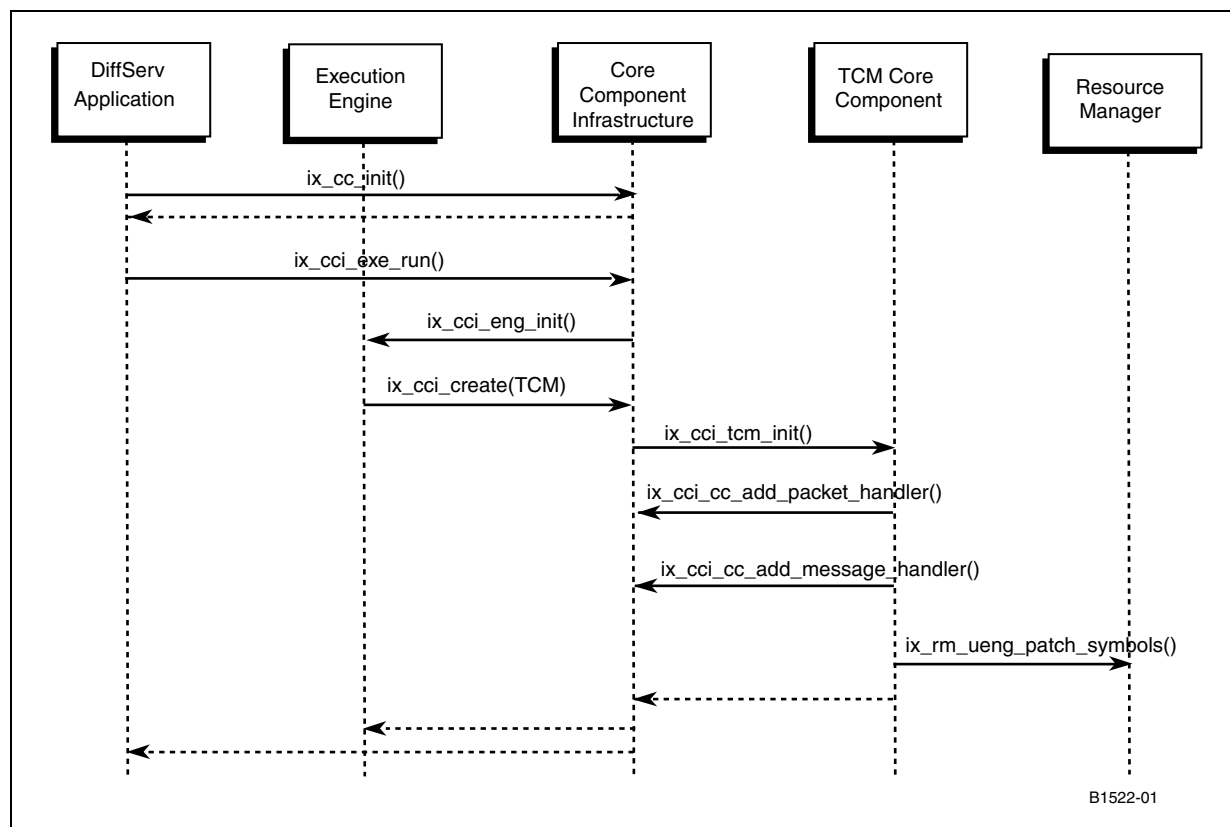
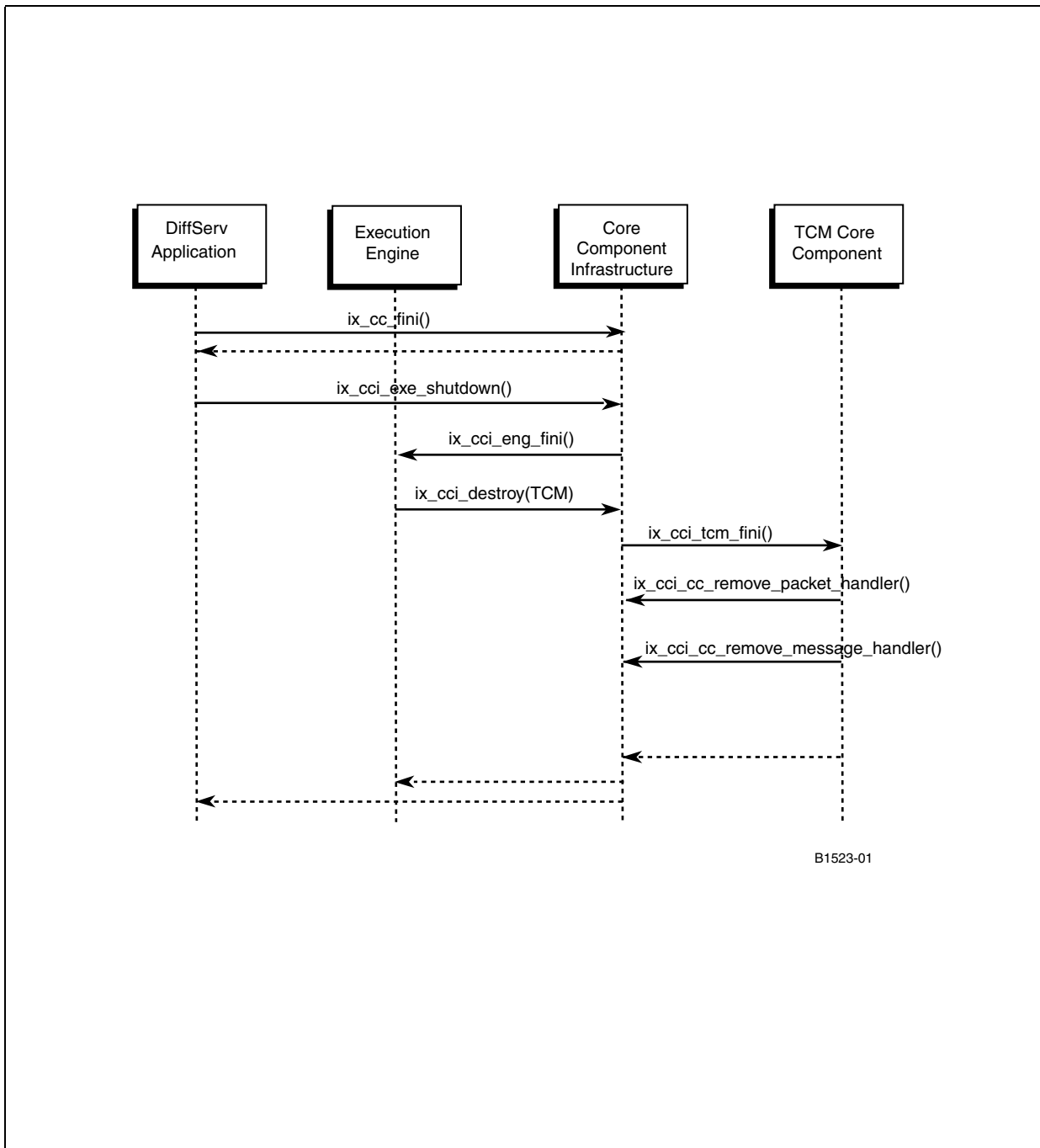


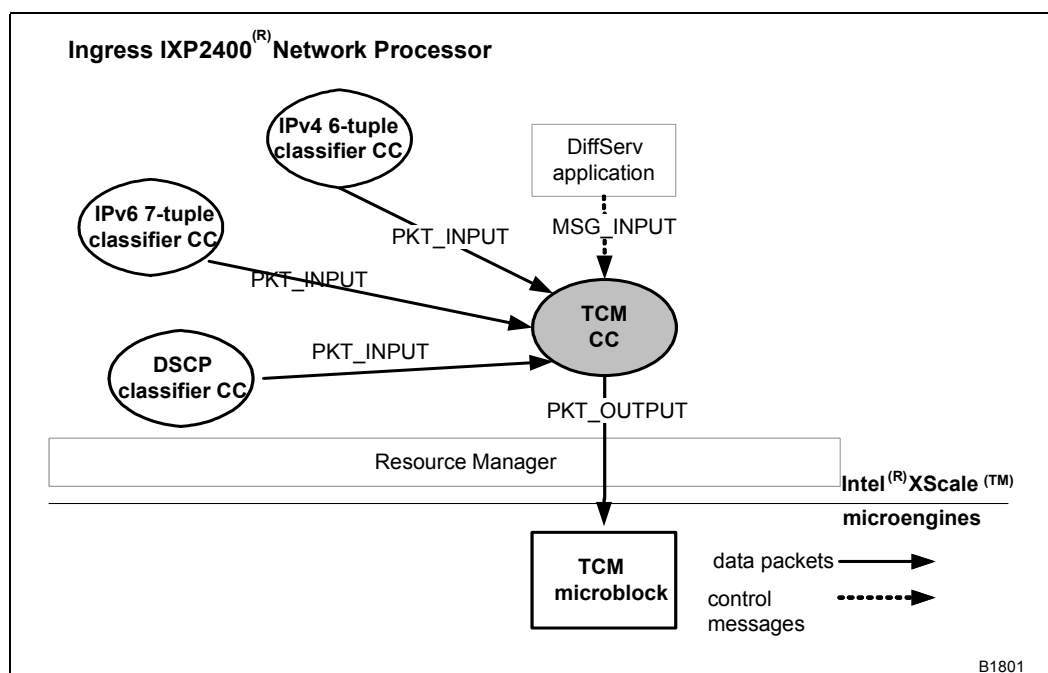
Figure 57-3. Shutdown of TCM Core Component



57.4 Data and Control Flow

Figure 57-4 illustrates the data flow of the TCM core components.

Figure 57-4. Data flow of the TCM Core Component



57.4.1 Packet Inputs

The TCM core component defines one input on which it receives packets from the 6-tuple classifier. This is a single-source input, defined in a system file, `bindings.h` as:

```
#define IX_CC_TCM_PKT_INPUT
```

57.4.2 Packet Outputs

The core component defines one packet output. All packets are moved to the TCM microblock. The output identifier is defined in `bindings.h` as:

```
#define IX_CC_TCM_PKT_OUTPUT
```

57.4.3 Message Inputs

The core component defines a one input for all configuration messages. This is a single-source input defined in `bindings.h` as:

```
#define IX_CC_TCM_MSG_INPUT
```


57.5 External API

This section lists the Single Rate Three Color Meter core components. For complete details, see Chapter 20, “Three Color Meter” of the *IXA Software Building Blocks Reference Manual*.

57.5.1 Data Structures

This section describes data structures used in calls to the TCM functional APIs—Message Helper API and Library API. Table 57-3 lists the two TCMTCM core component data structures.

Table 57-3. TCM Core Component Data Structures

Data Structure	Description
TCM Parameters Data Type	A set of configuration parameters for a meter instance.
TCM Statistics Data Type	A collection of statistics counters associated with a meter instance.

57.5.2 Core Component Infrastructure API

Table 57-4 lists the Core Component Infrastructure API supported by TCM core components. .

Table 57-4. TCM Core Component Infrastructure API

API Function	Description
<code>ix_cc_tc_meter_init()</code>	Initializes the core component
<code>ix_cc_tc_meter_fini()</code>	Terminates the core component
<code>ix_cc_tc_meter_pkt_handler()</code>	Messages handler for processing add/update/remove requests
<code>ix_cc_tc_meter_msg_handler()</code>	Packets handler for processing received packets

57.5.3 Message Helper API

Table 57-5 lists the functions which comprise the TCM Core Component Message Helper API.

Table 57-5. TCM Core Component Message Helper API

API Function	Description
<code>ix_cc_tc_meter_async_add_entry()</code>	Adds a new meter instance to the TCM table
<code>ix_cc_tc_meter_async_remove_entry()</code>	Removes a meter instance from the TCM table
<code>ix_cc_tc_meter_async_update_entry()</code>	Updates meter instance parameters in the TCM table
<code>ix_cc_tc_meter_async_get_statistics()</code>	Returns statistics associated with a meter instance

57.5.4 Library API

Table 57-6 lists the functions provided by the TCM Core Component Library API.

Table 57-6. TCM Core Component Library API

API Function	Description
<code>ix_cc_tc_meter_add_entry()</code>	Add a new meter instance to the TCM table
<code>ix_cc_tc_meter_remove_entry()</code>	Remove a meter instance from the TCM table
<code>ix_cc_tc_meter_update_entry()</code>	Update meter instance parameters in the TCM table
<code>ix_cc_tc_meter_get_statistics()</code>	Return statistics associated with a meter instance

57.6 Modularity

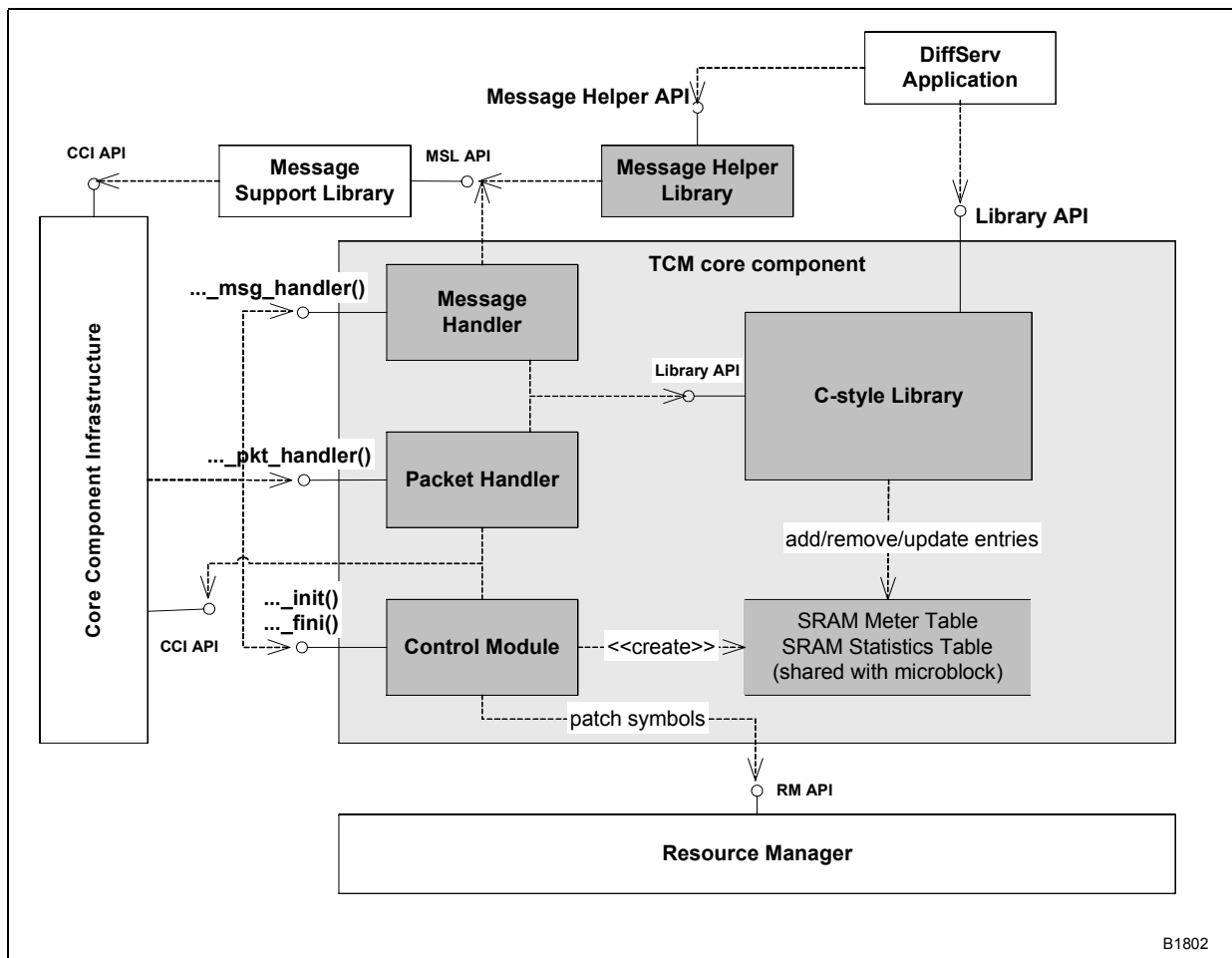
The classifier core component is comprised on the modules listed in Table 57-7.

Table 57-7. TCM Core Component Modules

Modules	Description
Control Module	This module controls initialization and shutdown process. It implements two Core Component Infrastructure API functions: <code>ix_cc_srtcm_init()</code> and <code>ix_cc_srtcm_fini()</code> .
Message Handler	This module implements Core Component Infrastructure API function <code>ix_cc_srtcm_msg_handler()</code> , registered as a single handler for all message types. Whenever a message arrives, the module parses a message payload and converts it to the corresponding Library API call. The module uses Message Support Library to return operation results in a user-defined callback routine.
Packet Handler	This module implements Core Component Infrastructure API function <code>ix_cc_srtcm_pkt_handler()</code> , registered as a single handler for all packet sources. Internally, the handler redirects all packets to the TCM microblock.
C-style Library	This module implements all Library API functions. This is the only module that directly accesses TCM table.
SRAM Meter Table	This is a data structure shared between TCM Core Component and SRCTM Microblock. Only the core component modifies table entries, while both the core component and microblock can read this table. The table layout is defined in Section 31.5, "Data Structures" on page 548 .

Figure 57-5 illustrates the architecture of the TCM Core Component.

Figure 57-5. TCM Core Component Architecture



B1802

In addition, the TCM Core Component comes with a wrapper library, the Message Helper Library, which implements all Message Helper API functions, as enumerated in [Section 57.5.3](#). This library constructs message payloads and sends them to a core component with a help of the Message Support Library.

Weighted Random Early Detection (WRED) Core Component

58

58.1 Overview

The Weighted Random Early Detection (WRED) Core Component provides the following functionalities:

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

For complete details on the WRED Microblock, see [Chapter 34, “Weighted Random Early Detection \(WRED\) Microblock”](#) and for external APIs see [Chapter 21, “Weighted Random Early Detection”](#) of the *IXA Software Building Blocks Reference Manual*.

58.2 Assumptions, Dependencies and Risks

58.2.1 Assumptions

The following design assumption is made:

- The core component does not process packets because a WRED algorithm contains a critical section. The core component only configures WRED instances.

58.2.2 Dependencies

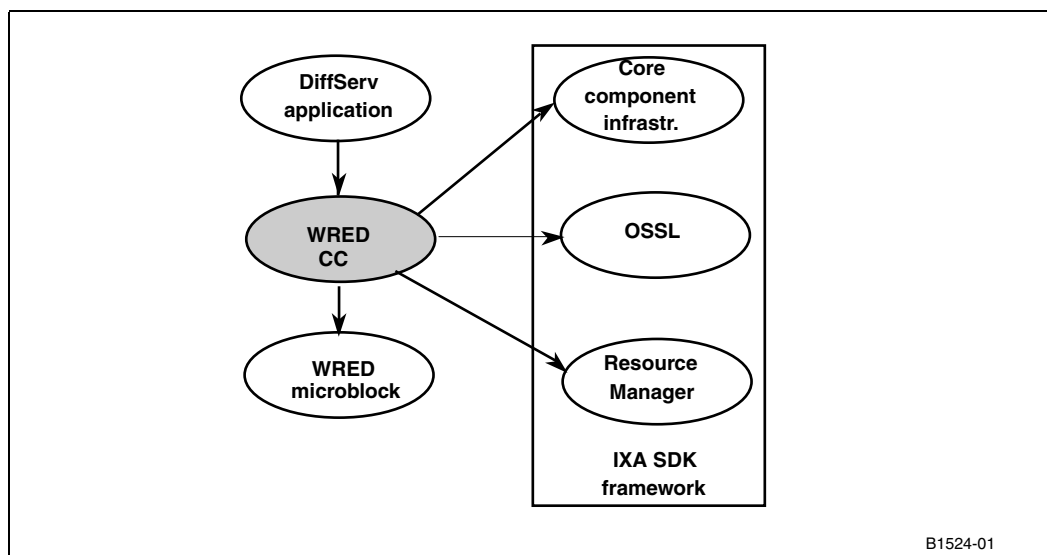
The WRED core component depends on the IXA SDK framework. It uses the services of Resource Manager for:

- allocating and freeing SRAM memory
- patching symbols
- accessing system registry to retrieve static parameters

The Core Components Infrastructure services are used for message and packet handling as well as sending packets to the WRED microblock. The Operating System Service Layer (OSSL) makes the core component code independent of the underlying OS.

The core component configures a WRED table used by the microblock. Execution of the microblock does not impact the core component.

Figure 58-1. WRED Core Component Dependencies



58.3 Configuration and Initialization

58.3.1 Static Configuration Data

The WRED Core Component defines the static configuration data listed in [Table 58-1](#).

Table 58-1. Static Configuration Items in WRED core component

Data (system property name)	Default	Description
WRED_ENTRY_SRAM_SIZE (\\WRED\\ENTRY_SRAM_SIZE)	64	Number of bytes occupied by one entry in a WRED table. Used to patch the imported variable and to calculate the amount of memory allocated for the table.
WRED_64BITSTAT_ENTRY_SRAM_SIZE (\\WRED\\64BITSTAT_ENTRY_SRAM_SIZE)	16	Number of bytes occupied by higher parts of 64-bit statistics counters associated with a WRED entry. Used to patch the imported variable and to calculate the amount of memory allocated for the table.
WRED_TABLE_SRAM_SIZE (\\WRED\\TABLE_SRAM_SIZE)	256	Number of entries in a WRED table and statistics table. Used to calculate the amount of memory allocated for the WRED and statistics tables.

These data are obtained from the system repository during the component initialization time. If the system repository is not present, then the default values defined in the component's global header file, `ix_cc_wred.h`, is used. The values are configuration parameters to be patched into the microblocks.

58.3.2 Dynamic Configuration Data

The dynamic configuration is pertinent to adding/removing entries from the WRED table.

58.3.3 Patching Symbols

The patching symbols correspond to setting an SRAM base address for the WRED table, Queue Descriptor table and 64-bit statistics table. Patching symbols are listed in [Table 58-2](#). They have no default values. A core component patches these symbols upon initialization.

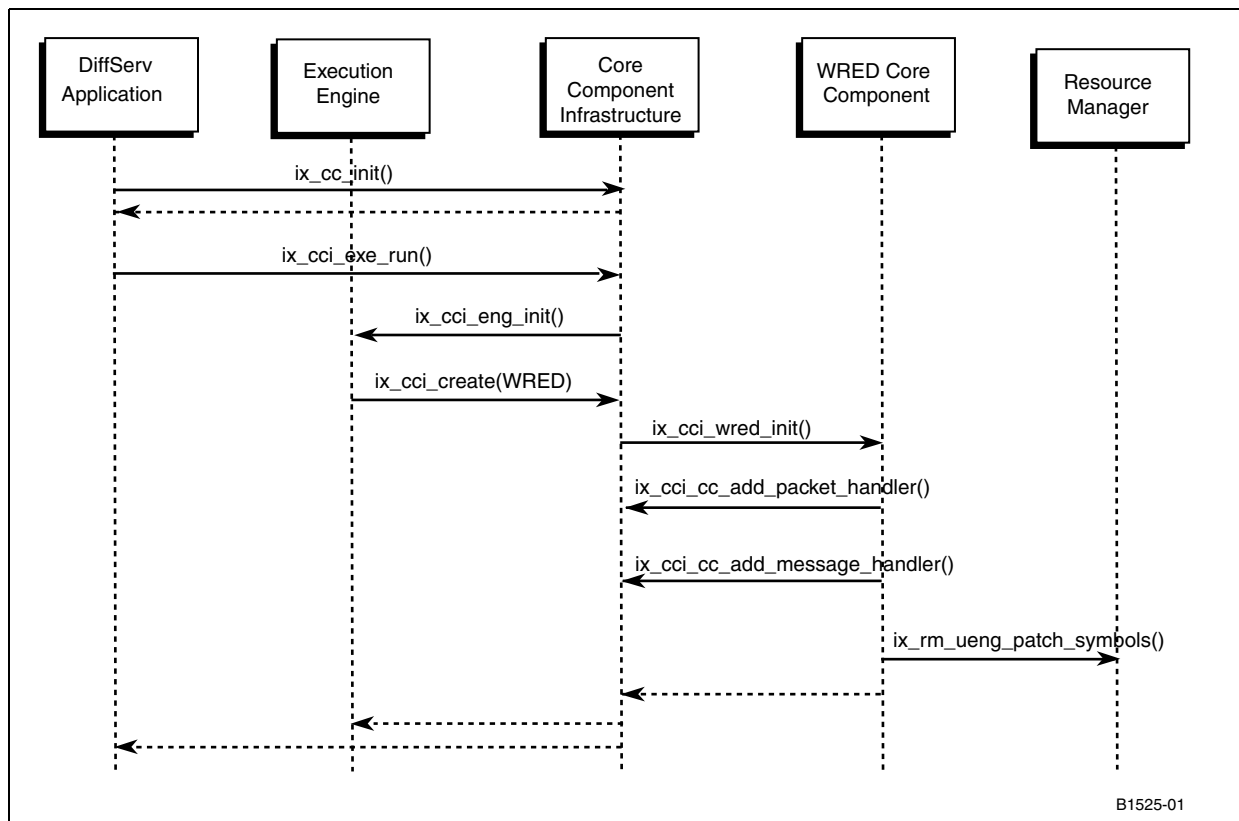
Table 58-2. Variables Imported by this Block

Variable	Default	Description
WRED_TABLE_SRAM_BASE	-	Base address of the WRED table maintained in SRAM
QD_SRAM_BASE	-	Base address of queue descriptors table maintained in SRAM by Queue Manager
WRED_64BIT_STAT_SRAM_BASE	-	Base address of the statistics table storing most significant long words of 64-bit long counters.

58.3.4 Initialization and Shutdown Data Flow

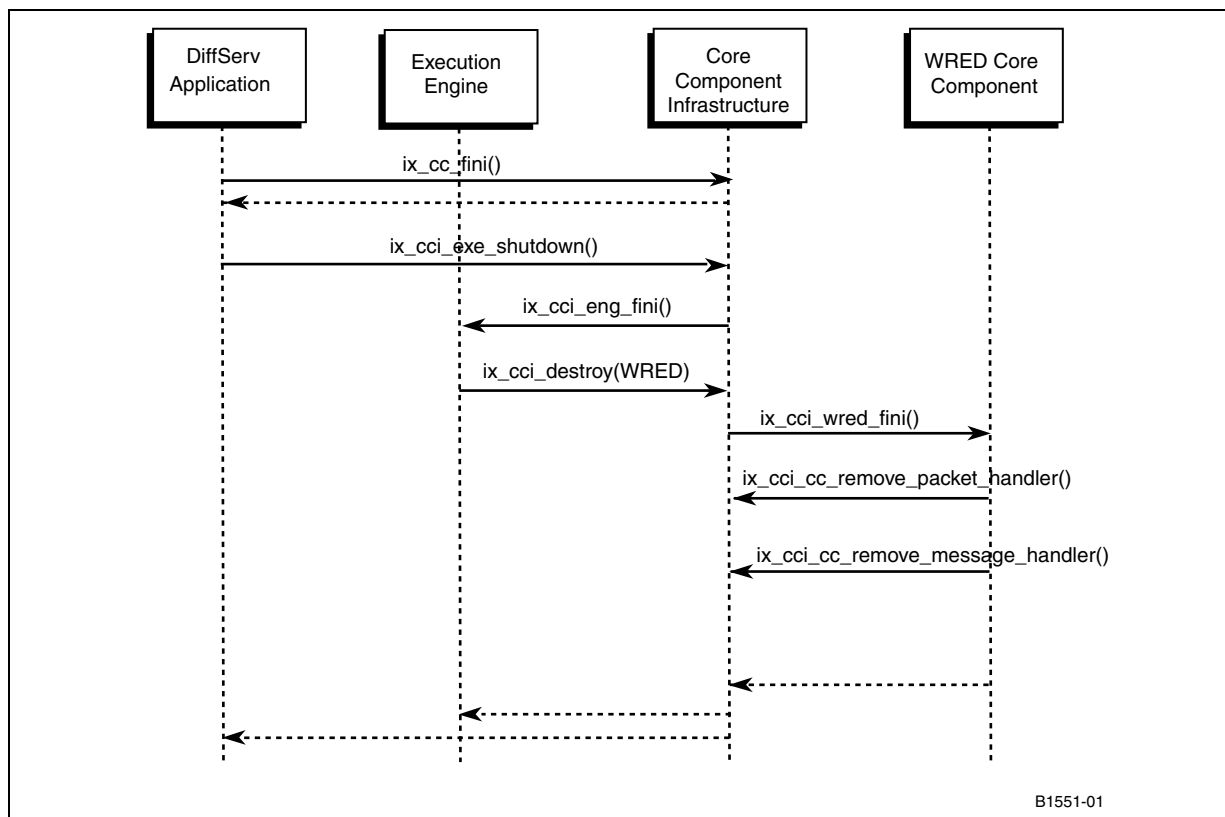
Figure 58-2 and Figure 58-3 show a sequence of calls upon system initialization and shutdown.

Figure 58-2. Initialization of the WRED Core Component



The WRED Core Component needs an SRAM base address for the Queue Descriptor table. This value should be stored in a system registry.

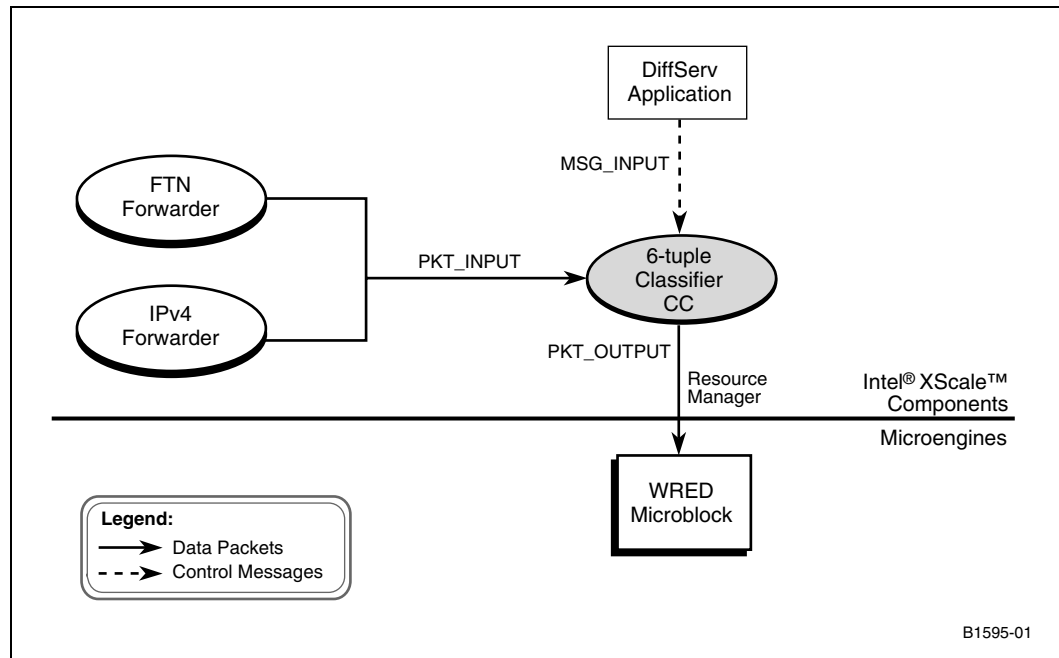
Figure 58-3. Shutdown of the WRED Core Component



58.4 Data and Control Flow

Figure 58-4 illustrates WRED core component data flow.

Figure 58-4. Data flow of the WRED Core Component



58.4.1 Packet Inputs

The WRED Core Component defines one input on which it receives packets. This can be a multi-source input (e.g. when MPLS forwarder is used). The input identifier is defined in a system file, `bindings.h` as:

```
#define IX_CC_WRED_PKT_INPUT
```

58.4.2 Packet Outputs

The WRED Core Component defines one packet output. All packets are moved to the WRED microblock. The output identifier is defined in `bindings.h` as:

```
#define IX_CC_WRED_PKT_OUTPUT
```

58.4.3 Message Inputs

The WRED Core Component defines one input for all configuration messages. This is a single-source input defined in `bindings.h` as:

```
#define IX_CC_WRED_MSG_INPUT
```

58.5 External API

This section lists the Weight Random Early Detection core components. For complete details, see Chapter 21, “Weighted Random Early Detection” of the *IXA Software Building Blocks Reference Manual*.

58.5.1 Data Structures

This section describes data structures used by the Message Helper API and Library API. The WRED core component defines three data structures.

Table 58-1. WRED Core Component Data Structures for Message Helper and Library APIs

Data Structure	Description
<code>ix_s_cc_red_instance</code>	A set of configuration parameters that are color-specific.
<code>ix_s_cc_wred_parameters</code>	A set of configuration parameters for a WRED-protected queue.
<code>WRED Statistics Data Type</code>	A collection of statistics counters associated with a WRED instance.

58.5.2 Core Component Infrastructure API

The WRED Core Component supports the Core Component Infrastructure API primitives listed in Table 58-2.

Table 58-2. WRED Core Component Data Structures for Message Helper and Library APIs

Data Structure	Description
<code>ix_s_cc_red_instance</code>	A set of configuration parameters that are color-specific.
<code>ix_s_cc_wred_parameters</code>	A set of configuration parameters for a WRED-protected queue.
<code>WRED Statistics Data Type</code>	A collection of statistics counters associated with a WRED instance.

58.5.3 Message Helper API

The Message Helper is a wrapper library that facilitates sending messages to the WRED Core Component. There is one-to-one mapping between Message Helper API primitives and message types supported by a core component. All API functions are asynchronous: a core component reports operation status in a callback routine.

Table 58-3 lists the functions in the WRED Core Component Message Helper API.

Table 58-3. WRED Message Helper API

API Function	Description
<code>ix_cc_wred_async_add_entry()</code>	Adds a new WRED instance to the parameters table
<code>ix_cc_wred_async_remove_entry()</code>	Removes a WRED instance from the parameters table
<code>ix_cc_wred_async_update_entry()</code>	Updates a WRED instance parameters in the parameters table
<code>ix_cc_wred_async_get_statistics()</code>	Returns statistics associated with a WRED instance

58.5.4 Library API

The Library API provides direct C-style calls to the classifier core component. The library API functions correspond one-to-one with message helper API primitives. The difference is that library functions are executed in a caller context. In addition, the Library API is synchronous. There are no callback functions to report operation results.

Table 58-4 lists the functions in the WRED Core Component Library API..

Table 58-4. WRED Library API

API Function	Description
<code>ix_cc_wred_add_entry()</code>	Adds a new WRED instance to the parameters table
<code>ix_cc_wred_remove_entry()</code>	Removes a WRED instance from the parameters table
<code>ix_cc_wred_update_entry()</code>	Updates WRED instance parameters in the parameters table
<code>ix_cc_wred_get_statistics()</code>	Returns statistics associated with a WRED instance

58.6 Modularity

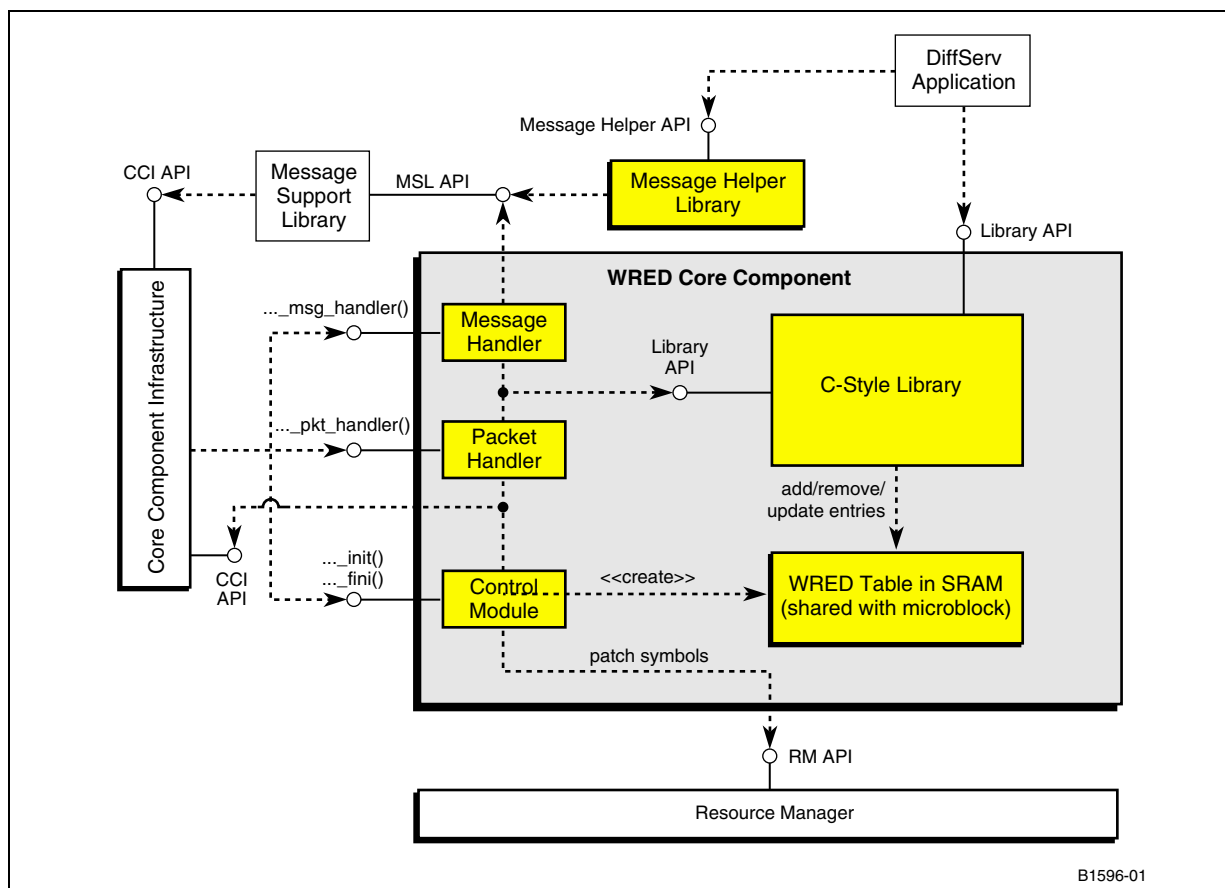
The classifier core component comprises the following modules:

Table 58-3. WRED Core Component Modules

Module	Description
Control Module	This module controls initialization and shutdown process. It implements two Core Component Infrastructure API functions: <code>ix_cc_wred_init()</code> and <code>ix_cc_wred_fini()</code> .
Message Handler	This module implements the Core Component Infrastructure API function <code>ix_cc_wred_msg_handler()</code> , registered as a single handler for all message types. Whenever a message arrives, the module parses a message payload and converts it to the corresponding Library API call. The module uses Message Support Library to return operation results in a user-defined callback routine.
Packet Handler	This module implements the Core Component Infrastructure API function <code>ix_cc_wred_pkt_handler()</code> , registered as a single handler for all packet sources. Internally, the handler redirects all packets to the WRED microblock.
C-style Library	This module implements all Library API functions, as enumerated in Section 58.5.4 . This is the only module that directly accesses the WRED table.
WRED Table	This is a data structure shared between the WRED Core Component and the WRED microblock. Only the core component modifies table entries, while both the core component and microblock can read the table. The table layout is defined in Section 34.5, "Data Structures" on page 586 .

Figure 58-1 illustrates the architecture of the WRED Core Component.

Figure 58-1. WRED Core Component Architecture



In addition, the WRED Core Component comes with a wrapper library, the Message Helper Library, which implements all Message Helper API functions, as enumerated in [Section 58.5.3](#). This library constructs message payloads and sends them to a core component with a help of the Message Support Library.

59.1 Overview

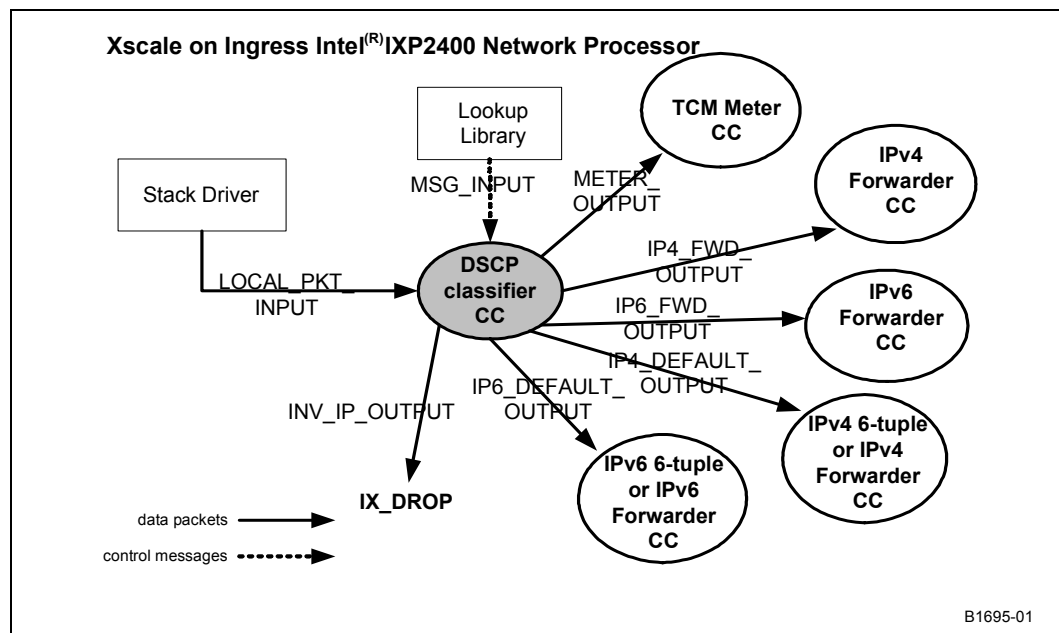
The DSCP Classifier core component provides the following functions:

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

59.1.1 Data and Control Flow

Figure 59-1 illustrates the Data flow of DSCP classifier core component.

Figure 59-1. Data flow of DSCP Classifier Core Component



59.1.1.1 Packet Inputs

The DSCP classifier core component defines a single packet input dedicated for local IP packets generated by the stack driver.

The input identifier is defined in a system file, `bindings.h` as follows:

```
#define IX_CC_CLASSIFIER_DSCP_LOCAL_PKT_INPUT
```

59.1.1.2 Packet Outputs

The core component defines four outputs for the following:

- Packets subjected to traffic conditioning action (for example TCM meter),
- IPv4 packets not requiring additional QoS action on ingress (for example IPv4 forwarder),
- IPv6 packets not requiring additional QoS action on ingress,
- Packets being neither IPv4 or IPv6 packets (for example IX_DROP output),
- Packets received from ports for which no DSCP classification rules has been configured (for example IPv4 6-tuple classifier).

Output identifiers are defined in `bindings.h` as:

```
#define IX_CC_CLASSIFIER_DSCP_METER_OUTPUT
#define IX_CC_CLASSIFIER_DSCP_IP4_FWD_OUTPUT
#define IX_CC_CLASSIFIER_DSCP_IP6_FWD_OUTPUT
#define IX_CC_CLASSIFIER_DSCP_INV_IP_OUTPUT
#define IX_CC_CLASSIFIER_DSCP_IP4_DEFAULT_OUTPUT
#define IX_CC_CLASSIFIER_DSCP_IP6_DEFAULT_OUTPUT
```

In addition, the core component bindings must be consistent with microblock bindings. [Table 59-1](#) shows a mapping between output identifiers.

Table 59-1. Mapping between Output Identifiers

Microblock Output ID	Core Component Output ID
CLASSIFIER_DSCP_METER	IX_CC_CLASSIFIER_DSCP_METER_OUTPUT
CLASSIFIER_DSCP_IP4_FWD	IX_CC_CLASSIFIER_DSCP_IP4_FWD_OUTPUT
CLASSIFIER_DSCP_IP6_FWD	IX_CC_CLASSIFIER_DSCP_IP6_FWD_OUTPUT
CLASSIFIER_DSCP_INV_IP	IX_CC_CLASSIFIER_DSCP_INV_IP_OUTPUT
CLASSIFIER_DSCP_IP4_DEFAULT_OUTPUT	IX_CC_CLASSIFIER_DSCP_IP4_DEFAULT_OUTPUT
CLASSIFIER_DSCP_IP6_DEFAULT_OUTPUT	IX_CC_CLASSIFIER_DSCP_IP6_DEFAULT_OUTPUT

59.1.1.3 Message Inputs

The core component defines a single input for all configuration messages.

```
#define IX_CC_CLASSIFIER_DSCP_MSG_INPUT
```

The DSCP core component receives messages from a single logical source: the DiffServ Application. However, the core component should not assume that the library serializes invocations from multiple upper-layer applications. Thus, the message input should be protected with mutual exclusion mechanisms.

59.2 Assumptions, Dependencies and Risks

59.2.1 Assumptions

The following design assumptions are made:

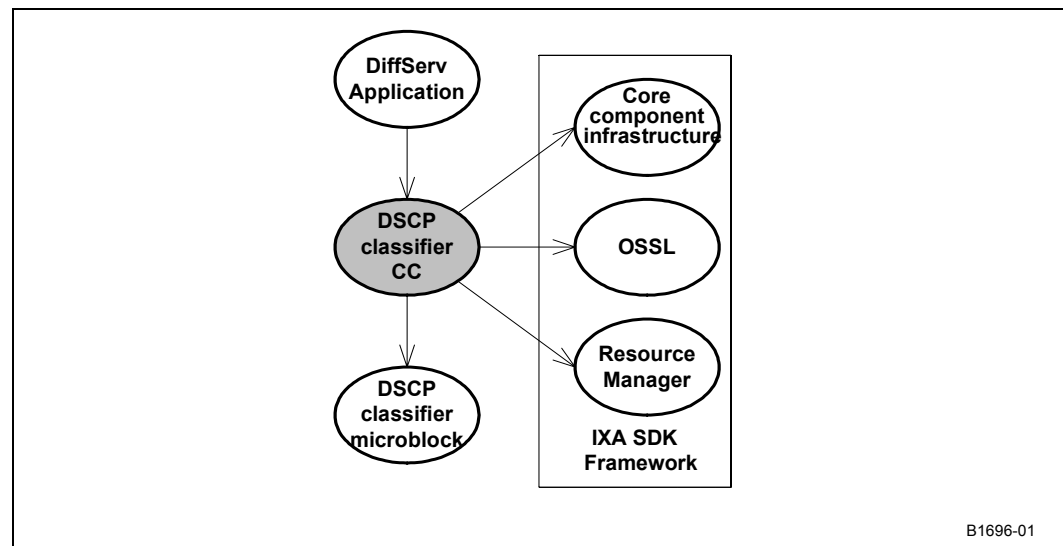
- The control interface is compliant with Lookup Library API.
- The core component configures a default rule in microblock by populating the configuration entries that do not have interface-specific rule with the default rule.

59.2.2 Dependencies

The DSCP classifier core component depends on the IXA SDK framework. It uses the services of Resource Manager (see *Intel® Internet Exchange Architecture Portability Framework Developer's Manual*) for allocating and freeing SRAM memory, patching symbols, and accessing system registry to retrieve static parameters. The Core Components Infrastructure services are used for message and packet handling between other core components. The Operating System Service Layer (OSSL) makes the core component code independent of the underlying OS.

The core component configures a SRAM configuration table entry used by the DSCP classifier microblock. Execution of the microblock, however, does not impact the core component.

Figure 59-2. DSCP Classifier Core Component Dependencies



59.3 Configuration and Initialization

59.3.1 Static Configuration Data

The DSCP classifier core component defines the following static configuration data.

Table 59-2. Static Configuration Items in 6-tuple Classifier

Data (system property name)	Default	Description
CLASSIFIER_DSCP_CONFIG_ENTRY_SRAM_SIZE (\\CLASSIFIER_DSCP\\CONFIG_ENTRY_SRAM_SIZE)	8	Number of bytes occupied by a single entry in a configuration table. It is used to calculate the size of the configuration table.
CLASSIFIER_DSCP_STATS_ENTRY_SRAM_SIZE (\\CLASSIFIER_DSCP\\STATS_ENTRY_SRAM_SIZE)	16	Number of bytes occupied by a single entry in a statistics table. It is used to calculate the size of hash table.
CLASSIFIER_DSCP_DEFAULT_STATS_ENABLE (\\CLASSIFIER_DSCP\\DEFAULT_STATS_ENABLE)	0	Initial value of the statistics flag used to populate the configuration table at the core component initialization.
CLASSIFIER_DSCP_DEFAULT_FLOW_ID (\\CLASSIFIER_DSCP\\DEFAULT_FLOW_ID)	0	Initial value of flow ID used to populate the configuration table at the core component initialization. It is applied to all the interfaces and DSCP values.
CLASSIFIER_DSCP_DEFAULT_CLASS_ID (\\CLASSIFIER_DSCP\\DEFAULT_CLASS_ID)	8	Initial value of class ID used to populate the configuration table at the core component initialization. It is applied to all the interfaces and DSCP values.
CLASSIFIER_DSCP_DEFAULT_COLOR_ID (\\CLASSIFIER_DSCP\\DEFAULT_COLOR_ID)	0 (green)	Initial value of flow ID used to populate the configuration table at the core component initialization. It is applied to all the interfaces and DSCP values.
CLASSIFIER_DSCP_IN_PORTS (\\CLASSIFIER_DSCP\\IN_PORTS)	256	The number of input ports in the system. The value is used to calculate the size of configuration and statistics table.
CLASSIFIER_DSCP_PACKET_COUNTER_FEATURE (\\CLASSIFIER_DSCP\\PACKET_COUNTER_FEATURE)	0	A flag indicating whether the microblock supports statistics.

The core component uses the CLASSIFIER_DSCP_PACKET_COUNTER_FEATURE flag to decide whether to allocate the statistics table in SRAM and patch CLASSIFIER_DSCP_STATS_SRAM_BASE symbol.

These data will be obtained from the system repository during the component initialization time. If the system repository is not present, then the default values defined in the component's global header file, ix_cc_classifier_dscp_defs.h, will be used.

59.3.2 Dynamic Configuration Data

The dynamic configuration is pertinent to changing classification rules in the configuration table.

59.3.2.1 Patching Symbols

All patching symbols are enumerated in [Section 33.2.4.3, “Imported Variables”](#) on page 572. The core component patches all symbols during initialization.

59.3.3 Initialization and Shutdown Data Flow

The figures 59-3 and 59-4 show a sequence of calls upon system initialization and shutdown.

Figure 59-3. Initialization of a DSCP Classifier Core Component

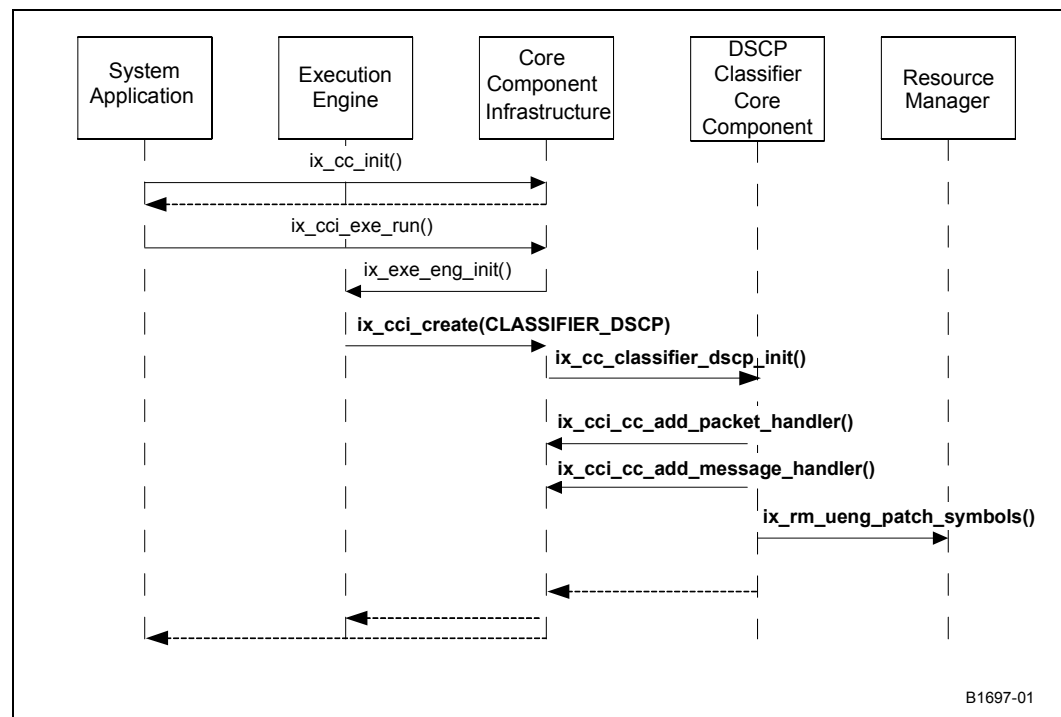
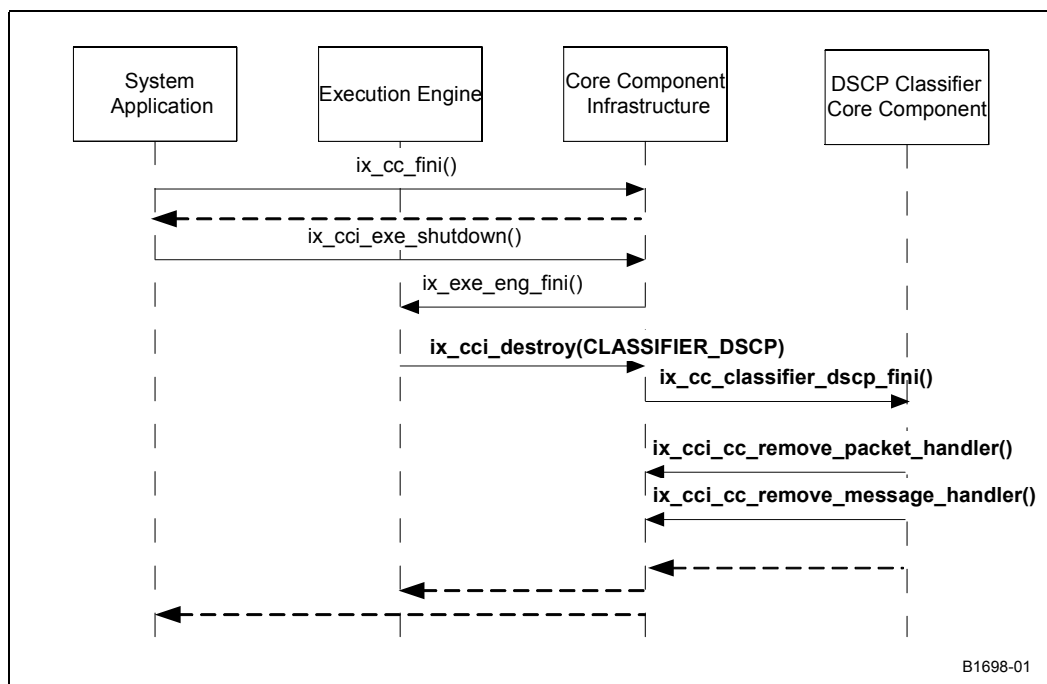


Figure 59-4. Shutdown of a DSCP Classifier Core Component



59.4 External API

This section lists the Single Rate Three Color Meter core components. For complete details, see Chapter 20, “Three Color Meter” of the *IXA Software Building Blocks Reference Manual*.

59.4.1 Data Structures

This section describes data structures used in calls to the SRTCM functional APIs—Message Helper API and Library API. Table 59-3 lists the two SRTCM core component data structures.

Table 59-3. SRTCM Core Component Data Structures

Data Structure	Description
TCM Parameters Data Type	A set of configuration parameters for a meter instance.
TCM Statistics Data Type	A collection of statistics counters associated with a meter instance.

59.4.2 Core Component Infrastructure API

Table 59-4 lists the Core Component Infrastructure API supported by SRTCM core components. .

Table 59-4. SRTCM Core Component Infrastructure API

API Function	Description
<code>ix_cc_tc_meter_init()</code>	Initializes the core component
<code>ix_cc_tc_meter_fini()</code>	Terminates the core component
<code>ix_cc_tc_meter_pkt_handler()</code>	Messages handler for processing add/update/remove requests
<code>ix_cc_tc_meter_msg_handler()</code>	Packets handler for processing received packets

59.4.3 Message Helper API

Table 59-5 lists the functions which comprise the SRTCM Core Component Message Helper API.

Table 59-5. SRTCM Core Component Message Helper API

API Function	Description
<code>ix_cc_tc_meter_async_add_entry()</code>	Adds a new meter instance to the SRTCM table
<code>ix_cc_tc_meter_async_remove_entry()</code>	Removes a meter instance from the SRTCM table
<code>ix_cc_tc_meter_async_update_entry()</code>	Updates meter instance parameters in the SRTCM table
<code>ix_cc_tc_meter_async_get_statistics()</code>	Returns statistics associated with a meter instance

59.4.4 Library API

Table 59-6 lists the functions provided by the SRTCM Core Component Library API.

Table 59-6. SRTCM Core Component Library API

API Function	Description
<code>ix_cc_tc_meter_add_entry()</code>	Add a new meter instance to the SRTCM table
<code>ix_cc_tc_meter_remove_entry()</code>	Remove a meter instance from the SRTCM table
<code>ix_cc_tc_meter_update_entry()</code>	Update meter instance parameters in the SRTCM table
<code>ix_cc_tc_meter_get_statistics()</code>	Return statistics associated with a meter instance

59.5 Modularity

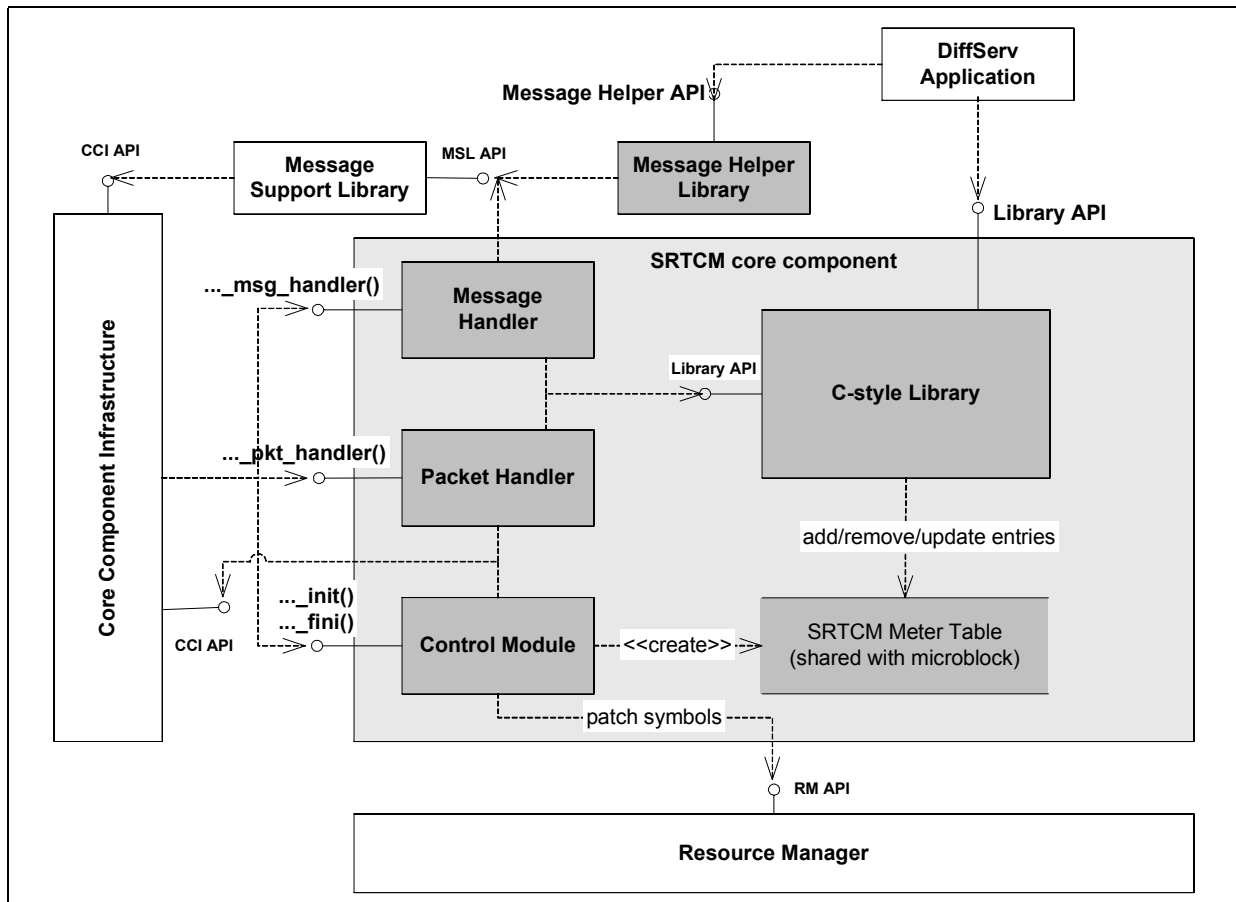
The classifier core component is comprised on the modules listed in [Table 59-7](#).

Table 59-7. SRTCM Core Component Modules

Modules	Description
Control Module	This module controls initialization and shutdown process. It implements two Core Component Infrastructure API functions: <code>ix_cc_srtcm_init()</code> and <code>ix_cc_srtcm_fini()</code> .
Message Handler	This module implements Core Component Infrastructure API function <code>ix_cc_srtcm_msg_handler()</code> , registered as a single handler for all message types. Whenever a message arrives, the module parses a message payload and converts it to the corresponding Library API call. The module uses Message Support Library to return operation results in a user-defined callback routine.
Packet Handler	This module implements Core Component Infrastructure API function <code>ix_cc_srtcm_pkt_handler()</code> , registered as a single handler for all packet sources. Internally, the handler redirects all packets to the SRTCM microblock.
C-style Library	This module implements all Library API functions. This is the only module that directly accesses SRTCM table.
SRAM Meter Table	This is a data structure shared between SRTCM Core Component and SRCTM Microblock. Only the core component modifies table entries, while both the core component and microblock can read this table. The table layout is defined in Section 31.5, "Data Structures" on page 548 .

[Figure 59-5](#) illustrates the architecture of the SRTCM Core Component.

Figure 59-5. SRTCM Core Component Architecture



In addition, the SRTCM Core Component comes with a wrapper library, the Message Helper Library, which implements all Message Helper API functions, as enumerated in [Section 59.4.3](#). This library constructs message payloads and sends them to a core component with a help of the Message Support Library.

Support Libraries

The Support libraries include the following:

- [Chapter 60, “Route Table Manager”](#)

Route Table Manager(RTM) for IPv4 pipeline primarily supports the IPv4 Forwarder Core Component.

RTM provides services to the other core components for maintaining route tables. It provides a way to create a new table, delete an existing table, add routes to a table, remove routes from a table, and look up a route in a table.
- [Chapter 61, “Route Table Manager for IPV6 Core Component”](#)

Route Table Manager(RTM) for IPv6 pipeline primarily supports the IPv6 Forwarder Core Component.

RTM provides services to the other core components for maintaining route tables. It provides a way to create a new table, delete an existing table, add routes to a table, remove routes from a table, and look up a route in a table.
- [Chapter 62, “L2 Table Manager”](#)

L2 Table Manager exposes API for initializing, managing, updating and searching L2 table.
- [Chapter 63, “Message Helper and Support Library”](#)

Message Support Library provides the layer of translation between messages and core component APIs. Message Helper Library translates message API into the messages and enforces the mechanism of delivering messages to the core component Library APIs and sending the result back to the calling application.

The following table lists the libraries supported on the Ingress and Egress side:

Libraries needed on th Ingress side	Libraries needed on the Egress side
Route Table Manager for IPv4	L2 Table Manager
Route Table Manager for IPv6	Message Helper and Message Support Library
Message Helper and Message Support Library	

60.1 Overview

The Route Table Manager (RTM) is utilized by the IPv4 Forwarder Core Component to add, remove or look up items in the route table. Route information is maintained and searched by a Longest Prefix Match (LPM) algorithm. To manage next hops, the Route Table Manager uses a Next Hop Database (NHDB).

The Route Table Manager uses data structures and stores the data in the same format as the IPv4 Forwarder microblock. These data structures are allocated through the Resource Manager.

This implementation of the Route Table Manager is specific to the IP protocol, version 4. Because a Route Table Manager designed for IPv6 would require a different interface (e.g. 128-bit parameters instead of 32-bit), the Route Table Manager exposes its entry points in such a way that it only deals with IPv4 data types and algorithms.

The Route Table Manager uses a lookup library and therefore could be used with TCAM hardware. For external APIs see [Chapter 23, “Route Table Manager”](#) of the *IXA Software Building Blocks Reference Manual*.

60.2 Usage Model

The IPv4 Forwarder Core Component uses the services of the Route Table Manager.

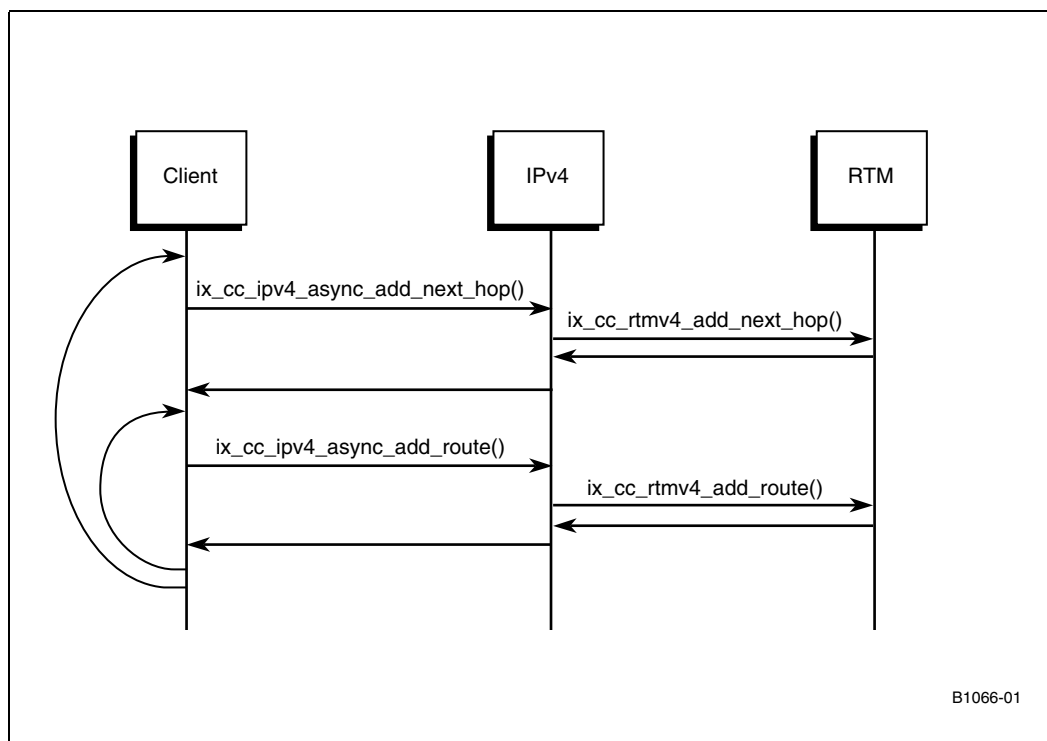
This section contains the following subsections:

- [“Using Routes and Next Hops”](#)
- [“Initialization”](#)
- [“Pre-assigned Next Hop Identifiers”](#)
- [“Default Route”](#)

60.2.1 Using Routes and Next Hops

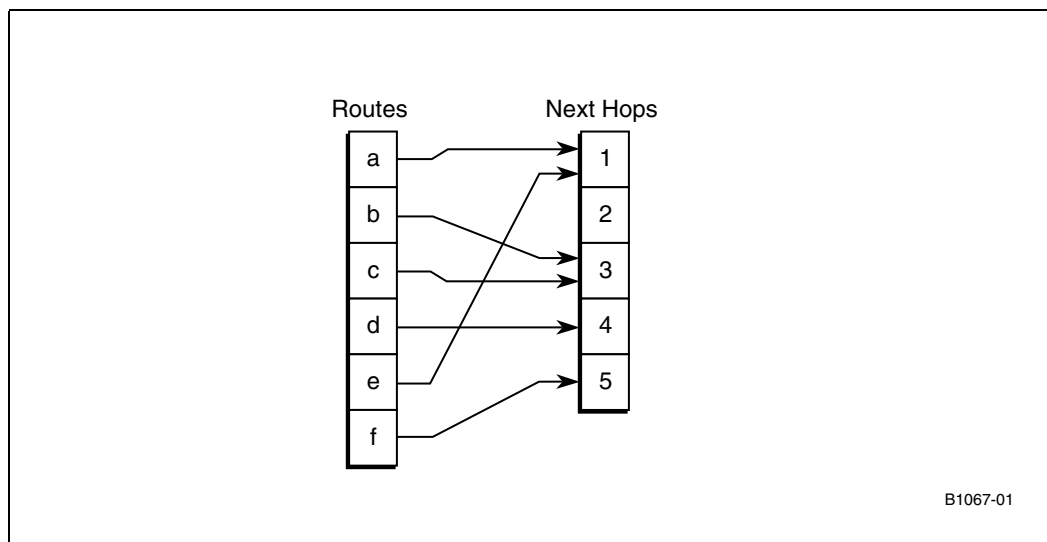
The IPv4 Forwarder Core Component uses the Route Table Manager to store and retrieve next hops and routes. Because routes refer to next hops, the client must store at least one next hop before adding routes, as shown in [Figure 60-1](#). A typical client of the IPv4 Forwarder may add one next hop and zero or more routes referring to that next hop. It may then repeat the cycle with more next hops and routes.

Figure 60-1. Adding Information using the Route Table Manager



A route may refer to any existing next hop and many routes may refer to the same next hop. The Route Table Manager does not require that a next hop be referenced by a route, but a route must reference a next hop. This relationship is illustrated in [Figure 60-2](#).

Figure 60-2. Routes and Next Hops



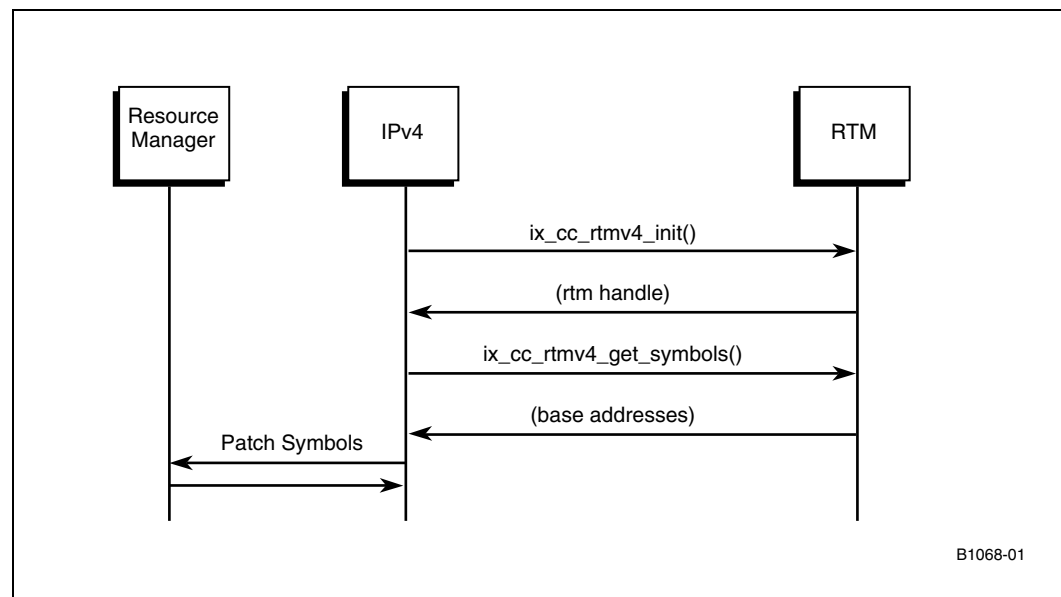
All routes refer to one next hop, but not all next hops are associated with a route.

Similarly, the Route Table Manager places restrictions on when next hops may be removed. Just as a next hop must be in place before a route (including the default route described in [Section 60.2.4, “Default Route” on page 863](#)) uses it, the next hop must remain valid for as long as the route exists. Therefore, the client must remove any routes using a specific next hop before that next hop may be removed. For example, in [Figure 60-2](#), if the client wanted to remove next hop #3, it must first remove routes 'b' and 'c'. However, since next hop #2 is not being referenced by any routes, it could be removed immediately.

60.2.2 Initialization

[Figure 60-3](#) shows how the Route Table Manager is initialized. The client (usually the IPv4 Forwarder Core Component) is responsible for creating the Route Table Manager and adding next hops and routes. It may then look up next hop information based on an IP address.

Figure 60-3. Route Table Manager Initialization



60.2.3 Pre-assigned Next Hop Identifiers

The Route Table Manager allows the client to assign Next Hop Identifiers (NHIDs), but an NHID with a value of `IX_CC_RTMV4_NHID_NO_ROUTE` (0xFFFFFFFF) has special meaning within the Route Table Manager. The client must not attempt to use this value when adding or deleting routes or next hops. The Route Table Manager detects this condition and returns an error.

60.2.4 Default Route

The Route Table Manager supports an optional default route. The client may set the default route by adding a route of 0.0.0.0/0, referring to a valid next hop. The client may remove the route by deleting 0.0.0.0/0.

60.3 Design Criteria

The IPv4 Forwarder Core Component and IPv4 Forwarder microblock rely on the Route Table Manager to provide specific functionality in each of the following areas:

- “Data Storage and Retrieval”
- “Guaranteed Table Validity”
- “Multiple Table Support”
- “Control Plane Design”
- “Microengine Timing”
- “High-level Route Table”
- “Lookup Library Initialization”
- “Single Direct Client”
- “Duplicate Routes”

60.3.1 Data Storage and Retrieval

The Route Table Manager stores and retrieves routes and next hops. In addition, the Route Table Manager must use the data stored to determine which next hop is appropriate for a given IP address. The Route Table Manager uses a Longest Prefix Match algorithm to make this determination.

60.3.2 Guaranteed Table Validity

Microengines have an extremely limited number of cycles in which to process packets. There is no room in the budget for them to perform error checking or synchronize with the core component. This implies several things.

- The core component must ensure the microengines always have access to valid data, because the microengines cannot test the data for themselves.
- The entire burden of synchronization falls on the core component.

In this case, the IPv4 Forwarder microblock relies on the Route Table Manager to handle these issues on its behalf.

The Route Table Manager guarantees valid data by adding and removing data in a specific order. However, the timing for removing routes and next hops is somewhat more difficult. The Route Table Manager removes items from the route table and next hop database in such a way that once the microblock has initiated a lookup, it always gets a valid response.

60.3.3 Multiple Table Support

The Route Table Manager supports multiple route tables by creating one route table and next hop database per Route Table Manager. There is a one-to-one mapping between Route Table Manager instances, route tables and next hop databases. If the IPv4 Forwarder Core Component wants two separate route tables, it would achieve this by calling `ix_cc_rtmv4_init()` twice.

60.3.4 Control Plane Design

The design of the Control Plane PDK, which is based on the NPF API, affects the design of the Route Table Manager to some extent. Some examples include:

- Adding a default route via `add_route` vs. `set_default_route`. The industry in general overloads the `add_route` operation to allow setting the default route, and the Route Table Manager follows this convention.
- Separating next hops from routes. The Route Table Manager follows a model where next hops and routes are treated separately, allowing many-to-one relationships between routes and next hops. The Control Plane PDK and NPF API uses this as well.
- Netmask vs. netmask length. The `add_route` and `del_route` operations can use a netmask (e.g. `0xFF000000`) or a netmask length (e.g. `8`). There are advantages and disadvantages to both; the Route Table Manager uses the former to remain consistent with the Control Plane PDK.
- NHIDs are defined by the caller with the exception of `0xFFFFFFFFC0` (-64) to `0xFFFFFFFF` (-1).

60.3.5 Microengine Timing

In order to ensure that the microengines always have valid data, as described in [Section 60.3.2, “Guaranteed Table Validity” on page 864](#), some knowledge of the microblock’s timing is required. Otherwise, it is possible that modifications to the microblocks’ timing could cause it to use route entries longer than expected, leading to difficult-to-find bugs where the microengines access invalid or re-allocated memory. Microengine timing may affect `ix_cc_rtmv4_delete_route()` and `ix_cc_rtmv4_delete_next_hop()`.

60.3.6 High-level Route Table

It is anticipated that higher-level clients (i.e. the Control Plane PDK or its clients) maintain a complete list of Next Hops and Routes for various reasons. High-level clients may be required to use their own lists of routes and next hops in order to remove next hops.

60.3.7 Lookup Library Initialization

The Route Table Manager relies on the Lookup Library to allow multiple calls to `ix_lkup_sw_init()` or `ix_lkup_tcam_init()`. Only the first call initializes the library; subsequent calls simply return a handle. This allows multiple clients of the library to exist (e.g. RTMv4 and RTMv6) without coordinating between each other. This also allows a single Route Table Manager to instantiate several tables without keeping the `lkup_handle` in a global location.

The Route Table Manager requires the Lookup Library to allow multiple clients, each with their own table(s). Multiple clients must not share a table, but two clients must be able to have their own tables without interfering.

60.3.8 Single Direct Client

The Route Table Manager supports a single client, the IPv4 Forwarder Core Component. The Route Table Manager is currently designed to only work with the current implementation of the IPv4 Forwarder Core Component.

While the Route Table Manager has design restrictions requiring a single client at run-time, the implementation of the Route Table Manager does not preclude other clients from using the Route Table Manager in a different environment.

60.3.9 Duplicate Routes

The Route Table Manager does not support duplicate routes - i.e. routes where the same network address / network mask pair refer to two different next hops. Attempting to add the same route twice results in an error. Supporting duplicate routes would require an algorithm to determine which route to use, which is beyond the scope of this implementation.

60.4 Modularity

The Route Table Manager's interface is based loosely on the Façade pattern. In brief, the Façade pattern provides a unified interface to a set of interfaces in a subsystem. The Route Table Manager API fills the role of the Façade; the Next Hop Database, Lookup API, Software LPM and TCAM are the subsystem classes.

Figure 60-4. Modularity of the Route Table Manager

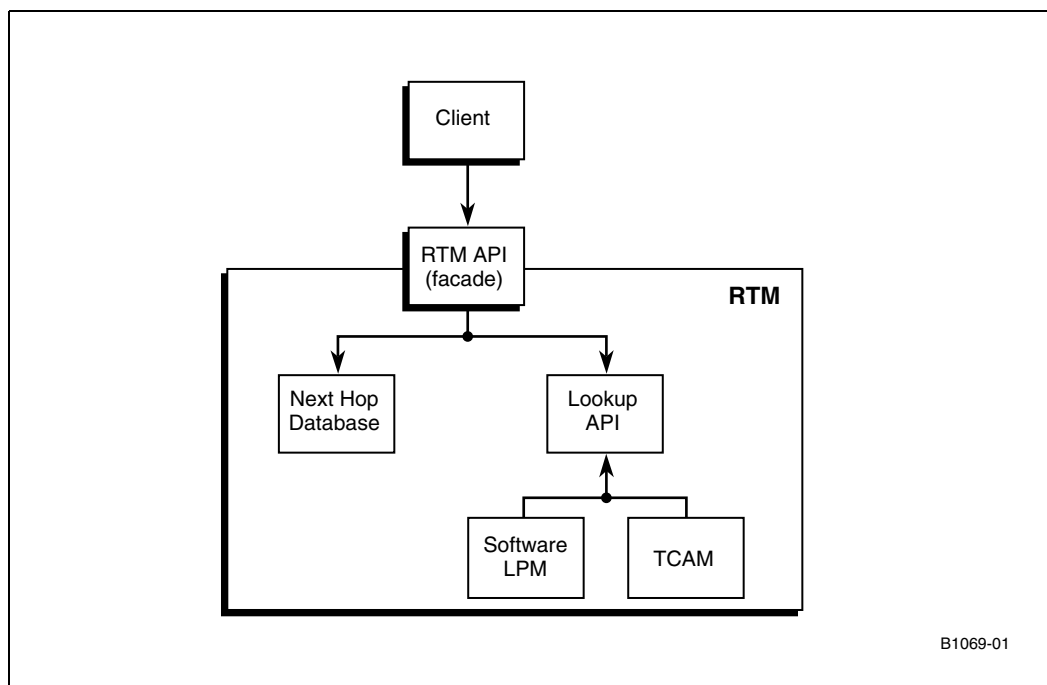
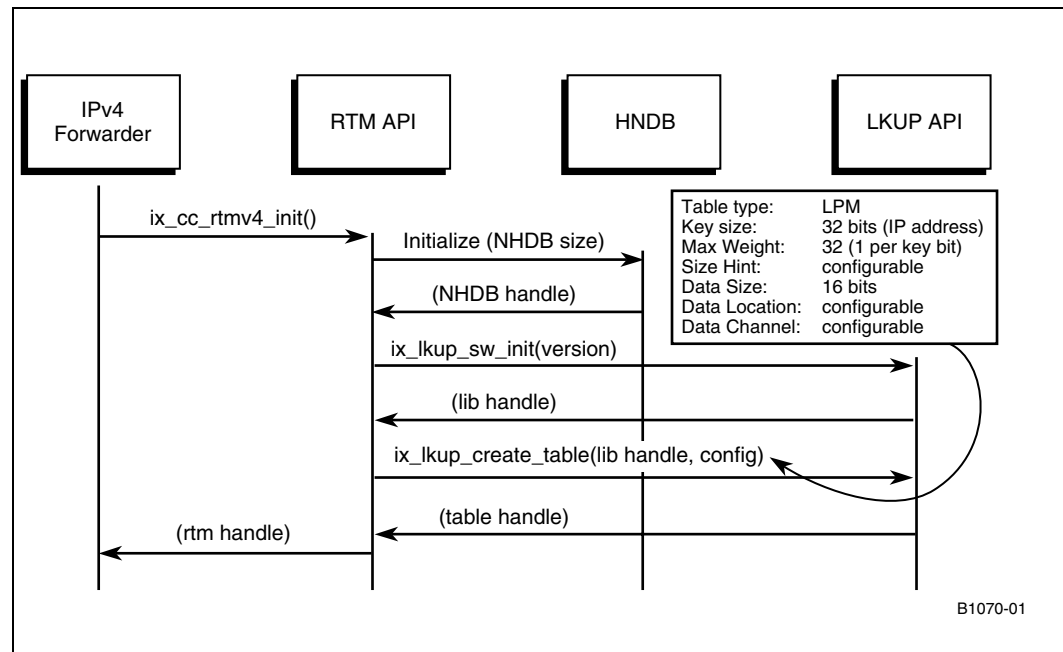


Figure 60-4 shows that the Route Table Manager uses an algorithm (Software LPM or TCAM) through the interface defined by the Lookup API. Internally the Route Table Manager stores Next Hop Information in a format microengines may access and use, and relies on the Next Hop Database to provide this functionality.

Entries in the NHDB are accessible via two different types of keys. Next Hop Identifiers (NHIDs) are used for maintaining the tables, but accesses into the NHDB using an NHID are not guaranteed to be fast. Therefore, the NHDB also supports indexes, and converts between NHIDs and indexes.

The IPv4 Forwarder Core Component starts the initialization by calling `ix_cc_rtmv4_init()`, as shown in Figure 60-5. The Route Table Manager API creates the lookup tables and next hop database according to the parameters provided.

Figure 60-5. Route Table Manager Initialization



The Route Table Manager uses the Lookup API to store a Next Hop index. This, combined with the lookup algorithm implemented by the Software LPM, satisfies the IPv4 Microblock's data lookup requirements.

60.5 External API

This section lists the Route table Manager core components. For complete details, see [Chapter 23, “Route Table Manager”](#) of the *IXA Software Building Blocks Reference Manual*.

60.5.1 Data Structures and Types

Table 60-1 summarizes the Data structures and type.

Table 60-1. Route Table Manager Data Structures and Types

Name	Description
<code>ix_cc_rtmv4</code>	Opaque handle to the Route Table Manager.
<code>ix_cc_rtmv4_nhid</code>	Data type representing an IPv4 Next Hop ID.
<code>ix_cc_rtmv4_next_hop_info</code>	Structure defining a next hop.
<code>ix_cc_rtmv4_symbols</code>	Structure containing values useful in the microengines.
<code>ix_cc_rtmv4_statistics</code>	Structure containing RTM statistics.
<code>ix_cc_rtmv4_lkup_type</code>	Enumeration of the supported TCAM and software algorithms.
<code>ix_cc_rtmv4_mem_type</code>	Enumeration of the supported memory types.
<code>ix_cc_rtmv4_config</code>	Data structure indicating how the Route Table Manager should be created.

60.5.2 Macros

Table 60-2 summarizes the Route Table Manager macros used for textual dumps during debugging. For complete details, see the *LXA Software Building Blocks Reference Manual*.

Table 60-2. Route Table Manager Macros

Name	Description
<code>IX_CC_RTMV4_DUMP_ROUTE_SIZE()</code>	Calculates the minimum size of a memory block for the Route Table—used for debugging only; returns a text string with the requested information.
<code>IX_CC_RTMV4_DUMP_NEXT_HOP_SIZE</code>	Calculates the minimum size of a memory block for the Next Hop Database—used for debugging only; returns a text string with the requested information.
<code>IX_CC_RTMV4_NHID_NO_ROUTE</code>	Returns a reserved next hop ID value.
<code>IX_CC_RTMV4_L2INDEX_NO_ROUTE</code>	Returns a reserved L2 index value.

60.5.3 Core Component Infrastructure API

The Route Table Manager external API is summarized in Table 60-3. For complete details, see the *LXA Software Building Blocks Reference Manual*.

Table 60-3. Route Table Manager API

Name	Description
<code>ix_cc_rtmv4_init()</code>	Creates a new Route Table Manager for the calling application.
<code>ix_cc_rtmv4_fini()</code>	Destroys an existing Route Table Manager, freeing all previously allocated resources.
<code>ix_cc_rtmv4_add_next_hop()</code>	Adds next hops to the Route Table Manager's next hop database.

Table 60-3. Route Table Manager API (Continued)

Name	Description
<code>ix_cc_rtmv4_delete_next_hop()</code>	Removes a next hop from the Route Table Manager provided that no routes currently refer to the next hop.
<code>ix_cc_rtmv4_update_next_hop()</code>	Updates the information contained in a next hop.
<code>ix_cc_rtmv4_get_next_hop()</code>	Retrieves a structure containing a copy of the information originally given through <code>ix_cc_rtmv4_add_next_hop()</code> .
<code>ix_cc_rtmv4_set_mtu()</code>	Updates the maximum transmission unit for a given next hop.
<code>ix_cc_rtmv4_set_flags()</code>	Updates the flags field for a given next hop.
<code>ix_cc_rtmv4_add_route()</code>	Adds routes, relating a range of network addresses to a next hop.
<code>ix_cc_rtmv4_update_route()</code>	Updates routes, relating an existing range of network addresses to a different next hop.
<code>ix_cc_rtmv4_delete_route()</code>	Deletes a route.
<code>ix_cc_rtmv4_lookup()</code>	Looks up routing information for a given IP address.
<code>ix_cc_rtmv4_dump_next_hops()</code>	Prints a list of all next hops to a block of memory—used for debugging purposes.
<code>ix_cc_rtmv4_dump_routes()</code>	Prints a list of all routes to a block of memory—used for debugging purposes.
<code>ix_cc_rtmv4_purge()</code>	Removes all routes and next hops.
<code>ix_cc_rtmv4_purge_routes()</code>	Removes all routes.
<code>ix_cc_rtmv4_get_symbols()</code>	Retrieves symbols to patch to the microengines.
<code>ix_cc_rtmv4_get_statistics()</code>	Retrieves statistics of the Route Table Manager.

Route Table Manager for IPV6

Core Component

61

61.1 Overview

The Route Table Manager (RTMv6) stores and retrieves data on behalf of the IPv6 Forwarder Core Component. In addition, the RTMv6 stores the data in a format consistent with the requirements of the IPv6 Forwarder microblock. RTMv6 uses the Lookup library API [LKUP] to access the route tables for IPv6. To manage next hops, RTMv6 uses a Next Hop Database (NHDB), as mentioned in Section Error! Reference source not found.. The RTMv6 exposes an API allowing the IPv6 Core Component to add to, remove from, or lookup items in the route table. This API is implemented as a library supporting a single client.

This implementation of RTMv6 is specific to the IP, version 6. Because an RTMv6 designed for IPv6 would require a different interface (for example, 128-bit parameters instead of 32-bit), it exposes its entry points in such a way as to indicate that it only deals with IPv6 data types and algorithms. This avoids naming conflicts with an RTMv6 designed for IPv4. For details on the RTM for IPv4 refer to [Chapter 60, “Route Table Manager”](#). For external APIs see [Chapter 24, “Route Table Manager for IPV6”](#) of the *IXA Software Building Blocks Reference Manual*.

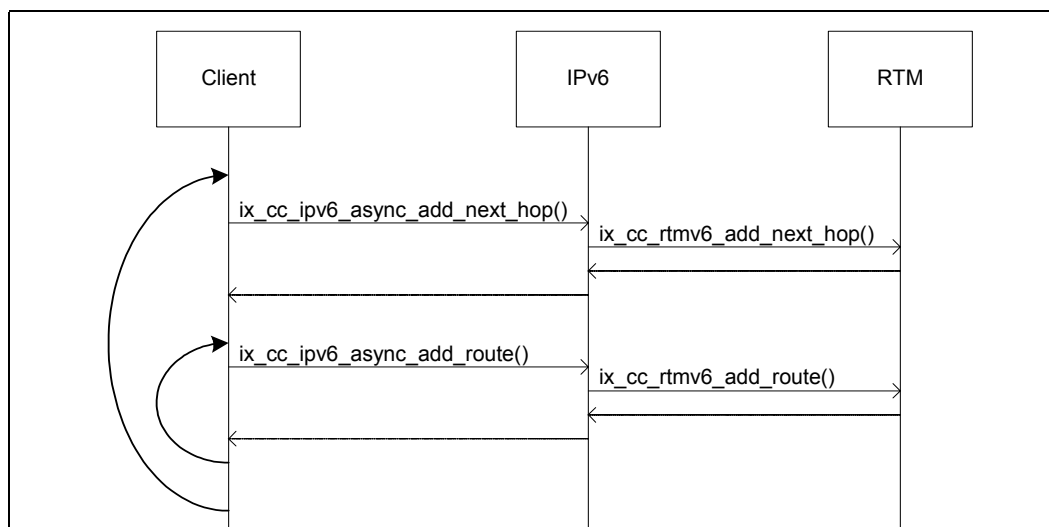
61.2 Usage Model

This section describes how the IPv6 Forwarder uses the RTMv6.

61.2.1 Using Routes and Next Hops

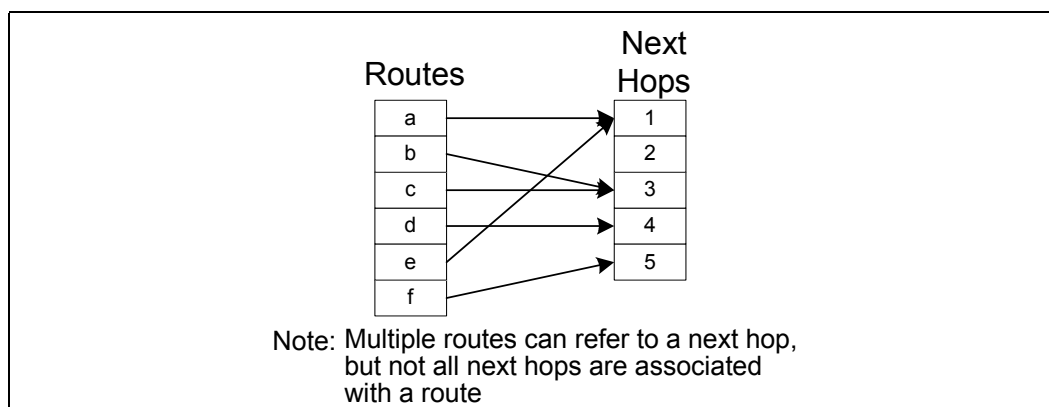
The IPv6 Forwarder uses the RTMv6 to store and retrieve next hops and routes. [Figure 61-1](#) illustrate the calling application that stores at least one next hop before adding routes—routes refer to next hops. A typical calling application of the IPv6 Forwarder may add one next hop and zero or more routes referring to that next hop. It may then repeat the cycle with more next hops and routes.

Figure 61-1. IPv6 RTM—Adding Information



A route may refer to any existing next hop, and any number of routes may refer to the same next hop. The RTMv6 does not require that a next hop is referenced by a route, but a route must reference a next hop. [Figure 61-2](#) illustrates this relationship.

Figure 61-2. Routes and Next Hops—IPV6 RTM



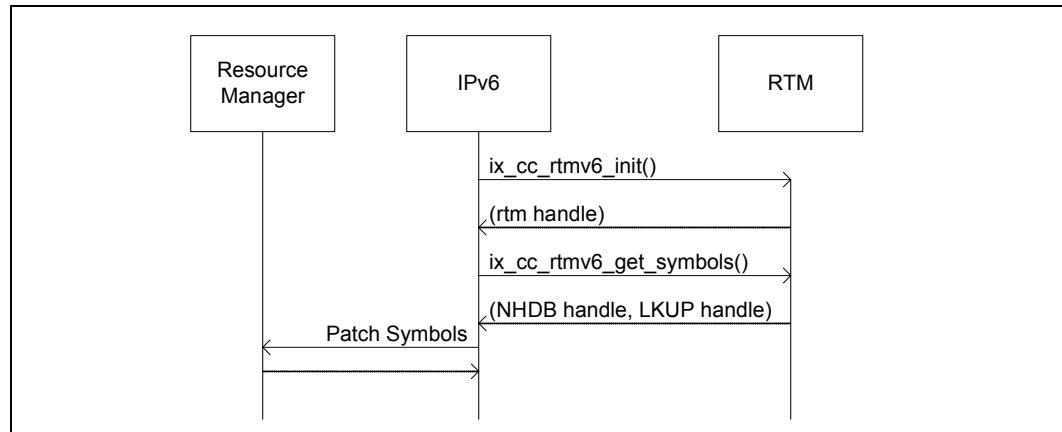
In addition to the routes and next hops previously described, the RTMv6 provides optional support for a default route. A default route is useful when none of the existing routes in the route table match the destination address of the packet. In this case, the packet is sent to the next hop specified by the default route. The alternatives to having a default route are to raise an exception (route lookup failed), or drop the packet. The client may enable the default route feature by adding a route referring to an existing next hop, or may disable the feature by deleting the default route.

Similarly, the RTMv6 places restrictions on when next hops may be removed. Just as a next hop must be in place before a route (including the default route) uses it, the next hop must remain valid for as long as the route exists. Therefore, the client must remove any routes using a specific next hop before that next hop may be removed. For example, in Figure 35, if the client wanted to remove next hop #3, it must first remove routes “b” and “c”. However, since next hop #2 is not being referenced by any routes, it could be removed immediately.

61.2.2 Initialization

Figure 61-3 shows how a calling application initializes the RTMv6. The client is responsible for creating the RTMv6 and adding next hops and routes. It may then lookup next hop information based on an IP address. Refer to [Section 61.3.6, “Lookup Library Initialization”](#) on page 874 for entry point descriptions and parameters.

Figure 61-3. RTMv6 Initialization



61.2.3 Pre-assigned Next Hop Identifiers

The RTMv6 allows the client to assign Next Hop Identifiers (NHIDs), but an NHID with a value of `IX_CC_RTMV6_NHID_NO_ROUTE` has special meaning within the RTMv6. The client must not attempt to use this value when adding or deleting routes or next hops. The RTMv6 detects this condition and returns an error.

61.2.4 Default Route

A default route is useful when the lookup on an IPv6 address fails (lookup returns next hop ID 0). The client may setup a default route by setting the NHID 0 to point to a default route.

61.2.5 Support for equal cost next hops

The RTMv6 supports up to four equal-cost next hops. The client may add a route that points to up to four equal-cost next hops. Multiple next hops (for a single next hop ID) are added by creating an 'intermediate' next-hop entry in the database, which contains pointers to the offsets at which the next hops are located.

61.3 Design Criteria

Other modules (particularly the [Chapter 53, “IPv6 Forwarder Core Component”](#) and [Chapter 25, “IPv6 Forwarder Microblock”](#)) rely on the RTMv6 to provide a set of functionality. This section briefly describes their requirements. In addition, it calls out dependencies the RTMv6 has on other components.

61.3.1 Data Storage and Retrieval

The IPv6 Forwarder requires the RTMv6 to store and retrieve routes and next hops. In addition, the RTMv6 must use the data stored to determine which next hop is appropriate for a given IPv6 address. The RTMv6 uses a Longest Prefix Match algorithm to make this determination.

61.3.2 Guaranteed Table Validity

Microengines have an extremely limited number of cycles in which to process packets. To maintain performance microblocks do not perform error checking or synchronize with the Core Component. This implies several things.

- The Core Component must ensure the microengines always have access to valid data.
- The entire burden of synchronization falls on the Core Component.

In this case, the IPv6 Forwarder relies on the RTMv6 to handle these issues on its behalf.

Validity of data accessed by the microblock is guaranteed by adding and deleting routes and next hops in a specific order.

61.3.3 Multiple Table Support

The RTMv6 creates one route table and next hop table per instantiation (returned when `ix_cc_rtmv6_init()` is called). If the a client of the RTMv6 requires two separate route tables, it would achieve this by calling `ix_cc_rtmv6_init()` twice.

61.3.4 Control Plane Design

The design of the Control Plane PDK, which is itself based on the NPF API, affects the design of the RTMv6. Some examples include:

- The RTMv6 does not support a separate `set_default_route` operation. Default routes are added by calling `add_route`.
- Separating next hops from routes. The RTMv6 follows a model where next hops and routes are treated separately, allowing many-to-one relationships between routes and next hops.

61.3.5 High-level Route Table

It is anticipated that higher-level clients (i.e. the CP-PDK or its clients) maintain a complete list of Next Hops and Routes for various reasons. High-level clients may be required to use their own lists of routes and next hops in order to remove next hops. Refer to Section 0

61.3.6 Lookup Library Initialization

The RTMv6 relies on the Lookup Library to allow multiple calls to `ix_lkup_sw_init()` or `ix_lkup_team_init()`. Only the first call initializes the library, subsequent calls simply return a handle. This allows multiple clients of the library to exist (for example, RTMv6 and RTMv4) without coordinating between each other. This also allows a single RTMv6 to instantiate several tables without keeping the `lkup_handle` in a global location.

Further, the RTMv6 requires the Lookup Library to allow multiple clients, each with their own table(s). Multiple clients must not share a table, but two clients must be able to have their own tables without interfering.

61.3.7 Single Direct Client

RTMv6 supports a single client, the IPv6 Forwarder. The reasons are described in section 5.1.2. While the IPv6 Forwarder is the only client supported, and the RTMv6 has design restrictions requiring a single client at runtime, the implementation of the RTMv6 does not preclude other clients from using the RTMv6 in a different environment. This is a reference design, and Section 4.13 describes the RTMv6 within that reference design. However that is not to say that the RTMv6 is designed to only work with the current implementation of the IPv6 Forwarder Core Component.

61.3.8 Duplicate Routes

The RTMv6 does not support duplicate routes, that is, routes where the same network address / network mask pair refer to two different next hops. Attempting to add the same route twice results in an error. However, the IPv6 RTMv6 provides support for up to four equal-cost next hops.

61.4 External API

This section lists the Route Table Manager for IPv6 core components. For complete details, see [Chapter 24, “Route Table Manager for IPV6”](#) of the *IXA Software Building Blocks Reference Manual*.

61.4.1 Data Structures, Types, and Macros

Table 61-1 lists the RTMv6 data structures, types and macros.

Table 61-1. RTMv6 Data Structures, Types and Macros

Data Structures, Types, macros	Description
<code>ix_cc_rtmv6</code>	Opaque handle to the RTM
<code>ix_cc_rtmv6_nhid</code>	Typedef to an intrinsic data type; represents a Next Hop ID
<code>ix_cc_rtmv6_next_hop_info</code>	Describes a Next Hop
<code>ix_cc_rtmv6_symbols</code>	Defines a structure containing values useful in the microengines
<code>ix_cc_rtmv6_statistics</code>	Retrieves statistics about the RTM
<code>ix_cc_rtmv6_lkup_type</code>	Enumeration of the supported algorithm types, TCAM and Software
<code>ix_cc_rtmv6_mem_type</code>	Enumeration of the supported memory types, SRAM and SDRAM

Table 61-1. RTMv6 Data Structures, Types and Macros

Data Structures, Types, macros	Description
<code>IX_CC_RTMV6_DUMP_ROUTE_SIZE</code>	Calculates the minimum size of a memory block useful for dumping all the routes in the RTM.
<code>IX_CC_RTMV6_DUMP_NEXT_HOP_SIZE</code>	Calculates the minimum size of a memory block useful for dumping all the next hops in the RTM
<code>IX_CC_RTMV6_NHID_NO_ROUTE</code>	Provides a macro for a reserved Next Hop ID value

61.5 Core Component Infrastructure API

Table 61-2 lists the RTMv6 Core Component Infrastructure API.

Table 61-2. RTMv6 Core Component Infrastructure API

API	Description
<code>ix_cc_rtmv6_init()</code>	Creates a new RTMv6 for the calling application
<code>ix_cc_rtmv6_fini()</code>	Terminates an existing RTMv6 and frees all previously allocated resources
<code>ix_cc_rtmv6_add_next_hop()</code>	Adds Next Hops to the RTMv6's next hop database
<code>ix_cc_rtmv6_delete_next_hop()</code>	Removes a Next Hop from the RTMv6
<code>ix_cc_rtmv6_update_next_hop()</code>	Updates a Next Hop in the RTMv6's next hop database
<code>ix_cc_rtmv6_get_next_hop()</code>	Retrieves a structure containing a copy of the information originally given
<code>ix_cc_rtmv6_set_mtu()</code>	Updates the MTU for a given Next Hop
<code>ix_cc_rtmv6_set_flags()</code>	Updates the Flags field for a given Next Hop
<code>ix_cc_rtmv6_add_route()</code>	Adds routes, relating a range of network addresses to a Next Hop
<code>ix_cc_rtmv6_delete_route()</code>	Deletes a route
<code>ix_cc_rtmv6_update_route()</code>	Updates routes, relating a range of network addresses to a Next Hop
<code>ix_cc_rtmv6_lookup()</code>	Looks up routing information for a given IPv6 address
<code>ix_cc_rtmv6_dump_next_hops()</code>	Prints all Next Hops to a block of memory for debugging purposes
<code>ix_cc_rtmv6_dump_routes()</code>	Prints all routes to a block of memory for debugging purposes
<code>ix_cc_rtmv6_purge()</code>	Removes all routes and next hops
<code>ix_cc_rtmv6_purge_routes()</code>	Removes all routes
<code>ix_cc_rtmv6_get_symbols()</code>	Retrieves symbols to patch to the microengines
<code>ix_cc_rtmv6_get_statistics()</code>	Retrieves statistics of the RTMv6

62.1 Design Considerations

The L2 Table Manager provides an API to manage, add and delete entries in the L2 Table, which corresponds to the Route Table and Next Hop Database on the ingress side. See [Chapter 60, “Route Table Manager”](#) for details on the Next Hop Database.

The L2 Table is created as an array of data structures containing L2 headers for the next hops. The L2 Index that is assigned to the packet and stored in each packet's metadata is based on the route table lookup on the ingress side. The L2 Index is used to access the appropriate entry in the L2 Table.

The L2 Table Manager library is not part of Core Component Infrastructure and does not need to conform infrastructure specific APIs.

The L2 Table Manager provides services for multiple calling applications. It uses the registry to share the base address of the L2 Table between calling applications. The property is created in the registry when the library is started and the base address of the L2 table is stored as a value of the property. [Section 62.4, “Multi-Client Support” on page 880](#) provides details about the sharing of the startup state.

The size of the L2 Table is determined by the System Application (see [Chapter 40, “System Application”](#)). The maximum number of entries is:

$$(2^{16} \text{ entries}) * (\text{size of L2 structure}) = 64K * 32 \text{ byte} = \sim 2MB$$

The limitation of 64K is based on the L2 Index, which is 16-bits. The table may be initialized in either SRAM or DRAM.

An entry must be usable from the microblocks' perspective—either it is known valid or known invalid. An entry must not claim to be valid if it has some incorrect information. This requirement applies both on the macro level (for example, while waiting for an ARP response), as well as on the micro level (during a 128-bit write operation). See [Table 62.5, “Microblock Synchronization” on page 880](#) for more information.

For external APIs see [Chapter 25, “L2 Table Manager”](#) of the *IXA Software Building Blocks Reference Manual*.

62.2 L2 Table Entries

This table is shared between microblocks and the core. The size of the table depends on number of next hops for this particular blade. The maximum number of entries could be calculated from following equation: (Number of Ports on Blade) x (Max Number of Hosts per Subnet).

Table 62-1 shows the contents of each entry in the L2 Table.

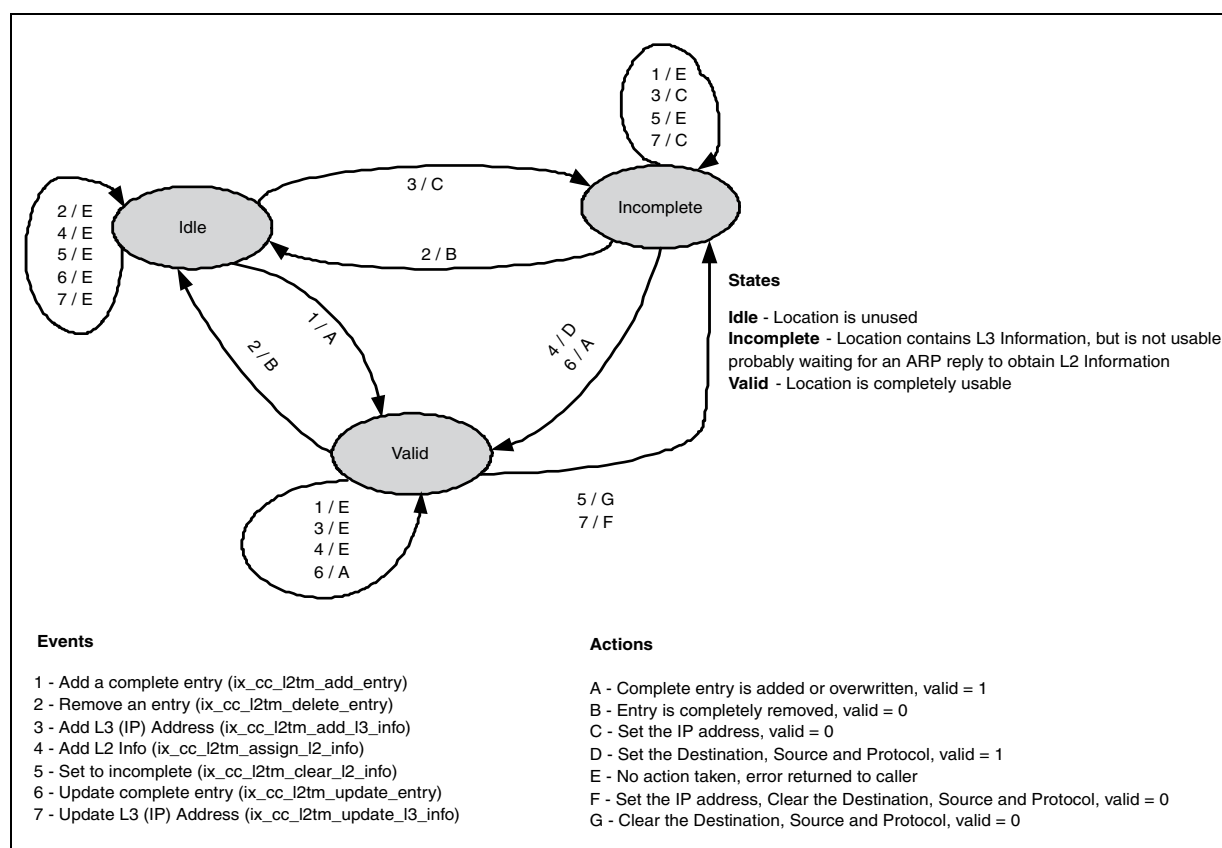
Table 62-1. L2 Table Entry

Field	Size (in bits)	Description
Flags	4	<ul style="list-style-type: none"> • Bit 31: Valid—The entry contains a valid L2 header • Bit 30: Send to Core—The microblock should send packets using this entry to the core • Bit 29: Reserved (zero) • Bit 28: Reserved (zero)
L2 Header	124	The contents of the L2 Header are determined by the microblock using that entry type.

The L2 Table Manager maintains related information for each entry in local memory, such as the IP address and other data required to maintain the table. Note that the L3 information (IP address) is not exported to the microblocks.

Figure 62-1 illustrates each entry in the L2 Table that cycles through states based on events.

Figure 62-1. L2 Table Manager Entry State Diagram



62.3 Data Flow and Behavior

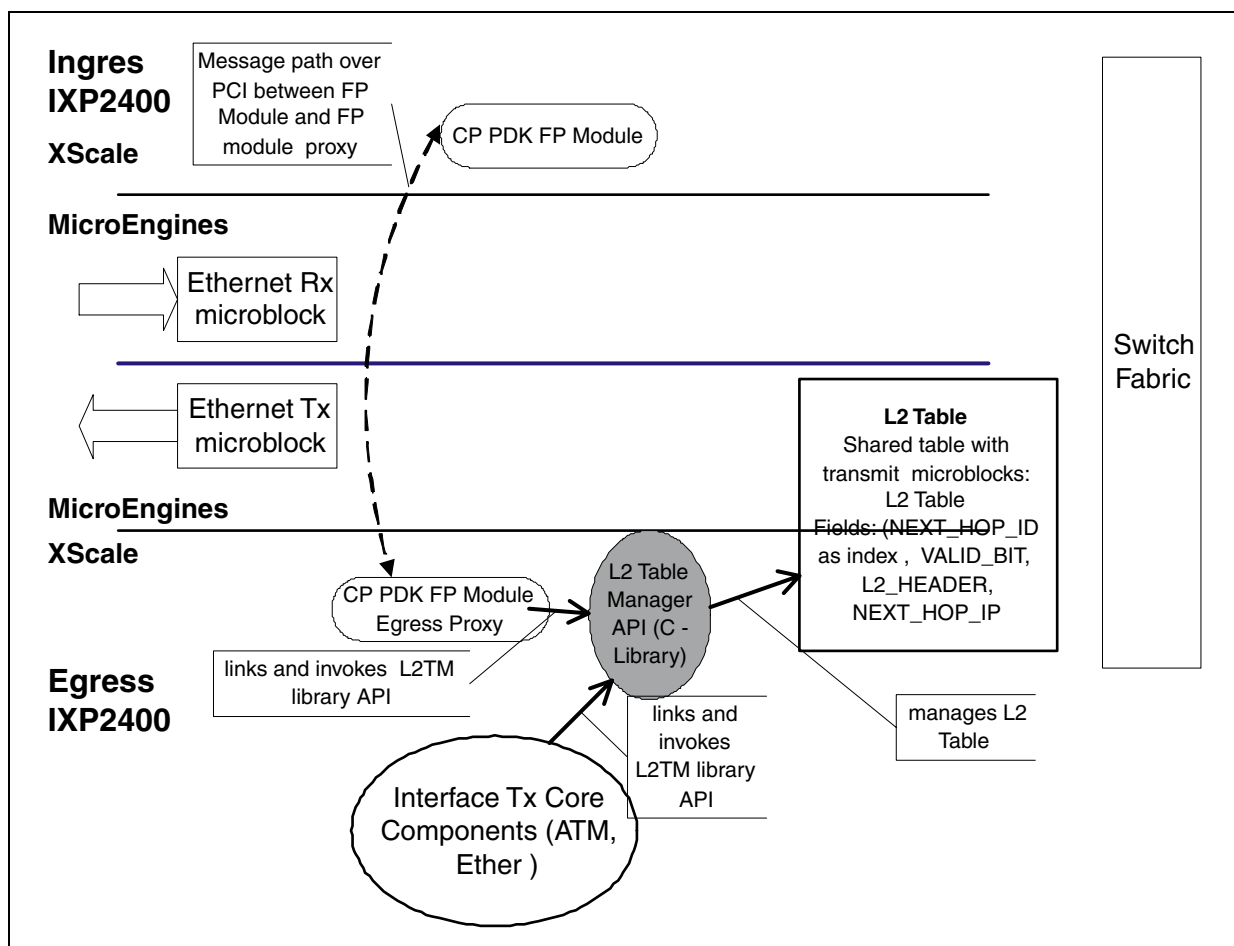
L2 Table is updated both by CP PDK NextHopId Manager and by ARP module of Ethernet Transmit Core Component.

The Control Plane PDK populates the L2 table through the FP module proxy running on the egress side. In addition, the ARP Module component updates the L2 Header field and Valid bit field of the table based on the ARP responses from the connected subnets.

The L2TM library is linked to the Interface Transmit (ATM, POS, Ethernet) core components and to FP module egress proxy. In the current system design, only one L2 table exists. Initialization of L2TM library is done when the first client (FP Module Proxy or one of the Interface core components) invokes `ix_cc_l2tm_create()`. The picture below shows the location of the L2TM library in the system. Upon initialization, the L2 Table is created and base address of the table is provided to the callers. The calling applications should not use this base address except for patching it to the microblocks. It is expected that only the ATM or Ethernet TX Core Components do the patching.

Figure 62-2 illustrates the L2 Table Manager in IXA SDK 3.1

Figure 62-2. L2 Table Manager in IXA SDK 3.1



62.4 Multi-Client Support

The L2TM library can have multiple clients. Several components on the egress side link to the L2TM library: Ethernet Tx, ATM TX and FP Module Egress proxy. Since these components must share the same physical table, a mechanism is provided to ensure that multiple clients may obtain a handle to a single table. See `ix_cc_l2tm_create()` for more information.

Because several clients may access the same table, it is important that the table does not become corrupted by several clients modifying it simultaneously. Also, a client looking up an item in the table should be assured the results are valid.

The current system assumes a non-preemptive environment. When the L2TM must operate in a preemptive environment, it uses a readers/writer lock to synchronize accesses to the table. The readers/writer lock allows many clients to read information as long as no clients are attempting to write information. It also prevents a writer from making changes until all readers have finished. This synchronization may be disabled at compile time to improve runtime performance.

62.5 Microblock Synchronization

To ensure microblocks always have accurate state about the contents of an L2 entry, the L2TM must invalidate each entry before changing the contents, whenever the changes span a long-word boundary. As an example, consider the case where a client calls `ix_cc_l2tm_update_entry()`. When writing the new values to shared memory, the L2TM must:

1. Clear the “valid” bit
2. Write the new data to shared memory
3. Set the “valid” bit

(These steps may be optimized on a case-by-case basis to minimize memory accesses and its impact on bandwidth).

Any microblocks accessing the L2 Table must do so using a single read request to retrieve all 128-bits of information. Doing so ensures the “valid” flag accurately represents the state of the entry.

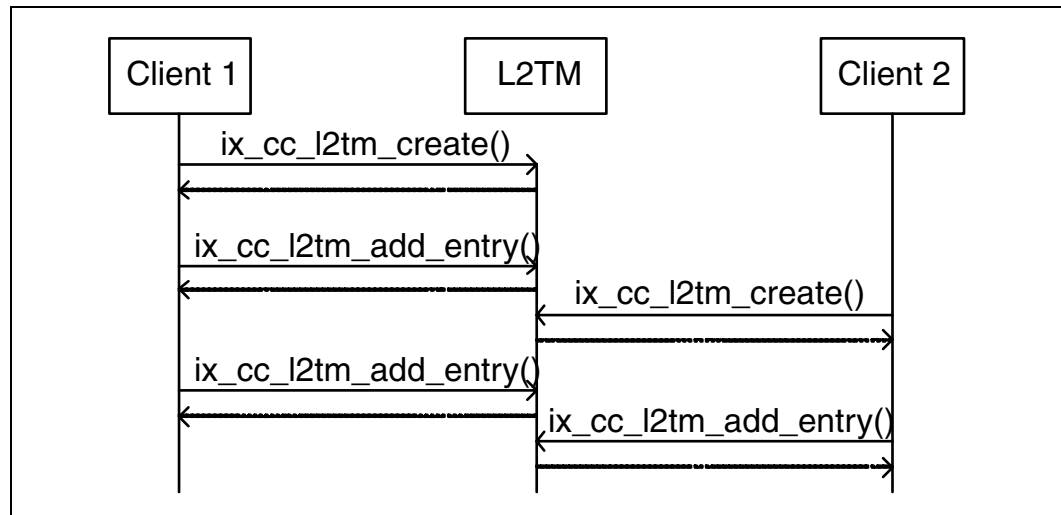
62.6 Initialization and Usage

The calling applications are responsible for initializing the L2TM library and adding L2 Headers and Next Hop Ip Addresses based on the L2 Index. Microblocks then can lookup L2 Destination by indexing into the table.

The size of L2 Table is determined during initialization. In the case of tables that are not shared among multiple clients, the size is provided as a configuration parameter to `ix_cc_l2tm_init()`; for shared tables, `ix_cc_l2tm_create()` retrieves the size from the registry. See `ix_cc_l2tm_create()` and `ix_cc_l2tm_init()` for more information.

Each client using the L2 Table Manager must initialize it. In the case that multiple clients want to use the same table manager, each client should call `ix_cc_l2tm_create()` to obtain a handle, and [Figure 62-3](#) illustrates this.

Figure 62-3. L2 Table Manager Usage



In cases where the registry does not exist, an alternate method is provided to create a table named “Default”.

62.7 Modularity

Internally, L2 Table Manager consists of following logical modules:

- Interface Module—provides interface to the outside clients
- Initialization Module—allocates data structures, handlers, and shares L2 Table base address
- L2 Table Module—internal interface to the L2 Table, packing of entry data structure, adds, deletes and looks up the entries.

62.8 External API

The L2 Table Manager is implemented as a library of functions providing services to the Interface TX core components (POS/ATM TX or Ethernet TX) and the FP Module Egress Proxy. For complete details, see [Chapter 25, “L2 Table Manager”](#) of the *IXA Software Building Blocks Reference Manual*.

62.8.1 Data Structures, Types, and Macros

Table 62-2 lists the data structures, types and macros defined in the L2 Table Manager.

Table 62-2. Data Structures, Types, Definitions and Enumerations in L2TM

Data Structures, Types and Definitions	Description
<code>ix_s_cc_l2tm_atm_header</code>	Describes an ATM entry in the L2 Table.
<code>ix_s_cc_l2tm_ether_header</code>	Describes an Ethernet entry in the L2 Table.
<code>ix_u_cc_l2tm_ipaddr</code>	Union containing an IP address.
<code>ix_s_cc_l2tm_entry</code>	Describes an entry in the L2 Table.
<code>ix_s_cc_l2tm_stats</code>	Statistics about the L2 Table Manager and its associated table.
<code>ix_s_cc_l2tm_symbols</code>	Contains values useful to microblocks.
<code>ix_s_cc_l2tm_config</code>	Configuration structure for creating a new L2 Table.
<code>ix_cc_l2tm</code>	Opaque handle to the L2Table Manager.
<code>ix_cc_l2tm_ipaddr_type</code> Enumeration	Indicates the type of the IP address to be stored.
<code>ix_cc_l2tm_l2addr_type</code> Enumeration	Indicates the type of the header to be stored.
<code>ix_e_cc_l2tm_error</code> Enumeration	Error codes specific to the L2 Table Manager.
<code>ix_cc_l2tm_memory_type</code> Enumeration	Types of memory supported for creating a new L2 Table.
<code>ix_cc_l2tm_state</code> Enumeration	Enumerations of the states each of the entries could occupy.

62.8.2 Library Functions

Table 62-3 shows the L2 Table Manager library functions..

Table 62-3. L2 Table Manager Library API

Functions	Description
<code>ix_cc_l2tm_create()</code>	Returns a handle to a named L2 Table Manager.
<code>ix_cc_l2tm_destroy()</code>	Destroys a named L2 Table Manager.
<code>ix_cc_l2tm_init()</code>	Creates a new L2TM for the calling application.
<code>ix_cc_l2tm_fini()</code>	Destroys an L2 Table Manager.
<code>ix_cc_l2tm_add_entry()</code>	Adds an L2 Entry to the L2TM's table database.
<code>ix_cc_l2tm_update_entry()</code>	Updates an L2 Entry to the L2TM's table database.
<code>ix_cc_l2tm_delete_entry()</code>	Removes an L2 Entry from the L2TM.

Table 62-3. L2 Table Manager Library API (Continued)

Functions	Description
<code>ix_cc_l2tm_get_entry()</code>	Gets an L2 Entry from the L2TM's table database.
<code>ix_cc_l2tm_add_l3_info()</code>	Adds L3 information into the L2 Table.
<code>ix_cc_l2tm_update_l3_info()</code>	Updates L3 information in the L2 Table.
<code>ix_cc_l2tm_assign_l2_info()</code>	Updates an L2 Entry with new information.
<code>ix_cc_l2tm_clear_l2_info()</code>	Clears the L2 Information from an entry from the L2TM.
<code>ix_cc_l2tm_get_l2_index()</code>	Gets the index of the entry containing a given IP address.
<code>ix_cc_l2tm_get_symbols()</code>	Gets symbols useful to the microblocks.
<code>ix_cc_l2tm_get_statistics()</code>	Gets statistics about the L2 Table.
<code>ix_cc_l2tm_purge()</code>	Remove all entries from the L2 Table.

Message Helper and Support Library 63

63.1 Overview

The Core Component Infrastructure provides a way to pass messages from one core component to another core component by using message handlers. A message can be anything - it is an array of bytes. An individual core components need to know how to interpret each message. The APIs exposed by the core components can be invoked through the messaging mechanism.

The Message Helper and Support Library is a translation layer to the actual Core Component Infrastructure function calls which simplifies calling the API from the clients of the core components.

For external APIs see [Chapter 26, “Message Helper and Support Library”](#) of the *IXA Software Building Blocks Reference Manual*.

63.2 Assumptions

- The reader should be familiar with the Core Components Infrastructure (see *Intel[®] Internet Exchange Architecture Portability Framework Developer's Manual*).
- The Message Helper API is independent of which network processor it is running on; the implementation for the IXP2400 and IXP2800 are the same.
- The overview is independent of XScale Operating System; The Message Helper API is identical on Linux and VxWorks.

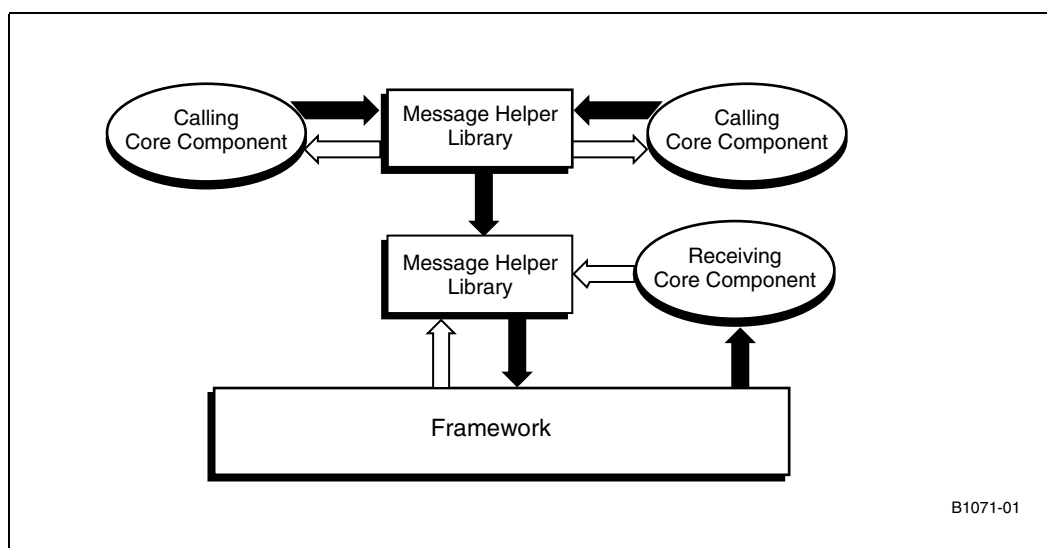
63.3 Overview

Descriptions of the operation of the Message Helper and Message Support libraries are in the context of a Core Component Infrastructure system. Generally, usage is the same for a system which only uses the Resource Manager. Details of a Resource Manager-only implementation are outlined in [Section 63.7, “Resource Manager Only System”](#) on page 897.

The APIs of core components are divided into two categories - API functions and Message Helpers. Message helpers simplify ways for clients of core components to call the API through the infrastructure messages. There is always a one-to-one mapping between APIs and Message Helpers.

API functions are regular C functions, such as `ipv4_route_add(route_desc* rdp)`. Each API function is defined by the core component - and is specific to the functionality of that core component.

Figure 63-1. Messaging Overview



The Message Helper has three primary functions.

1. Provide a simple interface to the APIs of each core component. This is accomplished by providing a set of functions that handle the invocation of the Core Component APIs. Code that uses these functions is independent of whether the system is operating on the Resource Manager or on the Core Component Infrastructure. The internal implementation of the Message Helper translates the call into either a message or a direct function call.
2. Provide a simple interface, which Core Component developers can use to implement their API calls. This interface handles the complexities of receiving return information from a call when messaging is used.
3. Allow the Core Component developer to write their Message Helper functions once in a way that is independent of whether the system is using messaging or direct calls. This helps the Core Components to be independent of the system configuration

The 'Message Helper' system is divided into two components, which are listed in [Table 63-1](#).

Table 63-1. Components of the Message Helper System

Component	Description
Message Helper Library	This library contains the 'Message Helper' functions developed for each core component. Each 'Message Helper' function translates its parameters and the required call semantic into a message sent to the 'Message Support' library.
Message Support Library	This library provides a group of support functions which take a message from the 'Message Helper' functions and applies the required call-semantic. Depending on the call type, state is stored internally to allow the core component's return message to be handled in the expected way. These support functions operate in the same way whether direct-calls or messages are used to perform the actual API call.

63.3.1 Message Helper Library

The Message Helper library contains all of the Message Helper functions for all of the core components that are running in the system. These functions are developed along with their corresponding core component.

The Core Components Message Helper Library (`ccmsghlp.lib`) is a placeholder for the implementation of Message Helper APIs for all core components. Each core component which is exposing the “C” API, needs to provide a Message Helper API. These functions are exported by the library. Each developer of core components needs to maintain the core component's specific functions in this centralized location.

Functions should be independent of each other and great care should be taken during implementation to avoid the following:

- circular dependencies (when one function depends on the second one, and second depends on third which is dependant on first)
- global variables
- optimizations based on other core components functions.

The prototype for a Message Helper function is up to the developer but must return an `ix_error` type. Each function must know the 'Communication ID' (COMID) to send the message to and the semantic to be applied to the call.

The COM ID is an ID used to determine where to send the message. In a Core Component Infrastructure Configuration, the message is simply sent to the actual Communication ID. In a Resource Manager only configuration, the COMID is used to look up a function pointer to the message handler, allowing a direct call to the same message handler function used in the Core Component Infrastructure.

There are three types of semantics that a message helper function can take. The type of call for each Message Helper can be different and should be noted in either the function name or the code documentation. The types are listed in [Table 63-2](#).

Table 63-2. Types of Calls Supported by the Message Helper

Call Type	Description
Fire-and-forget	This type of call sends a message to the core component and returns immediately. There is no return value or indication of the success/failure of the call. A message is sent and if an error occurs, the <code>IX_ERROR</code> data type is returned.
Fire-and-forget Broadcast	This type of call is fire-and-forget and is sent to a list of Communication IDs instead of just one. The primary purpose for this call type is for distributing the values of properties.
Asynchronous	An asynchronous call is defined as a call with a callback. After sending the message, the call returns immediately. If a transmission problem occurs, an <code>IX_ERROR</code> data type is returned. Later when the Core Component API has completed the call, a message is returned and the user-provided callback invoked.
Synchronous	Synchronous calls block until the message is received by the core component and a response sent back to the caller. This type of call can only be called by an application that is not a core component.

63.3.2 Support Library

In addition to the `ccmsgshlp.lib` library, a support library, `ccmsgsup.lib`, provides the semantics to messages sent by the Message Helper functions. These support functions sit atop the Core Component Infrastructure messaging functions and provides the following support services. In a Resource Manager-only system, the support library provides the same functionality using a direct call mechanism.

- Stores state for each Asynchronous and Synchronous call. Each synchronous or asynchronous call is given a unique call ID. The ID allows multiple calls to be made simultaneously. For example, two different core components could call the same Message Helper function at the same time. The handle to a return end-point is also provided so the reply can be sent back to the caller.
- Applies call semantics. For asynchronous calls, a pointer to the callback function and a user-provided context pointer are stored. When the reply arrives, the callback is invoked with the context. For synchronous calls, a semaphore is allocated and stored. The call is then blocked on the semaphore until the reply is received.
- Handle errors generated by the core component. If the Core Component API returns an `IX_ERROR`, it is sent along with the reply. For asynchronous calls, the error is then passed into the callback function along with the context. For synchronous calls, the `IX_ERROR` is returned to the caller when the call is unblocked.

63.4 Usage

This section describes the usage of the Message Helper and Message Support libraries. The internal operations are described in [Section 63.5, “Message Support Library Internal Design.”](#) on page 891.

The usage of a fully implemented Message Helper system is described first. This is followed by details of how the Message Helper functions are implemented.

63.4.1 Client Usage

- Initialization:
 - Core Component Infrastructure

During the initialization process of each execution engine, the message support `init` function must be called. During this step the message support library creates state tables and registers a message handler function. Each execution engine has a single Communication ID used for all return messages. The returns for all messages sent from an execution engine arrive via this communication ID.
 - Resource Manager

In a Resource Manager only system, a single initialization call is required since there is only one thread of execution.
- Use:

In order to make API calls to a core component, a client simply calls the corresponding Message Helper function. Message Helper functions which send synchronous messages cannot be called from a core component because of the blocking nature of synchronous messages. Each core component does not have it's own thread; it runs within an execution

engine which provides the execution context. If a core component were to block on a synchronous call, the return message would never be received because of a deadlock.

- **Callbacks:**

The message support library only provides a generic callback function that passes the return message as a “void *”. Each core component has a specific callback type defined for each of their API calls. To support these specific callback functions, a generic callback function must exist internally to the Message Helper API for each call or group of calls. These callbacks, of type `ix_cc_msghlp_callback`, reads the `void *` return message and makes a call to the specific API function passing appropriate arguments.

63.4.1.1 Example

The following example demonstrates how these functions and internal callback are implemented inside the Message Helper API.

Message Helper Library

```

/*****NOTE*****/
/*this callback definition is a reprint of the user
/* callback definition for get freelist. This is just here for
/* reference.
    ix_error (*ix_cc_sa_cb_get_freelist)(
        ix_error arg_Result,
        void *arg_pContext,
        ix_buffer_free_list_handle freeListHandle);
/*****/

/* a context structure used for the internal callback to
store user callback information*/
typedef struct _s_ix_cc_get_freelist_context{
    ix_cc_sa_cb_get_freelist *user_cb;
    void *user_ctx;
}ix_cc_get_freelist_context;

/* The actual MESSAGE HELPER FUNCTION */
ix_cc_sa_async_get_freelist(ix_cc_sa_cb_get_freelist *arg_cb,
                           void *arg_pContext,
                           ix_uint8 arg_freelist){
    ix_cc_get_freelist_context *usr_ctx;
    void *msg;
    ix_uint32 msg_len;

    /*****/
    /* Here we would */
    /*create your message based on parameters*/
    /*****/

    /*Create a context structure to hold user callback information*/
    usr_ctx = malloc(sizeof(ix_cc_get_freelist_context));

    usr_ctx->user_cb    = arg_cb;

```

```

usr_ctx->user_ctx    = arg_pContext;

ix_cc_msup_send_async_msg(THE_COMMID,          /*communication id*/
                          _ix_cc_sa_cb_get_freelist, /*internal cb*/
                          usr_ctx,             /*our context*/
                          MYMESSAGE,           /*our message type*/
                          msg,                 /*the message*/
                          msg_len);            /*message len*/
}

/*****
 *the internal callback
 *type 'ix_cc_msghlp_callback'
 *****/
ix_error _ix_cc_sa_cb_get_freelist(ix_error arg_Result,
                                   void *arg_pContext,
                                   void *arg_msg,
                                   ix_uint32 arg_msg_len){

    ix_cc_get_freelist_context *ctx;

    ctx = (ix_cc_get_freelist_context*)arg_pContext;
    /****
     *Here we would
     *Pull data out of arg_msg, (freelist_handle)*
     ****/

    /*call users callback function
     'freelist_handle is what we extracted from arg_msg*/
    ctx->user_cb(arg_Result,
                 ctx->user_ctx,
                 freelist_handle);
}

```

63.4.2 Core Component Usage

A Message Helper function needs to perform several operations in order to implement a call to a Core Component API function.

- Verify the validity of the arguments. If an error occurs, an `ix_error` should be returned.
- Create a structure containing all of the arguments. If the call takes a structure as an argument, this step is not necessary.
- Call the appropriate Message Support function for the type of call. i.e. Synchronous, Asynchronous or Fire-and-forget.
- Process post call data. For Asynchronous and Fire-and-forget messages, this means simply return any `ix_error` returned from the Message Support function. For a synchronous call, a return message is available when the Message Support function returns, this data must be passed back to the caller of the Message Helper function.

- Register a message handler function for the COM ID defined to receive API call messages. This function is called whenever a message is sent via the Message Helper API.

63.5 Message Support Library Internal Design.

The implementation of the Message Support library is dependent on whether the system is configured to use the Core Component Infrastructure or is a Resource Manager only system.

63.5.1 Core Component Infrastructure

When operating in a Core Component Infrastructure system, the Message Support library is implemented using messages. State for the call must be stored so that reply messages can be handled correctly. Errors generated by the Core Component APIs must be encoded in the return messages so they can be returned to the caller.

The Message Support library needs to know the current execution engine handle as a context. This is retrieved using the following function:

63.5.1.1 Initialization

During the initialization of each execution engine, the Message Support `init` function is called. As an argument, a return communication ID is provided to this function. Each execution engine has its own communication id used for return traffic to the core components running within it.

A call-table is allocated for the execution engine. This call table is used to dispatch return messages generated in response to Core Component API calls. This state information is required to allow return messages to be delivered to the correct place.

After the call-table is initialized, the `init` function registers a message handler for the return communication id. A pointer to the call-table is provided as context and is passed each time the message handler function is called by the infrastructure.

During the initialization, a core component registers a message handler with the Message Support library. In a Core Component Infrastructure system, this translates directly into a communication id message handler registration. This allows the same registration call to the Message Support library to be used in a Resource Manager only system.

63.5.1.2 Shutdown

When an execution engine is destroyed, its `fini` function calls the `fini` function for the Message Support library. During this call, the call-table is freed.

At destruction time, the call-table could contain outstanding calls waiting for service. Because the message handler is not called again while in the destruction phase, returns for all outstanding calls are dispatched. These returns contain a specific error message indicating that the return was triggered in response to a shutdown. This step is necessary to avoid deadlocks and memory leaks.

When the `fini` function is called, a flag is set which causes any calls to the Message Support functions to return a shutdown error message. This prevents messages from being sent to core components, which would not be valid during this phase.

63.5.1.3 Call-Table

The call table and its structures are described as follows. This table is a static size. This may prove to be inadequate, in which case a dynamically growing table may be used. When the static table is full, call attempts return a FULL error. In this case, the caller needs to try again at a later time.

```
#define IX_MSUP_MAX_MSG 32
enum ix_e_msup_calltype{
    IX_MSUP_CALLTYPE_FF,
    IX_MSUP_CALLTYPE_SYNC,
    IX_MSUP_CALLTYPE_ASYNC,
    IX_MSUP_CALLTYPE_LAST
}

struct ix_s_msup_call{
    ix_uint8    used;
    ix_e_msup_calltype calltype;

    /* Used for Asynchronous Only */
    CALLBACK    cb;
    void        *context;

    /* Used for Synchronous Only */
    SEMAPHORE    sema;
    void        *retmsg;
    ix_uint32    retmsg_len;
    ix_error     ret_error;
}
typedef struct ix_s_msup_call ix_msup_call;

struct ix_s_msup_call_table{
    COMID    return_id;
    ix_uint8 calls_valid;

    ix_uint32 free_call_hint;
    ix_msup_call calls[IX_MSUP_MAX_MSG];
};
typedef struct ix_s_msup_call_table ix_msup_call_table;
```

The above code is discussed in reverse order, starting with `ix_msup_call_table` and ending with `ix_msup_call`.

`ix_msup_call_table_s`

The call-table contains a fixed array of `ix_msup_call` structures. The size of the array is defined by `IX_MSUP_MAX_MSG`. This size represents the number of simultaneous outstanding calls. This is done to eliminate the repeated allocation of call structures. In most cases, the size of this array can be set relatively small. Under normal coding situations, a core component probably needs to have only a few outstanding asynchronous calls at a time. This value can be adjusted to meet the needs of the system design. If there are many core components in each execution engine or many

asynchronous calls need to be made simultaneously, this number can be increased. If the core components only have one outstanding call at a time, this number could be set to be equal to the maximum number of core components in an execution engine.

Table 63-3 describes the members of this structure.

Table 63-3. ix_msup_call_table_s Structure Members

Member	Description
return_id	This is the COM ID of the return path for the execution engine. When a message is sent, this ID is sent inside the message so the return message can be sent.
calls_valid	This flag is used during shutdown to disable new messages from being sent. It is checked each time one of the message support functions is called.
free_call_hint	This variable is an optimization. When a call return is complete and the call structure is returned to the free state, it's index is stored in this variable. This allows the next call to grab a known free call structure instead of looking for one in the array.
calls	The array of call structures used to store the state of synchronous and asynchronous calls.

ix_msup_call_s

An instance of this structure is used to store state for a call. Some members of this structure are only used for asynchronous calls and some are used only for synchronous calls.

Table 63-4. ix_msup_call_s Structure Members - Asynchronous and Synchronous

Member	Description
used	This flag is set when the instance of the structure is actively holding state for an outstanding call.
calltype	This flag designates whether the call is asynchronous or synchronous. The type of call determines what members of the structure are valid and how the return message is to be handled.

Table 63-5 lists the members used only for asynchronous call types.

Table 63-5. ix_msup_call_s Structure Members - Asynchronous only

Member	Description
cb	A pointer to the callback function to be called when the return message arrives.
context	This is the caller provided context passed when the original message was sent. It is passed to the callback function.

Table 63-6 lists the members used only for Synchronous call types.

Table 63-6. ix_msup_call_s Structure Members - Synchronous only

Member	Description
sema	The semaphore on which the original call is pending. This semaphore is cleared when the return message arrives and it's data is stored in the current call structure.

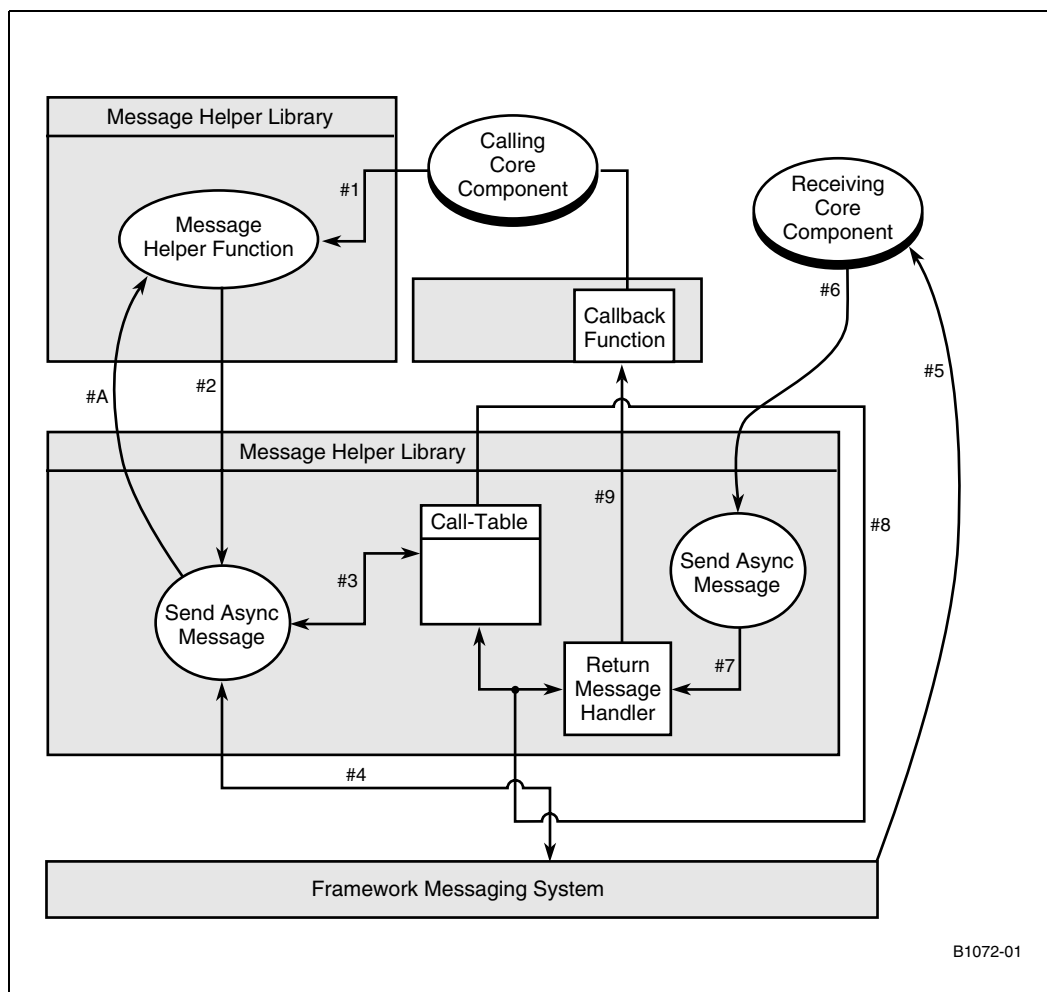
Table 63-6. ix_msup_call_s Structure Members - Synchronous only

Member	Description
retmsg	Pointer to the return message. This is stored here so it can be retrieved by the original caller when the semaphore is cleared.
retmsg_len	Length of the return message. Also stored for the original caller.
ret_error	If an error is stored along with the return message it is stored here to be returned to the calling function when the semaphore is cleared.

63.6 Data Flow

63.6.1 Asynchronous Call: Data Flow

Figure 63-2. Asynchronous Call: Data Flow

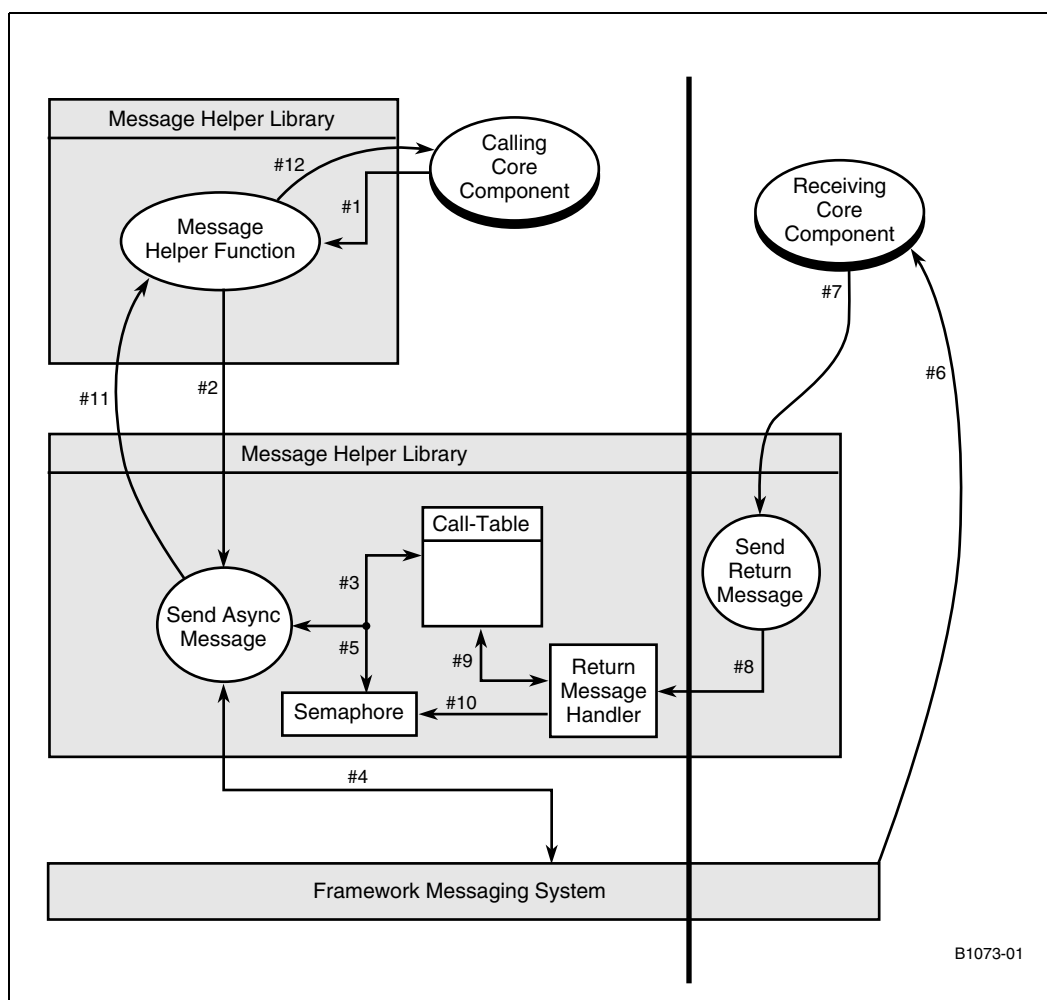


The following is a high-level examination of the data flow for an asynchronous call. `mycomp` and `myapi` are used as an example. The numbers below match the steps in the above diagram.

1. A client calls the `ix_cc_async_mycomp_myapi` Message Helper function that resides in the Message Helper library. Data, a callback and context are passed as arguments.
2. The Message Helper function (`cc_mycomp_myapi_async`) stores the data in a structure. The Message Helper function calls the asynchronous send function of the Message Support library, passing the callback, context and message.
3. The Message Support library selects a free call structure from the call-table and stores the callback and context. The user message is stored in an `ix_buffer` along with the communication ID to send the return message (this is the single communication ID for the execution engine).
4. The Message Support library sends the message via the infrastructure and returns (#A).
5. The Message Handler for the destination core component receives the message and the Core Component API is called.
6. The Core Component API does its work and calls the return function of the Message Support library.
7. The return message is then sent to the execution engine return communication ID.
8. The message handler function inside the Message Support library receives the return message and looks up the call structure in the call-table.
9. The user provided callback is called.

63.6.2 Synchronous Call: Data Flow

Figure 63-3. Synchronous Call: Data Flow



The following is a high-level examination of the data flow for a synchronous call. `mycomp` and `myapi` are used as an example.

1. A client calls the `cc_mycomp_myapi_sync` Message Helper function that resides in the Message Helper library. Data and a location to store the reply are passed as arguments.
2. The Message Helper function (`cc_mycomp_myapi_sync`) stores the data in a structure. The synchronous send function of the Message Support library is called, and passed the message.
3. The Message Support library selects a free call structure from the call-table and a semaphore is allocated. The user message is stored in an `ix_buffer` along with the communication ID to send the return message to. (this is the single communication ID for the execution engine).
4. The Message Support library sends the message via the infrastructure.
5. The call blocks on the semaphore.

6. The Message Handler for the destination core component receives the message and the Core Component API is called.
7. The Core Component API does its work and calls the return function of the Message Support library.
8. The return message is then sent to the execution engine return communication ID.
9. The message handler function inside the Message Support library receives the return message and looks up the call structure in the call-table. The return message and any errors are stored in the call structure.
10. The semaphore is cleared.
11. The original call that is blocked on the semaphore continues. The return message and error are read from the call structure and returned.
12. The message helper function returns, passing the return message and error information back to the original caller.

Note: Synchronous calls are prohibited from within a core component or from any client in case of single threaded system.

63.7 Resource Manager Only System

When operating in a Resource Manager only system, the Message Support library is implemented using direct calls. The Message Support API remains the same but reuses the communication IDs as an index into a predefined table. The table contains pointers to the message handler functions.

Certain assumptions about the environment have to be made. A developer can alter the Message Support library to allow it to operate under different conditions. The following assumptions are made about the operating environment in a Resource Manager only system.

1. All software sending or receiving calls must exist within the same thread. This is because it is important for a function to know exactly what context it is being called from. If this function is written in such a way as to allow for the problems of being executed by many threads, then this requirement does not apply.
2. All software sending or receiving calls must be in the same address space.

63.7.1 Initialization

In a Resource Manager only system, the `init` function of the Message Support library is called only once.

63.7.2 Internal Operation

Instead of sending messages when a call is made, the core component's message handler function is called directly. The communication ID passed into each message function is used to look up the correct destination message handler.

63.7.3 Asynchronous and Fire and Forget

When the Fire-and-forget Message Support function is called, the destination message handler is called and no information is returned to the caller.

In the case of the asynchronous Message Support function, the destination message handler is called. The specified callback function is then called passing the return message, context and errors. The call then returns.

63.7.4 Synchronous

When the synchronous Message Support function is called, the destination message handler is called and the return message and error are returned.

63.8 Message Support Library API

Several support functions are provided to enable the Message Helper functions. The `init` and `fini` functions are called by each execution engine and allow for the setup/tear-down of the support system. There are also three functions to allow messages to be sent in each of the defined ways.

Table 63-7 summarizes the functions in the Message Support Library API. For complete API details, see [Chapter 26, “Message Helper and Support Library”](#) of the *IXA Software Building Blocks Reference Manual*.

Table 63-7. Message Support Library API

API	Description
<code>ix_cc_msup_init()</code>	Initializes the message support library.
<code>ix_cc_msup_fini()</code>	Shuts down the message support library.
<code>ix_cc_msup_send_msg()</code>	Sends a fire-and-forget message.
<code>ix_cc_msup_send_async_msg()</code>	Sends an asynchronous message and invokes the callback function when the return message is received.
<code>ix_cc_msup_send_bcast_msg()</code>	Sends a fire-and-forget message.
<code>ix_cc_msup_send_sync_msg()</code>	Sends a synchronous message.
<code>IX_MSUP_EXTRACT_MSG()</code>	Converts the message passed by the framework (<code>msg</code>) to the message sent by the message helper function (<code>mymsg</code>).
<code>ix_cc_msup_send_reply_msg()</code>	Sends a reply message to the caller for asynchronous and synchronous call types.

Stack Driver Component

The Stack Driver includes the following core component:

- [Chapter 64, “Stack Driver”](#)

In SDK 3.1, the Stack Driver core component supports the IPv4 pipeline and does not support IPv6. Stack driver has been integrated with local TCP/IP stack for IPv4 pipeline running on VxWorks but not on Linux.

The Stack Driver provides a core component compatible abstraction of the operating system's network stack. It is a specialized type of core component that allows I/O to and from any network stack and other core components. It appears to the network stack as a media driver and appears to the IXA SDK as a core component. The main functions of the Stack Driver are:

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

The operating system's network stack with which the Stack Driver interacts can be either local or remote.

64.1 Overview

This chapter describes the design of the Stack Driver core component for the IXP2400 and IXP2800 network processors. The Stack Driver provides an interface between the IXA SDK and the local TCP/IP stack. The Stack Driver also supports packet transmission and basic interface configuration between the IXA SDK and the Control Plane PDK (CP PDK). However, it does not handle updates of forwarding tables.

This chapter focuses on an overview of the stack driver's high-level design but provides some low-level details, especially in the VxWorks-specific sections, as some of the components of the stack driver (i.e. local VIDD) are OS-specific.

For external APIs see [Chapter 27, “Stack Driver”](#) of the *IXA Software Building Blocks Reference Manual*.

64.2 Assumptions and Dependencies

64.2.1 Assumptions

- The core component infrastructure allows a user to specify multiple outputs for a single core component. However, this implementation of the Stack Driver uses only one output.
- This implementation of the Stack Driver deals only with IP packets.

64.2.2 Dependencies

- The Stack Driver depends on the availability of the Intel® Internet Exchange Architecture Portability Framework described in the *Intel® Internet Exchange Architecture Portability Framework Developer's Manual*.
- This implementation depends upon the availability of the Core Component infrastructure described in the *Intel® Internet Exchange Architecture Portability Framework Developer's Manual*, but this design is layered so that most of the code can be reused if the Core Component Infrastructure is not available or replaced with a customer-defined layer. Separation between these dependencies for this design is described in [Section 64.3.3.3, “Core Component Infrastructure Separation”](#) on page 907.

64.3 Stack Driver

64.3.1 Stack Driver Design

The Stack Driver consists two parts:

- A Core Component Module
- One or more communications handlers. An example of a communication handler is a local network driver (VIDD) or a “transport” module.

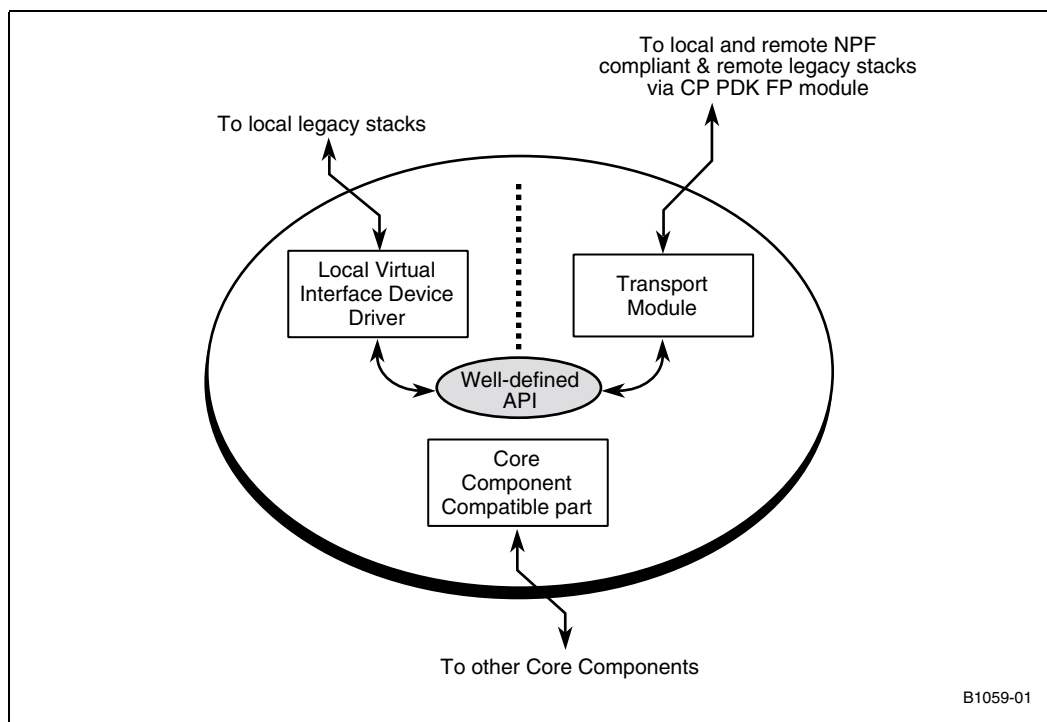
The Core Component Module is a conventional core component (meaning, running on a general-purpose processor) that usually runs in the same address space with the communication handler. Depending on the operating system and capabilities of the framework it may have to run in a different address space.

To the TCP/IP stack, the driver appears as a standard (operating system) network (interface) driver except that it only indirectly interacts with the hardware. The driver implements a set of functions to manage the interfaces.

In addition, the Stack Driver provides a transport module to send packets to and receive packets from a remote TCP/IP stack. It exports the same or similar set of functions as the Local VIDD.

Figure 64-1 shows the basic packet flow between the core component, local VIDD, and transport module of the Stack Driver to send packets to and from local and remote stacks.

Figure 64-1. Stack Driver Packet Flow



The Stack Driver exports a set of functions to initialize and configure each interface it supports. It also exports a set of functions for synchronization of dynamic property updates.

The Stack Driver can have multiple outputs that can be bound to any downstream core component. All outgoing packets are sent to these outputs. In this release, the stack driver supports classification between IPv4 and IPv6 outgoing packets under Linux* only. A stub is used in place of a classification function under VxWorks* and in this case, all outgoing packets are sent to the IPv4 Forwarder Core Component.

64.3.1.1 Stack Driver Design

The Stack Driver architecture separates the OS-dependent parts of the Stack Driver from its core component-dependent parts. Only the core component portion of the Stack Driver knows about Stack Driver inputs and outputs. Communication handlers (Local VIDD and Transport Module) of the Stack Driver are independent of the Core Component Infrastructure.

The Core Component Module of the Stack Driver does not know about interactions between the OS and network stack. Only the Local VIDD and Transport Module are responsible for interactions with the OS, the network stack, and the remote host's interconnect mechanism.

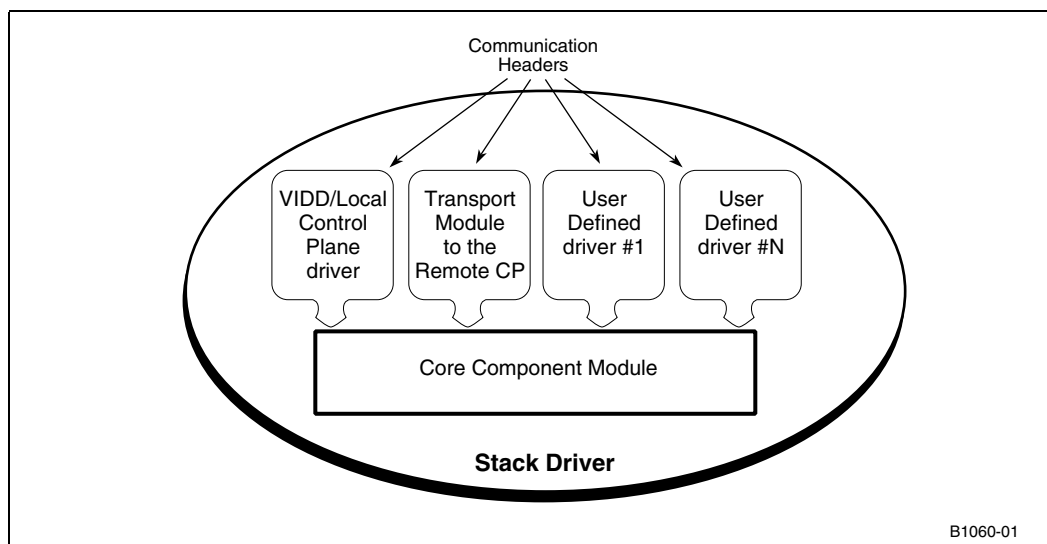
The core component-dependent part of the Stack Driver (Core Component Module) is an interface between those drivers and the IXA SDK infrastructure. It binds to other core components, passes `ix_buffer` handles, and handles SDK messages. In addition, the Core Component Module can decide which communication handler the packets should go to by performing a classification on incoming packets. The Core Component Module of the Stack Driver is described in more detail in [Section 64.3.2, “Design of the Core Component Module” on page 905](#).

The OS-dependent part of the Stack Driver is usually a data link device driver. This part of the Stack Driver must conform to OS-specific driver registration and APIs. One instance of such a driver can deliver packets to the local Control Plane Stacks, another instance to the remote CP Stacks. The OS-dependent part, in this case the local VIDD, is described in [Section 64.3.6, “VIDD for VxWorks” on page 914](#).

The design for the Transport Module is described in [Section 64.3.10, “Transport Module Design” on page 920](#).

The current design allows a number of OS-dependent device modules to be registered with the Core Component Module. [Figure 64-2](#) illustrates the internal design of the Stack Driver.

Figure 64-2. Stack Driver Modularity



64.3.1.2 Packet Flow

The Core Component Module receives incoming packets and hands them over either to the Local VIDD or to the Transport Module, which in turn sends the packet up their respective TCP/IP stack. Outgoing packets (from TCP/IP stack) are received by the driver or Transport Module and handed over to the Core Component Module, which then sends the packet over its output.

Most of the time, either the local or remote stack is used. By default, the local stack support is enabled. In order to enable the remote stack, set the `REMOTE_STACK_FLAG` as 1 in the system registry, if the registry support exists. If the registry support does not exist, then set `IX_CC_STKDRV_REMOTE_SUPPORT_FLAG` to 1.

However, because of this requirement the Core Component Module may have to do some classification of the incoming packets to decide whether to send the packets to the Local VIDD or to the Transport Module. This classification can be compiled out, however, if a system designer wants the Stack Driver to send all packets to the local stacks, or to send all packets to the remote stacks. The design of the Packet Classifier is described in [Section 64.3.4, “Packet Classifier Design”](#) on page 909.

64.3.1.3 Synchronizing Properties

The Receive (RX) and Transmit (TX) Core Components and the Stack Driver may share a set of properties, such as IP address and mask, MTU, and MAC Address, for each of physical interfaces.

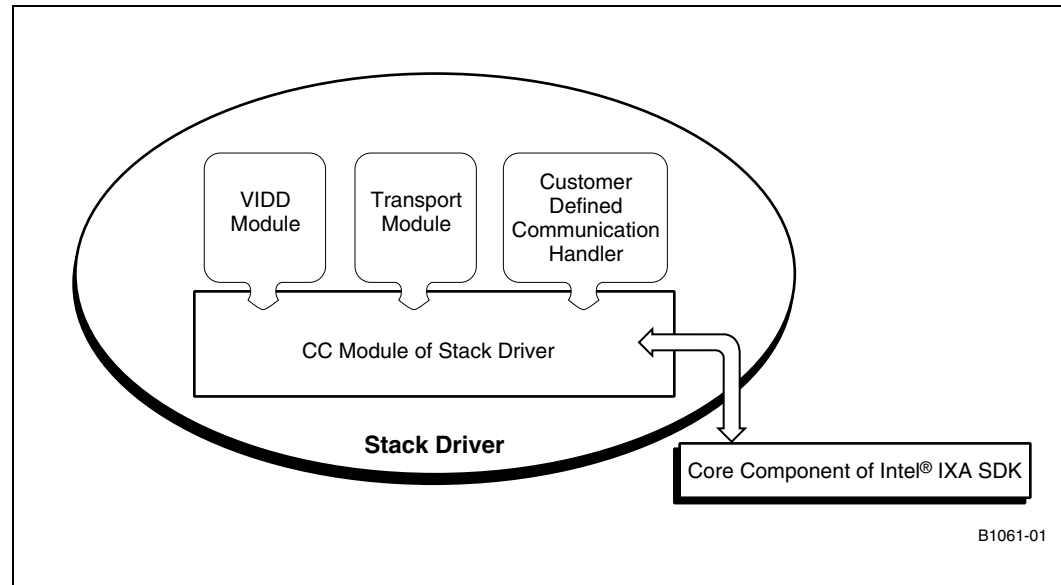
Property synchronization uses messaging as described in the [Chapter 39, “Core Components Overview.”](#) Dynamic properties are maintained by Master and Client core components, where the Master is responsible for propagating specific property updates to the Client(s).

These functions are described in [Section 64.3.3.7, “Properties API”](#) on page 909.

64.3.2 Design of the Core Component Module

Figure 64-3 shows the modularity of the Stack Driver.

Figure 64-3. Stack Driver Internal Modularity



64.3.2.1 Core Component Module Design

The Core Component Module interfaces with other IXA core components, such as the IPv4 Forwarder Core Component, IPv6 Core Component, and the Queue Manager Core Component. It performs the following actions:

- Receive and send packet data
- Receive and send control and configuration messages
- Optionally classify packets for delivery to the local VIDD or the Transport Module.

For receiving and sending packet data, the Core Component Module does the following:

- Passes `ix_buffers` to and from a registered handler. The handler could be any of the following:
 - VIDD - local VxWorks or Linux device driver.
 - Transport Module - driver responsible for communicating with the remote Control Plane (CP).
 - Customer defined handler - the module provided by the user of the IXA SDK for special handling of the packets.

Currently, a combination of the Core Component Module and local VIDD on the VxWorks or Linux OS is used. The packet handler is described in [Section 64.3.3.2, “Core Component Infrastructure API” on page 907](#). Packet flow is described in [Section 64.3.8, “Packet Processing—VxWorks Example” on page 917](#).

For receiving and sending control and configuration messages, the Core Component does the following:

- Passes control and configuration messages from the IXA SDK to the CP stacks and from the CP stacks to the IXA SDK. Examples of control messages are:
 - IXP (logical) port addition and deletion
 - Changes in the system.

For example, if one of the ports is brought down, then the Core Component Module forwards that message to the upper registered handler. If the interface is brought down by a CP stack, the changes need to be propagated into the IXA SDK RX and TX core components. Control and configuration messages can be integer values or strings.

The Core Component Module can optionally classify the packet and messages based on an object classification token. The token could be a string, an integer id, or a combination of both. For PDU and control messages traveling up to the stack, the classifications are performed in the Core Component Module. For this release only a Packet Classifier is implemented as part of the Core Component Module to determine which registered communication handler the packets should go to on their way up the stack. The Packet Classifier design is described in [Section 64.3.4, “Packet Classifier Design” on page 909](#).

64.3.2.2 Execution Context

The Stack Driver Core Component Module and VIDD reside in kernel space for Linux.

Packet processing in the Stack Driver Core Component Module and VIDD for packets going up the stack run in the context of the execution engine for the Stack Driver Core Component Module.

For packets going down the stack, packet processing in the Stack Driver Core Component Module and VIDD run in whatever context the task that sent the packet down is running in (e.g. socket program, etc.). Or, if the stack decided to fragment the packet, Stack Driver packet processing runs in the task context of the stack.

64.3.3 External API

This section summarizes the APIs for the Stack Driver core components. For complete API details, see [Chapter 27, “Stack Driver” of the IXA Software Building Blocks Reference Manual](#).

64.3.3.1 Core Component Module Data Structures and Types

Table 64-1 lists the data structures and types used by the Stack Driver Core Component Module. See [Section 27.1, “Core Component Module Data Structures and Types” on page 561](#).

Table 64-1. Stack Driver Core Component Data Structures and Types

Name	Description
<code>ix_cc_stkdrv_packet_type</code>	Used to label a packet's destination—either to the local legacy stacks, local NPF stacks, remote legacy stacks, or remote NPF stacks.
<code>ix_cc_stkdrv_handler_id</code>	Identifies the type of an OS-dependent packet handler.
<code>ix_cc_stkdrv_handler_func</code>	During initialization, each handler object populates this data structure with function addresses and corresponding contexts.
<code>ix_cc_stkdrv_virtual_if</code>	This structure represents a node in the linked list of available virtual interfaces.

Table 64-1. Stack Driver Core Component Data Structures and Types

Name	Description
<code>ix_cc_stkdrv_physical_if_info</code>	This structure represents physical interface properties.
<code>ix_cc_stkdrv_physical_if_node</code>	This structure represents a node in the link list of available physical interfaces.
<code>ix_cc_stkdrv_handler_module</code>	This data structure describes a registered communication handler.
<code>ix_cc_stkdrv_fp_node</code>	This is a linked list of forwarding plane information structures.
<code>ix_cc_stkdrv_ctrl</code>	The structure represents the control block for the Stack Driver core component.

64.3.3.2 Core Component Infrastructure API

The Stack Driver Core Component infrastructure API is summarized in [Table 64-2](#). See [Section 27.3.1, “Core Component Infrastructure API”](#) on page 572.

Table 64-2. Stack Driver Core Component Infrastructure API

Name	Description
<code>ix_cc_stkdrv_init()</code>	Initializes the core component.
<code>ix_cc_stkdrv_fini()</code>	The termination function used to free Stack Driver core component memory and data structures.
<code>ix_cc_stkdrv_high_priority_pkt_handler()</code>	Packet processing function for high-priority signaling packets.
<code>ix_cc_stkdrv_low_priority_pkt_handler()</code>	Packet processing function for low-priority data packets.
<code>ix_cc_stkdrv_pkt_to_remote_handler()</code>	Sends incoming packets to the Transport module without any classification.
<code>ix_cc_stkdrv_msg_handler()</code>	Message processing function for the stack driver.

64.3.3.3 Core Component Infrastructure Separation

This API abstracts the Core Component Infrastructure so that as much of the code for the Stack Driver Core Component Module can be reused in a system that does not use the Core Component Infrastructure. [Table 64-3](#) lists this API. See [Section 27.3.2, “Core Component Infrastructure Separation”](#) on page 578.

Table 64-3. Stack Driver Core Component Infrastructure Separation API

Name	Description
<code>_ix_cc_stkdrv_process_pkt()</code>	Processes a locally-destined packet.
<code>_ix_cc_stkdrv_process_pkt_to_remote()</code>	Sends a packet directly to the forwarding plane module for remote processing, forgoing any classification.

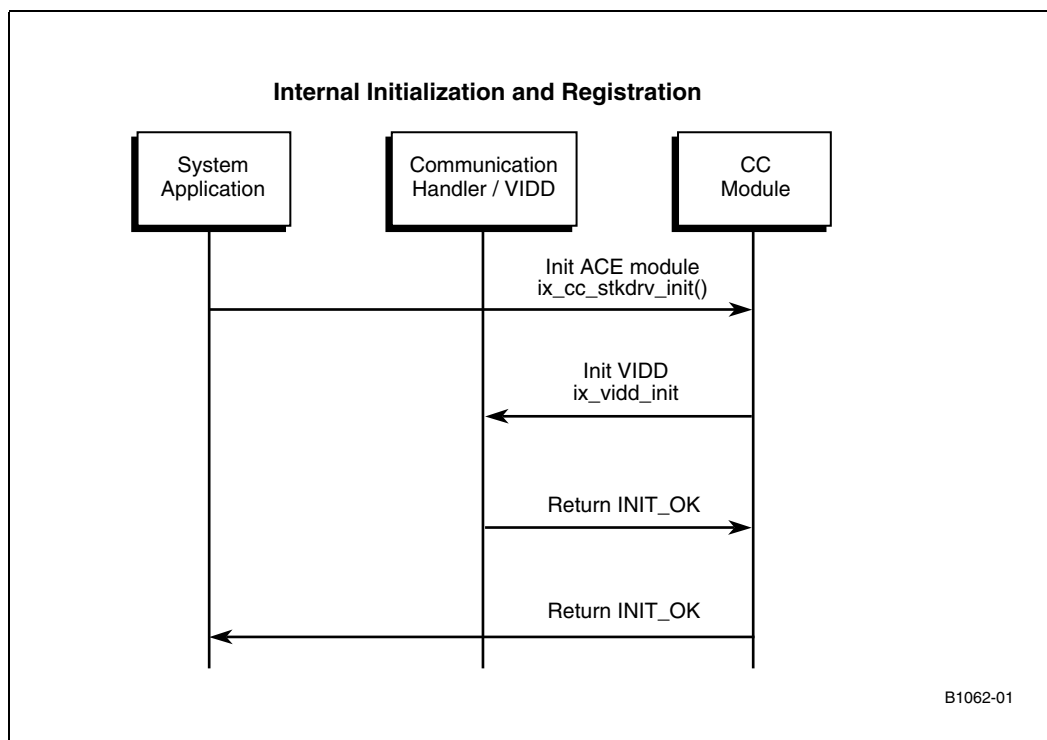
64.3.3.4 Initialization

At system startup, the System Application initializes the Stack Driver with the following steps:

1. Initialize the Stack Driver Core Component Module by calling `ix_cc_stkdrv_init`.
2. `ix_cc_stkdrv_init` calls the initialization function for the communication handler (`ix_cc_stkdrv_vidd_init` for the local VIDD and `ix_cc_stkdrv_tm_init` for the Transport Module), which performs handler specific initialization, and registers callback functions for handling packets and messages.
3. After all handlers are registered with the Core Component Module, initialization is complete. However, local interfaces must still be loaded by an external source in order to successfully send packets up to the local stack. An external source (in this release, the Interface RX core components) loads local interfaces for the local VIDD by calling the property API to add interfaces, as described in [Section 64.3.3.7, “Properties API” on page 909](#).

Figure 64-4 illustrates the initialization sequence between the System App, a Communication Handler (e.g. local VIDD), and the Core Component Module.

Figure 64-4. Stack Driver Internal Initialization and Registration



64.3.3.5 Shutdown

The Core Component Module implements a termination function, `ix_cc_stkdrv_fini` which frees any memory allocated by the Core Component Module.

64.3.3.6 Packet and Message Processing API

In addition to the callback definition, each communication handler uses direct API calls for packet and message forwarding to the Stack Driver Core Component from the driver handler. This API is summarized in [Table 64-4](#). See [Section 27.3.3, “Packet and Message Processing API” on page 579](#).

Table 64-4. Stack Driver Packet and Message Processing API

Name	Description
<code>ix_cc_stkdrv_send_packet()</code>	Queues a packet for transmission.
<code>ix_cc_stkdrv_send_msg_str()</code>	Sends a string message from a handler module to the Stack Driver core component.
<code>ix_cc_stkdrv_send_msg_int()</code>	Sends an integer message from a handler module to the Stack Driver core component.

64.3.3.7 Properties API

The Stack Driver must export APIs to set dynamic properties for which it is a master. See [Section 27.3.4, “Properties API” on page 582](#).

Table 64-5. Stack Driver Properties API

Name	Description
<code>ix_cc_stkdrv_async_get_property()</code>	Returns the interface properties of a specific port.
<code>ix_cc_stkdrv_cb_get_property()</code>	The callback function prototype used by the function <code>ix_cc_stkdrv_async_get_property()</code> .
<code>ix_cc_stkdrv_async_get_num_ports()</code>	Returns the number of ports configured on the Stack Driver core component.
<code>ix_cc_stkdrv_cb_get_num_ports()</code>	The callback function prototype used by the function <code>ix_cc_stkdrv_async_get_num_ports()</code> .

64.3.4 Packet Classifier Design

In order to classify a packet correctly, the Stack Driver core component has the flexibility to activate different types of filters and apply filters based on certain ranges.

64.3.4.1 Theory of Operation

The Packet Classifier or packet filter is a logical block of the Stack Driver Core Component Module that directs packets from the Core Component Module to the local VIDD or Transport Module. Incoming packets can have three destinations:

- Local Legacy Stack
- Remote Legacy Stack
- Local or Remote NPF Stack

To reach the Local Legacy Stack, packets are directed to the Local VIDD Module. For the other two cases, packets are directed to the Transport Module.

In this system, classification is based on the following fields:

- destination IP address/range
- input port ID/range

Other types of filters (IPv4 - 5-tuple, IPv4 - 6-tuple, IPv6 - 5-tuple, IPv6 - 6-tuple, ARP filter, ATM filter, FRAME filter, POS filter, etc.) can be added, but for IXA SDK 3.1, only the minimum classification needed to distinguish packets headed for the local VIDD or Transport Module is implemented. The modularity of this design allows a developer to disable the Packet Classifier or to replace it.

Note: The Packet Classifier is by no means meant to have full classification capabilities; its sole purpose is to determine which packets go to the local VIDD and which packets go to the Transport Module.

64.3.4.2 Packet Classifier Design

In order to classify the packet correctly, the Packet Classifier must have the flexibility to activate different types of filters and apply the filters based on certain ranges. Filters should not be hard-coded, because this prevents filters from dynamically changing in the future.

Below is the summary of the design features of the Packet Classifier:

- Ability to dynamically activate different type of filters
- Ability to apply different filters to the same packet until we find a match or have applied all filters.
- Ability to mix and match different classification categories and flag active categories- for example, select destination IP as a decision maker over input port ID.
- Filters should provide well-defined fields of classification (IP addresses).
- Ability to install these filters through a separate VxWorks task.

64.3.4.3 Packet Classifier Data Structures

Packet Classification data structures are summarized in [Table 64-6](#). See [Section 27.4.1, “Packet Classifier Data Structures”](#) on page 586.

Table 64-6. Stack Driver Packet Classifier Data Structures

Name	Description
<code>ix_cc_stkdrv_filter_type</code>	Represents filter type.
<code>ix_cc_stkdrv_filter_priority</code>	Represents the priority of a filter—either high, medium, or low priority.
<code>ix_cc_stkdrv_ipv4_address_range</code>	Specifies an inclusive address range.
<code>ix_cc_stkdrv_port_range</code>	Specifies a port range.
<code>ix_cc_stkdrv_ipv4_range_filter</code>	
<code>ix_cc_stkdrv_filter</code>	Represents a filter.
<code>ix_cc_stkdrv_filter_index_node</code>	Represents an element in a linked list of filter indices.
<code>ix_cc_stkdrv_filter_handle</code>	An opaque handle to a filter.
<code>ix_cc_stkdrv_filter_ctrl</code>	Specifies the control structure for all filters.

64.3.4.4 Packet Classifier External API

This section describes the Packet Classifier external interface of the Stack Driver Core Component Module. This interface is summarized in Table 64-7. See Section 27.4.2, “Core Component Infrastructure API” on page 591.

Table 64-7. Stack Driver Packet Classifier External API

Name	Description
<code>ix_cc_stkdrv_init_filters()</code>	Creates an empty array of filter handlers and initializes each priority list to NULL.
<code>ix_cc_stkdrv_async_add_filter()</code>	Adds a filter to the array of filters and also places the filter index into the appropriate priority list.
<code>ix_cc_stkdrv_async_remove_filter()</code>	Removes a filter from the array of filters and from the appropriate priority list.
<code>ix_cc_stkdrv_async_remove_all_filters()</code>	Removes all filters from the array of filters and empties all priority lists.
<code>ix_cc_stkdrv_async_modify_filter()</code>	Modifies a filter with new input filter criteria and data.
<code>ix_cc_stkdrv_classify_pkt()</code>	Classifies a packet using the installed packet filters.

64.3.4.5 Packet Classifier Data Flow

During initialization of the system, the Core Component Module does the following:

- Instantiates a filter data structure
- Creates all fields in the structure and calls `ix_cc_stkdrv_add_filter`. Inside this function, a `memcpy()` of the `ix_cc_stkdrv_ipv4_range_filter` structure is done. The calling application is not required to preserve memory for the structure.
- After the successful allocation of the Stack Driver packet filter structure, this function returns a pointer to the `ix_cc_stkdrv_filter_handle`. The filter handle should be preserved by the calling application in order to do proper a `ix_cc_stkdrv_remove_filter()` call.

The following steps represent the external data flow for adding and removing the filter:

- The Core Component Module parses the system registry or an external header file for filter info.
- It constructs the `ix_cc_stkdrv_ipv4_range_filter` structure, which specifies the types of packets and filters.
- The Core Component Module (or the Control Plane PDK) calls `ix_cc_stkdrv_add_filter()` to install the filter. The function has a pointer to `ix_cc_stkdrv_filter_handle` as an output parameter and the Core Component Module stores the `ix_cc_stkdrv_filter_handle`.
- On shutdown or after input from the user's application, the Core Component Module calls `ix_cc_stkdrv_remove_filter()` with `ix_cc_stkdrv_filter_handle` as an argument.

Internally, the Stack Driver uses the filter data structure to store and handle filter information.

Filters are applied on each incoming packet by looking up entries by linear search in the filter array and matching its contents with the packet header. Since this step affects performance, a conditional compilation should be used to produce a version without filtering.

Filtering returns the type of the packet - local, remote or NPF—and the communication handler ID. Based on this classification, the Core Component Module sends the packet either to the local VIDD or to the Transport Module along with the packet type. If a packet does not meet any of the filtering criteria, then, by default, it is classified as `IX_CC_STKDRV_PACKET_TYPE_LOCAL_LEGACY` and sent to the local VIDD.

64.3.5 Outgoing Packet Classifier Design

The Outgoing Packet classifier is a logical block of the Stack Driver core component module that directs different type of packets to different core-components. For example, an outgoing IPv4 packet may be directed to IPv4 core-component and an IPv6 packet may be directed to IPv6 core-component. Outgoing packets can have multiple destinations. However, the current implementation supports the following destination:

- IPv4 core component
- IPv6 Forwarder core component
- Queue Manager core component

64.3.5.1 Outgoing Packet Classifier Data Structures

Table 64-8 lists the data structures and data types used by the Outgoing Packet Classifier.

Table 64-8. Stack Driver Outgoing Packet Classifier Data Structures

Name	Description
<code>ix_cc_stkdrv_og_pkt_type</code>	Defines the packet type.
<code>ix_cc_stkdrv_og_filter_type</code>	Defines the filter type.
<code>ix_cc_stkdrv_og_cc_type</code>	Defines the outgoing core component type.
<code>ix_cc_stkdrv_port_range</code>	Defines the filter range.
<code>ix_cc_stkdrv_cb_og_chk_filter</code>	Defines the function pointer for the actual filtering function.
<code>ix_cc_stkdrv_og_filter</code>	Defines the data structure for the filter.
<code>ix_cc_stkdrv_og_comm_cc_map</code>	Defines a map between the core component and communication ID.
<code>ix_cc_stkdrv_og_filter_ctrl</code>	Defines the control structure for the filters.

64.3.5.2 Outgoing Packet Classifier Internal API

This section lists API functions implemented to support outgoing packet classification. These APIs are exposed only to the stack driver module.

Table 64-9 lists the data structures and data types used by the Outgoing Packet Classifier.

Table 64-9. Stack Driver Outgoing Packet Classifier Internal API

Name	Description
<code>ix_cc_stkdrv_init_og_filters()</code>	Initializes the filter control structure with available filters.
<code>ix_cc_stkdrv_fini_og_filters()</code>	Clears the filter control structure.
<code>ix_cc_stkdrv_classify_output_id()</code>	Obtains the communication ID for the outgoing packet.

64.3.5.3 Outgoing Packet Classifier Data Flow

During initialization of the system, the core component module instantiates the filter control structure. The filter control structure is un-initialized when the core component module is un-initialized.

Some of the key points in the packet filtering process are listed below:

- Obtain the filter type for each outgoing packet. The filter type for each outgoing packet is selected by comparing the outgoing port ID for the packet with the port ID range for that filter.
- Each filter type may be associated with a filtering criterion. The associated filtering function may use the filtering criterion, if defined.

Once the filter has been selected, the associated filtering function is invoked which obtains the outgoing core component type. The filtering function may have nested classification logic. For example, a packet destined for physical interface 0 may have a filter type defined as “IX_CC_STKDRV_OG_FILTER_TYPE_PKT” which means obtaining the outgoing core component using the information carried in the packet.

The packet type information may result in identification of packet as IPv4 packet which may mean destination as IPv4 forwarder core component, but the second level of classification may result in identification of destination IP address as a multicast address, which identifies the destination as Queue Manager core component. In all cases, it would be the responsibility of the defined filtering function to take care of all possible scenarios and provide the correct destination core component.

Based on the outgoing core component type, the outbound communication ID is obtained from the communication ID - core component map.

64.3.5.4 Outgoing Packet Classifier Design Scalability

This design currently provides for communication between stack driver and IPv4 and IPv6 components. In the course of development, it is possible that the stack driver may need to communicate with new core components. The following needs to be done to add communication between stack driver and any core component.

- Define the new core component type in `ix_cc_stkdrv_og_cc_type`
- Initialize the corresponding communication ID in the Map field of the `ix_cc_stkdrv_og_filter_ctrl` structure during outgoing filter initialization.
- Identify the filter type to be used for this communication.
If the filter type is “IX_CC_STKDRV_OG_FILTER_TYPE_PKT”, modify the associated filtering function to obtain the core component accordingly.
If the filter type is “IX_CC_STKDRV_OG_FILTER_TYPE_USR_DEFINED” (See Note)

- Add a new filter type in `ix_cc_stkdrv_og_filter_type`.
- Define any associated filtering criterion, and associate it with the filter during initialization.
- Define a new filtering function whose type is defined in section `ix_cc_stkdrv_cb_og_chk_filter`, and initialize it in the outgoing filter control structure within the appropriate filter.

Note: The option for a user-defined filter type is provided to support scenarios where application-specific criterion is needed, along with information carried in the packet, to identify the outgoing core component.

64.3.6 VIDD for VxWorks

The Virtual Interface Device Driver (VIDD) for VxWorks has the same design requirements as its counterpart for a Linux System. The goal is to expose the IXP ports as regular network interfaces to the XScale TCP/IP stack. Essentially, the VIDD is a network device driver. There two types of network drivers under VxWorks:

- **Enhanced Network Drivers:**
An Enhanced Network Driver (END) is a data link layer driver model that uses MUX functions to communicate with network protocols. By registering with the MUX interface, END drivers have greater flexibility to work with different network protocols. The MUX is an OS layer through which network protocols communicate with the data link layer. Traditional model drivers communicate directly with protocol layer.
The MUX-based model for network drivers contains standardized entry points that are not present in the traditional drivers. VIDD on VxWorks is an Enhanced Network Driver, and conforms to MUX APIs. The VIDD for IXA SDK 3.1 is implemented as an NPT (Network Protocol Toolkit) driver, which has the functionality of an END except that it deals with IP packets rather than Ethernet packets.
- **Traditional network drivers—drivers supporting 4.3 BSD driver interface.**
This driver supports the 4.3 BSD driver interface and is tightly coupled with upper level network protocols. It does not use the MUX interface, which can result in issues related to portability across network protocols. However, WindRiver plans on conforming all future OS network protocols to the MUX interface; contact your WindRiver representative for details.

In addition, VxWorks OS has a flat memory model without division between kernel and user mode. This leads to availability of threads and pre-emptive multitasking for VxWorks device drivers. This is different from the Linux driver model, which is not preemptive, runs to completion, does not have threads, and does not care about locking the data from synchronous access.

64.3.6.1 VIDD System Data Structures

Table 64-10 lists the data structures required by the VxWorks kernel for any network driver. See Section 27.6.1, “VIDD System Data Structures for VxWorks” on page 606.

Table 64-10. Stack Driver VIDD System Data Structures

Name	Description
DEV_OBJ	The DEV_OBJ structure is the glue linking the device-generic END_OBJ structure with the device-specific data object referenced by pDevice.
END_ERR	This error data structure holds system and user defined errors and is used to pass errors from virtual interface device driver to the control-plane protocol.
END_OBJ	Defines device-independent state that is maintained by all drivers and devices.
M2_INTERFACETBL	An M2_INTERFACETBL structure tracks the MIB-II variables used in the driver.

64.3.6.2 VIDD Local Data Structures

Table 64-11 lists the VIDD local data structures. See Section 27.6.2, “VIDD Local Data Structures for VxWorks” on page 611.

Table 64-11. Stack Driver VIDD Local Data Structures

Name	Description
ix_cc_stkdrv_vidd_physical_if_node	A list of data structures created upon initialization by the VIDD and representing the hardware network interfaces of the Intel® IXDP2400 Advanced Development Platform base card.
ix_cc_stkdrv_vidd_fp_node	Represents a forwarding plane object.
ix_cc_stkdrv_vidd_ctrl	Represents the control information for interfaces and forwarding planes and is used as a context for all communication between the Stack Driver core component and the VIDD.

64.3.7 MUX Interface API

Table 64-12 lists the required functions used by the VIDD driver to support the MUX interface. See Section 27.7, “MUX Interface API” on page 614.

Table 64-12. Required Functions for the Stack Driver MUX Interface

Function Name	Description
nptLoad()	Load a device into the MUX and associate a driver with the device.
nptUnload()	Release a device, or a port on a device, from the MUX.
nptSend()	Accept data from the MUX and send it on towards the physical layer.
nptMCastAddrAdd()	Add a multicast address to the list of those registered for the device.
nptMCastAddrDel()	Remove a registered multicast address from the list of those registered for the device.
nptMCastAddrGet()	Retrieve a list of multicast addresses registered for a device.
nptPollSend()	Send frames in polled mode rather than interrupt-driven mode.

Table 64-12. Required Functions for the Stack Driver MUX Interface (Continued)

Function Name	Description
<code>nptPollReceive()</code>	Receive frames in polled mode rather than interrupt-driven mode.
<code>nptStart()</code>	Connect device interrupts and activate the interface.
<code>nptStop()</code>	Stop or deactivate a network device or interface.
<code>nptIoctl()</code>	Support various ioctl commands.

64.3.7.1 VIDD System Function Calls

These function calls are implemented by the VIDD in order to conform to the MUX/END interface. These functions control all interactions between the VIDD and VxWorks:

- Configuration and initialization
- Shutdown processing
- Packet receiving
- Packet sending
- Memory allocation
- IOCTL handling

The addresses of these functions are in the `END_OBJ` structure so they can be called from the MUX library. The functions are summarized in [Table 64-13](#). See [Section 27.7.2, “VIDD System Function Calls”](#) on page 616.

Table 64-13. VIDD System API

Name	Description
<code>ix_cc_stkdrv_vidd_npt_load()</code>	Creates a logical interface on the VIDD side and registers it with the VxWorks MUX layer.
<code>ix_cc_stkdrv_vidd_npt_unload()</code>	This function is called by the MUX to release the device.
<code>ix_cc_stkdrv_vidd_npt_start()</code>	Brings up the IXP network interface specified by the <code>END_OBJ</code> structure and makes the interface active and available to the OS.
<code>ix_cc_stkdrv_vidd_npt_stop()</code>	Brings down the IXP interface specified by the <code>END_OBJ</code> structure and deactivates the interface.
<code>ix_cc_stkdrv_vidd_npt_ioctl()</code>	Provides VIDD support for IOCTL commands.
<code>ix_cc_stkdrv_vidd_npt_send()</code>	The network protocol uses this function to send a network packet to the Stack Driver.
<code>ix_cc_stkdrv_vidd_npt_mCastAddrAdd()</code>	Adds a new link-layer multicast address to the table of multicast addresses for the IXP interface.
<code>ix_cc_stkdrv_vidd_npt_mCastAddrDel()</code>	Removes a previously added link-layer multicast address.
<code>ix_cc_stkdrv_vidd_npt_mCastAddrGet()</code>	Returns the list of all multicast addresses that are active on the interface.
<code>ix_cc_stkdrv_vidd_npt_pollSend()</code>	A polling mode equivalent to the <code>send()</code> routine.

Table 64-13. VIDD System API

Name	Description
<code>ix_cc_stkdrv_vidd_npt_pollRcv()</code>	The current implementation of the core component model does not support <code>PollReceive()</code> .

64.3.7.2 MUX APIs used by the VIDD

Table 64-14 lists the functions exposed by MUX library that are called from the driver to the MUX interface to perform the following actions (See Section 27.7.3, “MUX API used by the VIDD” on page 625):

- Transfer data from driver to the protocol
- Report an error condition on the driver
- Restart network protocol to resume transmission, stopped previously because of error conditions.

Table 64-14. VIDD MUX API

Name	Description
<code>muxTkReceive()</code>	Sends data to the control plane protocol atop the MUX interface.
<code>muxError()</code>	The VIDD calls this function to report an error condition to the network protocol.
<code>muxTxRestart()</code>	The VIDD calls this function to resume sending data from the network protocol—as, for example, when the protocol has stopped sending data when an error was returned from <code>muxSend()</code> .

64.3.8 Packet Processing—VxWorks Example

Incoming network packets arrive at the Core Component Module in the form of `ix_buffer_handles`. The entry point for packets in the Stack Driver is its packet handler function. The following is pseudo code for incoming packet processing:

64.3.8.1 Core Component Module Side

```

ix_error ix_cc_stkdrv_high_priority_pkt_handler(ix_buffer_handle buf unsigned
int exceptionCode, void *ctx)
{
    - Call ix_cc_stkdrv_process_pkt()
}

ix_cc_stkdrv_process_pkt (ix_buffer_handle arg_hDataToken, ix_cc_stkdrv_ctrl*
arg_pStkdrvCtrl)
{
    - If classification is enabled, allocate memory for a handler ID and a packet
type variable, and call ix_cc_stkdrv_classify_pkt, which fills in the handler
ID and packet type for the input packet. If classification is not enabled,
find the local VIDD in the list of handlers stored in arg_pStkdrvCtrl, and
calls the receive_pkt callback function with the input packet and return
IX_SUCCESS if the callback returns successfully, or the matching error code if
the callback returns an error.
    - If packet classification returns an error, return
IX_CC_STKDRV_ERROR_CANNOT_CLASSIFY_PKT.

```

```

- If the packet has been successfully classified, we have a handler ID and
packet type associated with the input packet. Find the handler in the handler
list with the appropriate handler ID, and call the receive_pkt callback for
the handler with the input packet. Return IX_SUCCESS if the callback returns
successfully, or IX_CC_STKDRV_ERROR_HANDLER_RECV if the callback returns an
error.
}

```

64.3.8.2 VIDD Side

```

ix_error ix_cc_stkdrv_vidd_receive_pkt (ix_buffer_handle buffer, void* ctx,
ix_cc_stkdrv_packet_type arg_packetType)
{
    - Extract the input port ID from the metadata of arg_pBuffer. Each
interface in VIDD is represented by an END_OBJ data structure.
    - Extract the VIDD control structure from arg_pCtx and return
IX_CC_ERROR_NULL if it is invalid. Given the input port ID, look-up the
END_OBJ data structure from the list of interfaces contained in the VIDD
control structure.
    - Get an mBlk (VxWorks buffer) from the memory pool. If we cannot get an
mBlk return IX_CC_ERROR_OOM.
    - Check the alignment of the IP header in the ix_buffer. If it is double-
word aligned, then set the data pointer in the mBlk to the ix_buffer data
pointer. Otherwise we must copy IP data from ix_buffer into the mBlk. This is
a necessary performance hit as VxWorks requires its network buffers to be
double-word aligned.
    - Call the receive routine for the stack with a pointer to the mBlk chain.
    - Free the ix_buffer_handle.
    - Return IX_SUCCESS to the Core Component Module.
}

```

Packets that are sent from control plane stacks first come into the MUX library. Network protocols call into the MUX with `muxSend()` function. This function in turn calls `endSend()` callback into the VIDD. Below is the processing done inside the VIDD `ix_cc_stkdrv_vidd_npt_send()` function.

64.3.8.3 Outgoing Packet Processing Pseudocode

The following is pseudocode for outgoing packet processing (i.e. from control plane stacks to the microengines).

```

STATUS ix_cc_stkdrv_vidd_npt_send(END_OBJ* pEnd, M_BLK_ID pMblk, char*
dstMacAddr, long netType, void* pSpare)
{
    - From pEnd pointer extract interface id and store it.
    - If the packet is an IP packet, turn off the most significant bit in the IP
ID.
    - From pEnd extract the VIDD control structure and then the freelist handle.
Get an ix_buffer from the freelist. If one cannot be obtained, return ERROR.
    - Copy buffer from the pMblk chain into the ix_buffer.
    - Free the pMblk chain.
    - If the packet is multicast Ethernet, construct the L2 header based on the
MAC address of the interface, and add it to the front of the packet data.
    - Update if_id, and call ix_cc_stkdrv_send_packet()
}

```

```

- If the return value indicates an error, return ERROR.
- Return OK to the MUX.
}

```

64.3.8.4 Pseudocode for Outgoing Packets—in the Core Component Module

```

ix_error ix_cc_stkdrv_send_packet(ix_buffer_handle arg_hBuffer, ix_uint32
arg_ifId)
{
    - Perform a downstream classification based on the output port ID by
    calling ix_cc_stkdrv_classify_output_id().
    - Call ix_cci_send_packet() to send the packet to the output core
    component. If it returns successfully return IX_SUCCESS, else return
    IX_CC_STKDRV_ERROR_TX_PKT.
}

```

64.3.9 VIDD for Linux*

The local VIDD implements all the functions called between the host's TCP/IP stack and the Stack Driver. These implementations are OS-specific and thus will include OS-specific data structures for net devices, ether devices, etc.

64.3.9.1 VIDD System Data Structures for Linux

Table 64-15 lists the data structures used by the VIDD from the Linux TCP/IP stack.

Table 64-15. VIDD System Data Structures for Linux

Name	Description
<code>sk_buff</code>	The Linux socket buffer structure.
<code>net_device</code>	A container for the interactions between the Linux kernel and the VIDD.
<code>ifreq</code>	The interface request structure used for socket ioctl's.
<code>ix_cc_stkdrv_vidd_physical_if_node</code>	A VIDD proprietary structure which represents an interface on the VIDD.

64.3.9.2 VIDD System API for Linux

The driver functions listed in Table 64-16 are called by the Linux kernel to communicate with the driver module and are implemented in the VIDD.

Table 64-16. VIDD System API for Linux

Name	Description
<code>ix_cc_stkdrv_vidd_ifd_open</code>	An open function for the VIDD driver.
<code>ix_cc_stkdrv_vidd_ifd_stop</code>	A stop function for the VIDD driver.
<code>ix_cc_stkdrv_vidd_ifd_tx</code>	Sends a packet to the specified device.
<code>ix_cc_stkdrv_vidd_ifd_set_config</code>	Configures the VIDD driver.

Table 64-16. VIDD System API for Linux

Name	Description
<code>ix_cc_stkdrv_vidd_ifd_do_ioctl</code>	Sets the MAC address for each interface for private IOCTL calls.
<code>ix_cc_stkdrv_vidd_ifd_get_stats</code>	Retrieves interface statistics.
<code>ix_cc_stkdrv_vidd_ifd_set_multicast_list</code>	Sets multicast address and flag settings.
<code>ix_cc_stkdrv_vidd_ifd_init</code>	Initializes the VIDD driver.

64.3.9.3 VIDD Linux Driver Support API

The driver functions listed in [Table 64-17](#) are called by the VIDD module to communicate with the OS kernel. Part of these procedures are called in response to the control and configuration messages received by the core component module.

Table 64-17. VIDD Linux Driver Support API

Name	Description
<code>ix_cc_stkdrv_vidd_init</code>	Initializes the local VIDD driver and registers entry points.
<code>ix_cc_stkdrv_vidd_fini</code>	Terminates the VIDD driver and frees any allocated memory.
<code>ix_cc_stkdrv_vidd_if_devinet_ioctl</code>	Wrapper function for protocol-dependent ioctl operation.
<code>ix_cc_stkdrv_vidd_if_dev_ioctl</code>	Wrapper function for protocol-independent ioctl operation.
<code>ix_cc_stkdrv_vidd_if_up</code>	Enables an interface.
<code>ix_cc_stkdrv_vidd_if_down</code>	Disables an interface.
<code>ix_cc_stkdrv_vidd_receive_pkt</code>	Transfers received packets to the TCP/IP stack.
<code>ix_cc_stkdrv_set_ip_mask</code>	Sets the IP and netmask for a given interface.

64.3.10 Transport Module Design

The transport module sends packets to and receives packets from a remote TCP/IP stack.

64.3.10.1 Transport Module Data Structures

The Transport Module data structures are listed in [Table 64-18](#). See [Section 27.9.1, “Transport Data Structures”](#) on page 639.

Table 64-18. Transport Module Data Structures

Name	Description
<code>ix_cc_stkdrv_tm_ctrl</code>	This is the control structure for the Transport module and is used as a context in receiving packets from the forwarding plane module.

64.3.10.2 Transport Module External API

[Table 64-19](#) lists the Transport Module API. See [Section 27.9.2, “Transport API”](#) on page 639.

Table 64-19. Transport Module External API

Name	Description
<code>ix_cc_stkdrv_tm_receive_pkt()</code>	Receives packets from the core component and passes them up to the forwarding plane module
<code>ix_cc_stkdrv_tm_pkt_handler()</code>	Receives packets from the forwarding plane module passing them down to the core component.

64.3.11 Start-up Configuration File Requirements

The start-up configuration values are passed in via an input context.

The configuration needs to provide the following information during start-up:

- number of IXP ports
- MAC addresses of IXP ports
- IP interface configuration for each of the port, including:
 - IP address
 - Subnet mask
 - Broadcast address
 - Promiscuous or non-promiscuous interface.
- IP information for local interface, including
 - IP address
 - Subnet mask
 - Default gateway
 - Broadcast address.

SoftSAR Components

The SoftSAR includes the following core component:

- [Chapter 65, “SoftSAR Core Components”](#)

The ATM SoftSAR Core Components cooperate together and include the following:

- The SAR Control Core Component is responsible for providing API for user application. It controls all core sub-components (ATM RX, ATM TX and TM4.1) so called SAR Control Plug-ins. It is responsible to for distributing user requests to appropriate sub-components in both single- and dual-IXP processor configurations.
- The ATM RX Core Component is responsible for symbol patching, data structures initialization and configuration of ATM RX microblock. Moreover, it must service exception messages with incoming AAL5 frames on any VC that has the exception flag set in the VC Reassembly Context.
- The TM4.1 Core Component is responsible for symbol patching, data structures initialization and configuration of TM4.1 shaper, scheduler and write-out microblocks. It does not service any exception messages.
- The ATM TX Core Component is responsible for symbol patching, data structures initialization and configuration of QM microblock. It does not service any exception messages.

This chapter introduces general environment and architecture of the SoftSAR core component and general design concepts used for the SoftSAR core components (Section 65.7, “SoftSAR Core Components” on page 933).

The chapter also includes the following ATM core component design:

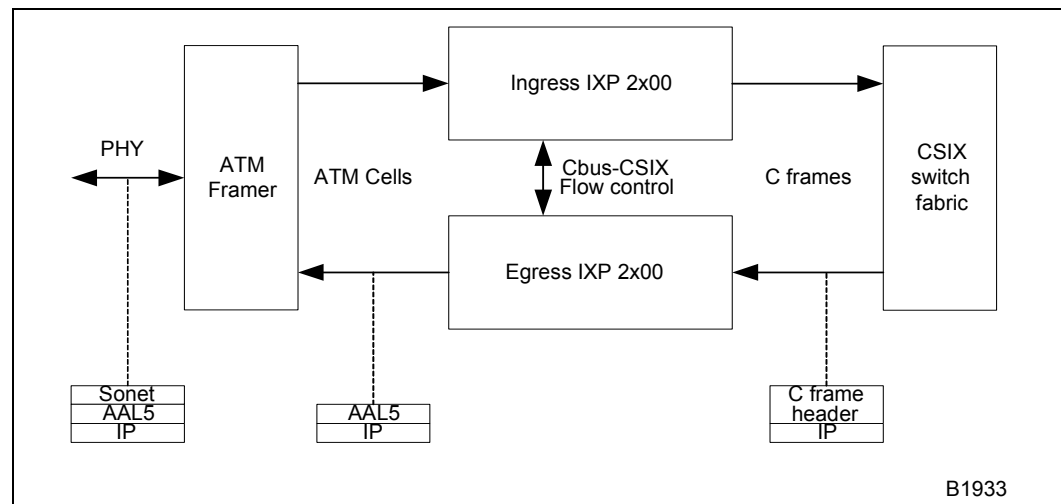
- Section 65.9, “ATM RX Core Component” on page 944
- Section 65.10, “ATM TX Core Component” on page 948
- Section 65.11, “TM4.1 Core Component” on page 951

65.1 Architecture Overview

65.1.1 Hardware Architecture Overview

Figure 65-1 illustrates example of hardware where ATM SoftSAR building blocks can be used.

Figure 65-1. Example of Hardware with Two IXP Processors

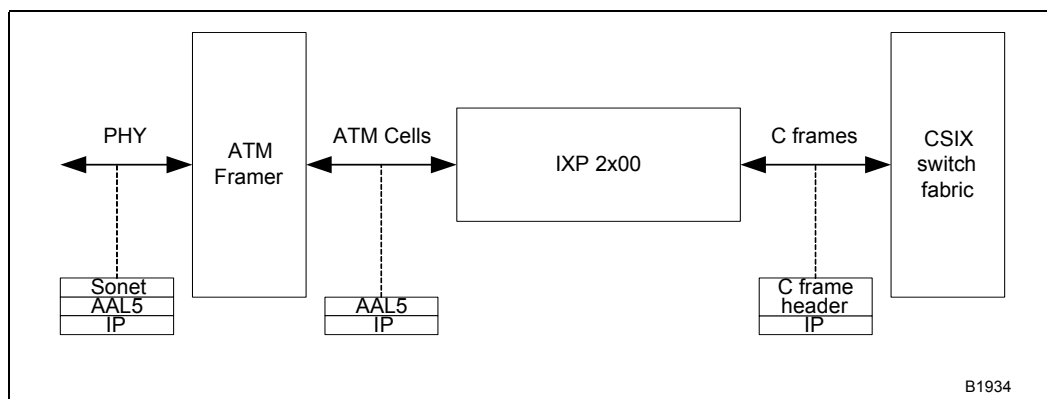


The Ingress IXP2x00 is responsible for receiving ATM cells, processing them into CSIX-C frames and sending the frames via CSIX switch fabric. The egress processor is responsible for receiving CSIX-C-frames, processing them into ATM cells and sending via ATM interface.

Figure 65-2 shows an example architecture where a second hardware platform that supports both directions on single IXP processor.

Note: IXP Network Processor may service two ATM interfaces (the second ATM interface would replace the CSIX). In this case, two SAR instances would be required.

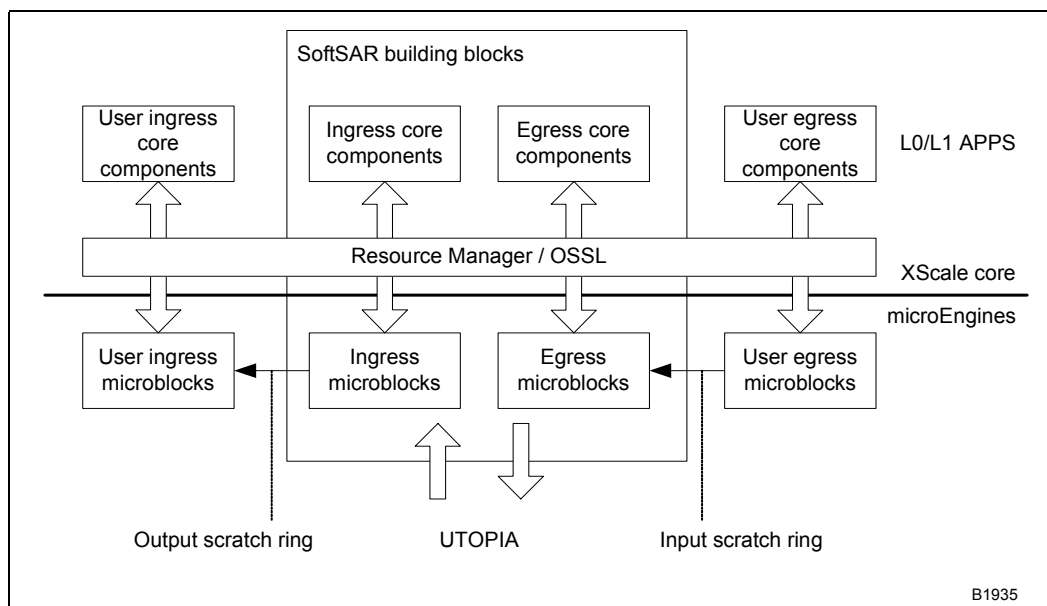
Figure 65-2. Example of Hardware with Single IXP Processor



65.1.2 General Software Architecture Overview

The SoftSAR software components are located partly on the XScale processor (core components) and partly on microengines (microblocks). Figure 65-3 high-level view of the software and the ATM building blocks are compliant this architecture.

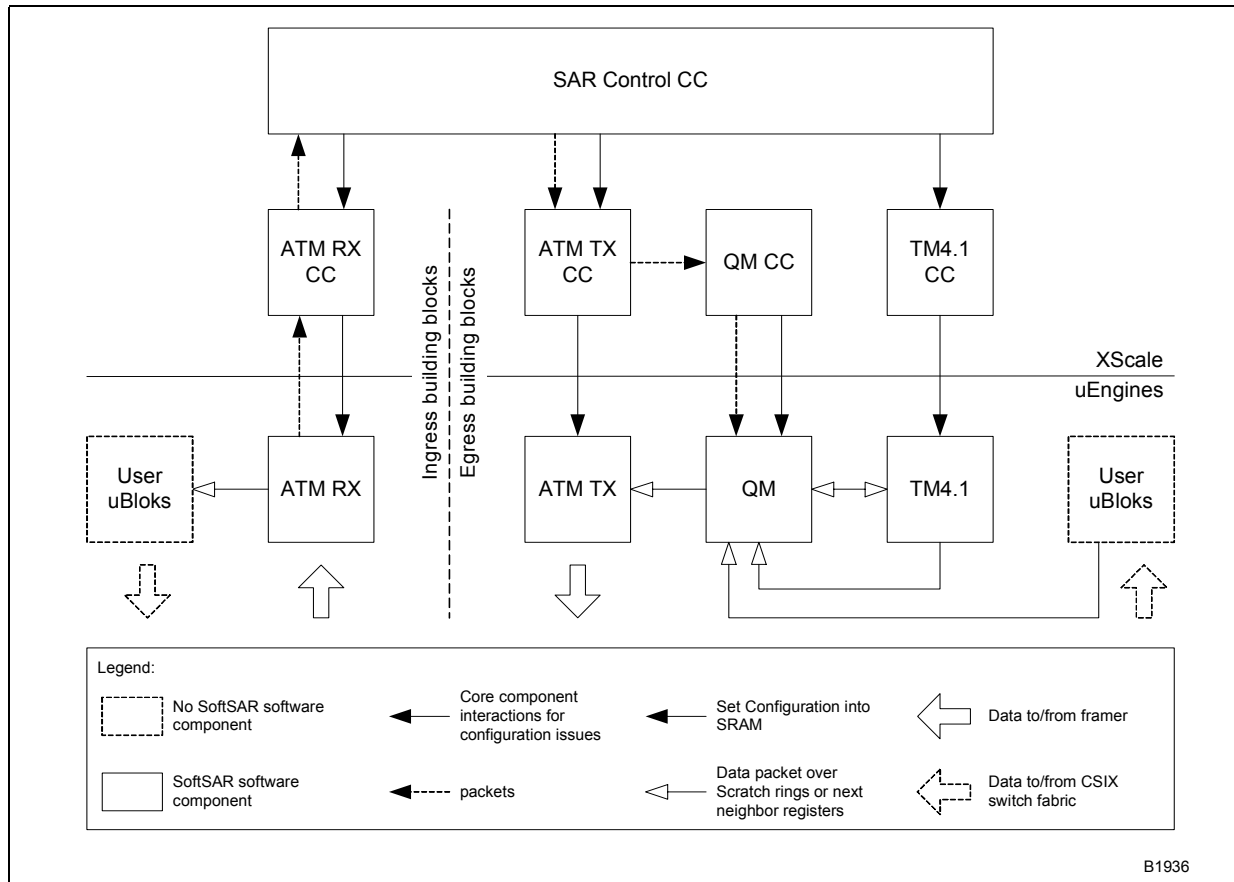
Figure 65-3. General SoftSAR Software Overview



65.1.3 ATM Building Blocks

Figure 65-4 illustrates current architecture of ATM building blocks. The following sections provide a brief overview of the blocks.

Figure 65-4. SoftSAR Software Components



65.1.3.1 Microblocks Overview

The ATM RX microblock receives ATM cells in mpackets coming in on the media interface. It reassembles ATM cells into AAL5 PDU's, writes the data into a buffer in DRAM and queues the packet buffer handle on a ME-ME scratch ring for processing by the next stage of the pipeline. OAM cells are queued to the core as exception packets. In addition, AAL5 frames received on any VC may be sent to the XScale Core, if the exception flag is set in the VC Reassembly Context. The ATM RX, as an option, may also handle AAL2 cells at CPS-packet and SSSAR SDU levels. Future versions will add OAM F4 and F5 handling (and performance monitoring).

Note: AAL-2 has a limited level of support in the Intel® Internet Exchange Architecture Software Development Kit (Intel® IXA SDK). Contact your Intel sales representative for more information.

Figure 65-4 presents only one microblock for TM4.1. The TM4.1 shaper, write-out and scheduler microblocks constitute TM4.1 functionality in microengines.

The TM4.1 shaper microblock ([Chapter 23, “TM4.1 Shaper and Scheduler Microblock”](#)) determines transmission times for ATM cells. The cell transmission times are calculated to be TM 4.1 conformant per the associated VC Usage Parameter Control (UPC) (that is, ensures transmission of VC cells do not violate the traffic contract established for the VC).

The TM4.1 write-out microblock is responsible for populating and updating the ATM cell transmit time queues (that is, the calendar queues the TX scheduler microblock services to schedule ATM cells for transmission at specific times).

The TM4.1 scheduler microblock schedules cells from the time queues and the UBR queues. It is essentially a priority scheduler with the highest priority for the real-time time-queue, the next priority for non-real-time time-queue and the lowest priority for UBR.

The ATM TX microblock transmits ATM AAL5 cells (and AAL2 as an option) over MSF interface ([Chapter 11, “ATM AAL5 TX Microblock”](#)). It receives transmit messages from the queue manager. With each transmit request, the microblock moves an ATM cell into a TBUF, which is then transmitted into the media by the MSF Transmit State Machine.

The Queue Manager microblock is responsible for performing enqueue and dequeue operations on the transmit queues which are implemented using the hardware SRAM link lists. It accepts enqueue requests from the pipeline via a scratch ring. After every enqueue, QM sends enqueue request via scratch ring to TM4.1 shaper. The enqueue requests are on a per-packet basis. The dequeue requests come from the TM4.1 scheduler microengine on a per-cell basis. Also after every dequeue, the QM passes a transmit request via a scratch ring to the ATM TX microblock.

65.1.3.2 ATM Core Components Overview

All core components initialize data structures for their corresponding microblocks and patch symbols in the microblocks. The core components perform dynamic action such as setting configuration, accessing statistics or handling exception packets coming from a microblock.

[ATM RX Core Component](#) is responsible for symbol patching, data structures initialization and configuration of ATM RX microblock. It also services exception messages with incoming OAM cells and with AAL5/AAL2 frames on any VC that has set the exception flag in the VC Reassembly Context.

Note: AAL-2 has a limited level of support in the Intel® Internet Exchange Architecture Software Development Kit (Intel® IXA SDK). Contact your Intel sales representative for more information.

[TM4.1 Core Component](#) is responsible for symbol patching, data structures initialization and configuration of TM4.1 shaper, scheduler and write-out microblocks. It does not service any exception messages.

The Queue Manager core component (See [Chapter 48, “Queue Manager Core Component”](#)) is responsible for symbol patching, data structures initialization and configuration of QM microblock. It does not service any exception messages.

[ATM TX Core Component](#) is responsible for symbol patching, data structures initialization and configuration of QM microblock. It does not service any exception messages.

[SAR Control Agent Core Component Design](#) is responsible for providing API for user application. It controls all core sub-components (ATM RX, ATM TX and TM4.1), SAR Control Plug-ins ([Section 65.4, “Plug-in Core Components” on page 930](#)). It is responsible to for distributing user requests if SAR works on hardware platform with two IXP processors.

65.2 Functionality

The ATM core components provide the following functionality:

- Control and configuration of the ATM microblocks.
- Support for AAL5 and AAL2 traffic
- Support for VC, CID and virtual port adding/deleting
- Support for service classes
- Support for up to 2048 ports
- Support for static virtual port configuration
- Support for matching VC Queue number into VC parameters
- Support for per virtual port shaping
- Support for OAM
- Support for multi-instances of SAR

65.2.1 Handle and Handle Manager Concept

A VC is identified by virtual port, VPI and VCI. The ATM microblocks identify VC with VC queue number (VCQ#), which is a 16-bit unsigned integer. The TM4.1 microblock has additional requirement for VCQ mapping to a newly created VC. If the created VC has traffic parameters that qualify the VC to be 'high bit rate' (HBR) VC it has to use VCQ# in range from 0 to 127. The low-speed (LBR) VCs (that is, slower than line rate/128) must use VCQ# in range from 128 to 65535. Therefore, the assignment should be done in the TM4.1 core component and the VCQ# has to be distributed to other core components and the user application.

As a similar method could be applied also to other objects created/controlled within the ATM block (e.g., CIDs in AAL2), a generic name - handle - is used instead of the specific name like the VCQ#. The core component library that manages handles is called handle manager.

The handle manager API can be called from different core components.

The handle manager provides mechanism for allocating and freeing handles. The allocate operation validates parameters of the requested object.

65.3 ATM Objects Control

SAR Control API allows controlling and managing the following objects:

- ATM virtual ports,
- VCs
- CIDs (AAL2 future option).

Note: AAL-2 has a limited level of support in the Intel® Internet Exchange Architecture Software Development Kit (Intel® IXA SDK). Contact your Intel sales representative for more information.

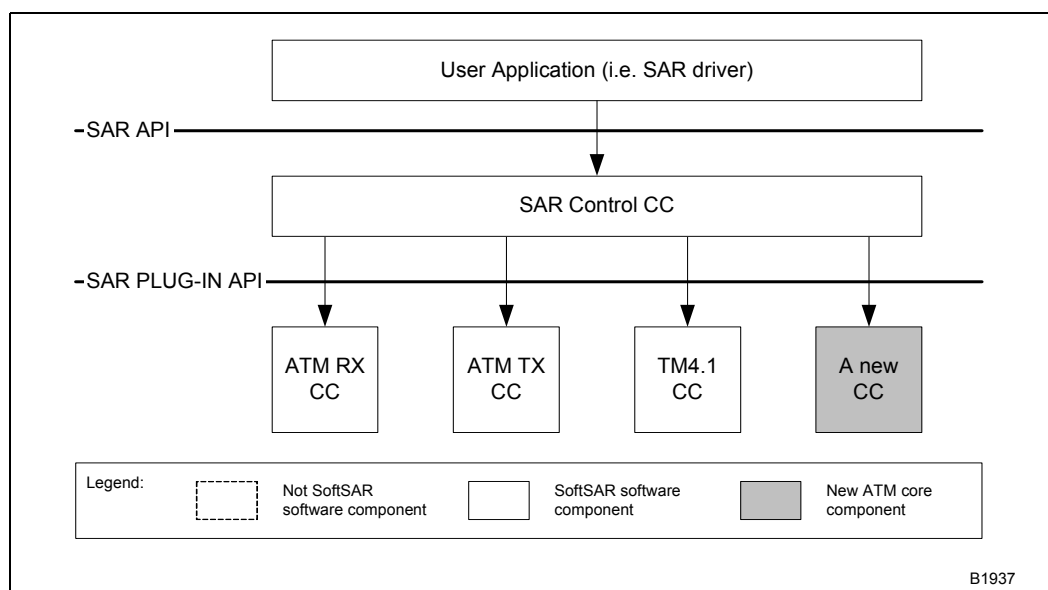
For each object ATM core components provide operations for creation, removing and reading statistics for the object. The objects in SAR Control API are identified by handles. The handle is unique, 16-bit identifier used in microblocks to access an object. The handles are described [Section 65.2.1, “Handle and Handle Manager Concept” on page 929](#).

65.4 Plug-in Core Components

SAR Control core component provides SAR Control API ([Section 65.7, “SoftSAR Core Components” on page 933](#)) for user application. The API performs all control and management operations on ATM building blocks. The SAR Control module distributes user requests to sub-modules that are registered with SAR Control. The registered sub-modules are called SAR Control plug-ins. The architecture allows adding/replacing ATM building blocks without changing user application and the rest of ATM building blocks. The interactions and structures between SAR Control and its plug-ins are described along with SAR Control plug-in API. [Figure 65-5](#) shows the overall architecture.

Note: The plug-in API must never be called directly from the user/calling application.

Figure 65-5. SAR Control Architecture Overview



65.4.1 Plug-in Registering

The SAR Control core component is independent on architecture of microblocks. The SAR Control knows only a list of plug-ins to service—the list is configurable for each application and is read from configuration file or a header file.

The SAR Control core components provides mechanism for registering plug-ins and their services into SAR Control. The available services are:

- Operations on objects. Registration for different kind of objects (that is, VC, CID or virtual port) may be performed independently.
- Handle management.
- OAM services

65.5 Support for Single/Dual IXP Hardware Configuration

This section describes how the single- and dual-processor hardware platforms are supported. Figure 65-6 shows software architecture with hardware platform utilizing a single IXP processor. Additionally, Figure 65-6 also shows SAR Control APIs used in this architecture—user application and SAR Control plug-ins.

Figure 65-6. Place of Software Blocks in Single IXP Hardware Platform

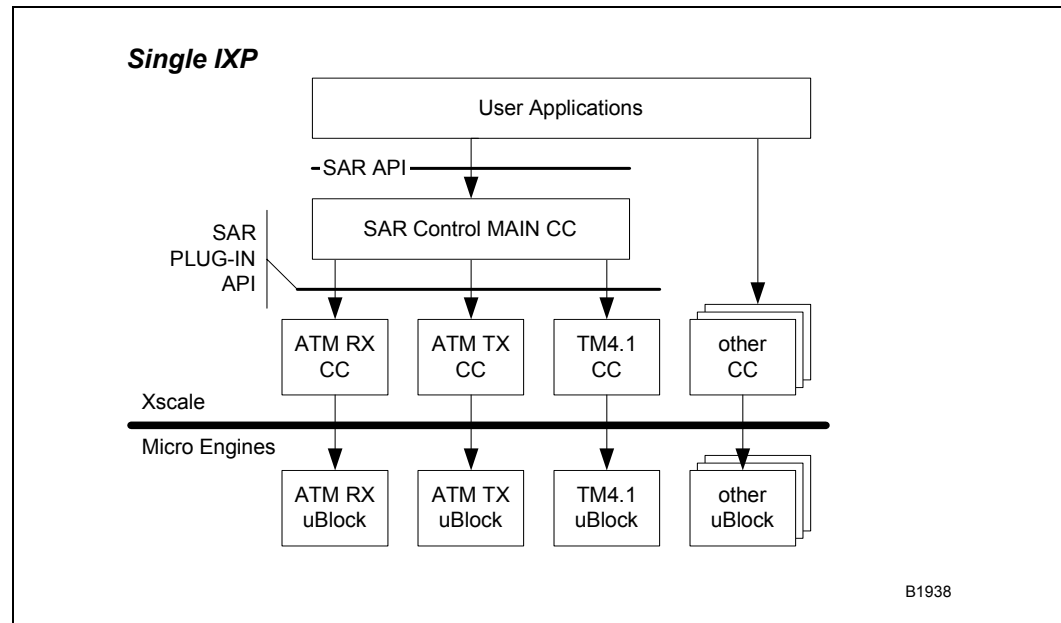
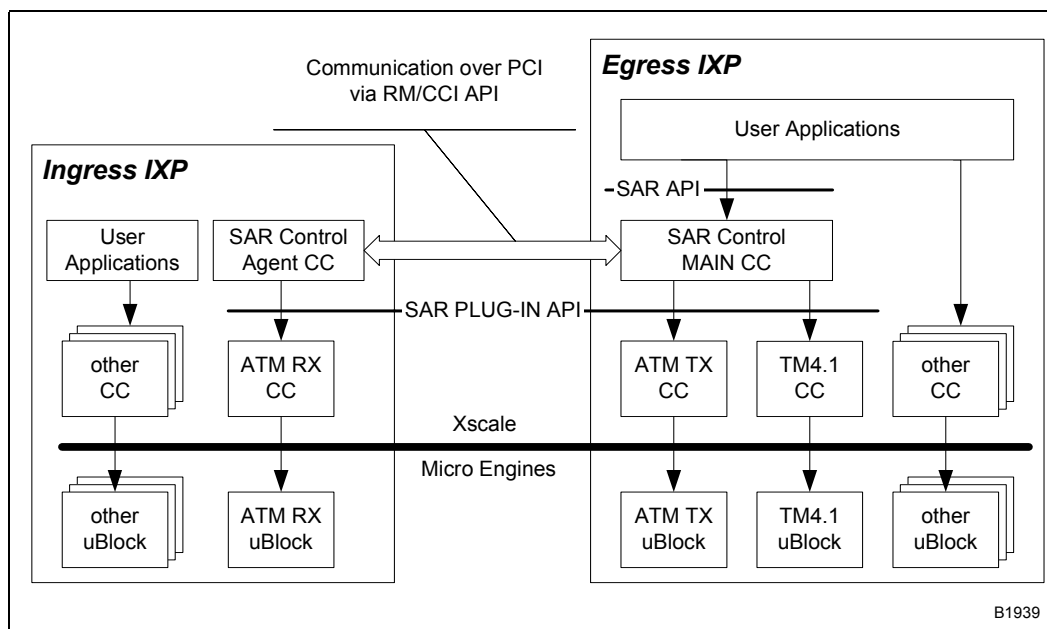


Figure 65-7 shows software architecture with hardware platform with dual IXP processors.

Note: The SAR Control core component is now distributed and consists of two parts—Main core component and Agent core component. From the point of view of a user application, however, there is no difference between single and dual-processor SAR core component versions.

Figure 65-7. Place of Software Blocks in Dual IXP Hardware Platform



65.6 Multi-Instances Support

ATM core components comply with multi-instance requirements summarized below as follows:

- The System Application and init functions must establish proper mapping between core component instances, Registry properties subtrees and communication ID sets to support multiple instances.
- The Messaging API must address the multi-instance need. In order to achieve different communication ID bindings, all invocations of `ix_cc_add_xxx_handler()`, `ix_rm_handler_register()` and `ix_rm_packet_send()` should not have communication IDs put directly as a parameter (the value of parameter should be determined based on the core component instance).
- There must be a way for a core component to determine in which instance it operates, which should be accessible from core component handle.

The main features of the multi-instance IXA Portability Framework compliant solutions are as follows:

- The core component handle is the central place holding crucial information on core component—index, execution engine, the used policy and instance number
- Each created instance of core component receives its private context
- Each user core component init function must save assigned core component handle in the context
- Each core component function, depending on the instance number, may retrieve the instance number from the core component handle stored in the context (using a macro)

- Each user core component init function accesses different copy of properties set in Registry based on specific name prefix indexed by the instance number
- Assignment of core component ID, instance numbers and communication ID binding is static (resolved at compile time)
- All message/packet handlers registration functions and message/packet send function must choose a proper Communication IDs based on the instance number
- All single-instance core components need not be changed
- The header file `bindings.h` is specific for each application so consequently it must be placed in somewhere in the application's directory tree

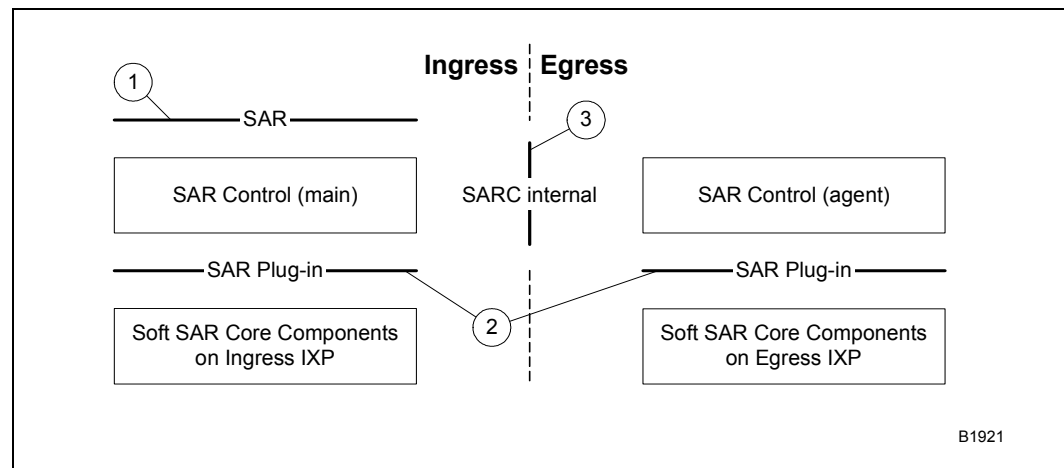
65.7 SoftSAR Core Components

This section presents general design concepts used for the SoftSAR (Software Segmentation and Reassembly) core components.

The two sub-components constitute the SAR Control—the main SAR Control ([Section 65.7.2, “SAR Control Main Core Component” on page 936](#)) and the SAR control agent ([Section 65.8, “SAR Control Agent Core Component Design” on page 941](#)). The SAR Control agent is used exclusively in dual IXP hardware architecture—[Section 65.1.1, “Hardware Architecture Overview” on page 925](#).

[Figure 65-8](#) illustrates the decomposition of SAR Control and discusses the scheme for core component inter-communication.

Figure 65-8. SAR Control Decomposition



The Core Component Infrastructure framework defines two types of APIs provided by a core component—library API and messaging (asynchronous) API. This section describes details of the APIs used in SoftSAR.

[Figure 65-8](#) shows the main SoftSAR programming interfaces:

- SAR API—API for user modules. SAR Control supports both messaging and library API.
- SAR Plug-in API—Used for communication between the SAR Control and plug-in SAR core components. The messaging API is not required.

- SARC internal API—It is internal path of communication between SAR Control and SAR Control Agent. This API uses RM messaging scheme for communication between SAR Control sub-components and the rest of SoftSAR building blocks.

The SAR Control is able to work on a single processor as well as on egress or ingress processor in dual processor configuration. The SAR Control Agent (if exists in the system) has to reside on a different IXP processor than the SAR Control.

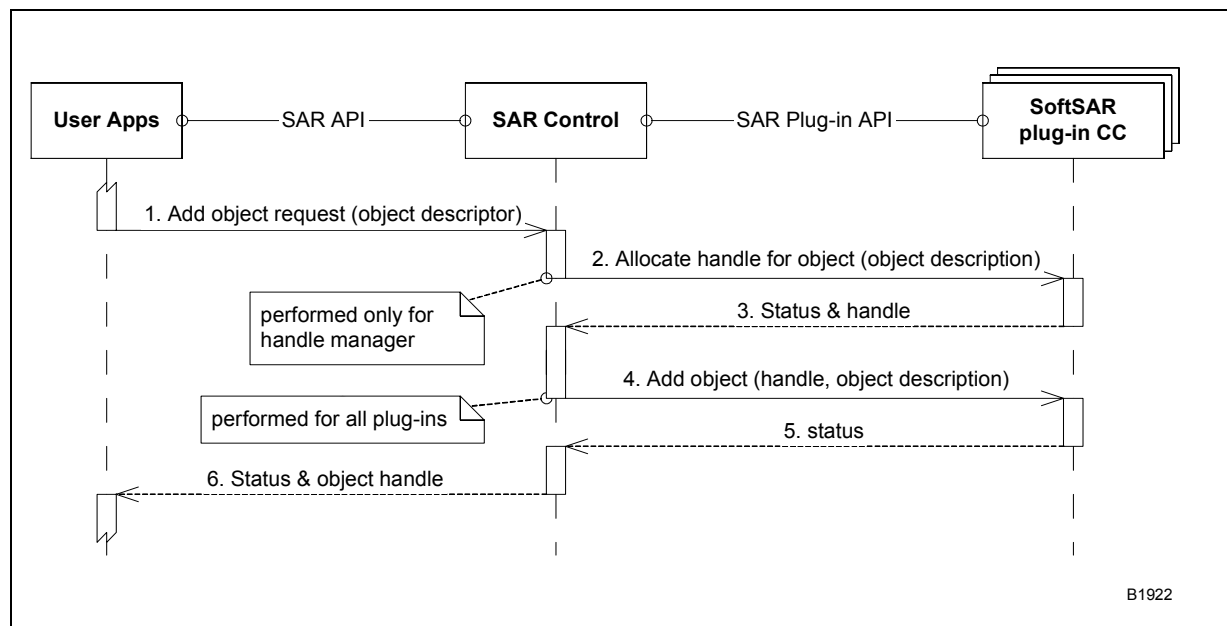
The SAR core components initialize in the following sequence:

1. All SoftSAR sub-components (ATM RX, ATM TX and TM4.1) gets initialized
2. SAR Control Main (and optional Agent) gets initialized. SAR control Main retrieves the list of plug-ins to be initialized from the system registry (or takes defaults)
3. All plug-ins init functions gets called. The functions register with the SAR Control Main
4. After all expected plug-ins are initialized, the SAR core component is ready to be used

65.7.1 Dynamic Behavior

This section describes interactions between user application, SAR Control and SAR Control plug-ins. [Figure 65-9](#) shows scenario with handle allocation/assignment, and propagation of the handle to the rest of the plug-ins and user application during “create” operation.

Figure 65-9. Create Operation Data Flow



The following steps are performed for create operation:

1. User application calls routine provides by SAR Control core component, the request contains parameters for a new object
2. SAR control calls handle manager for the object
3. If return status is OK, a valid handle is returned. If not, the handle is NULL

4. SAR control calls add object function with the handle as one of parameters
5. The function performs the requested action and returns the final status
6. Finally, user application is passed the status along with the handle, to be used for subsequent accesses to the object

Figures 65-10 and 65-11 illustrate similar sequences of action connected with object removal and statistics access.

Figure 65-10. Remove Operation Data Flow

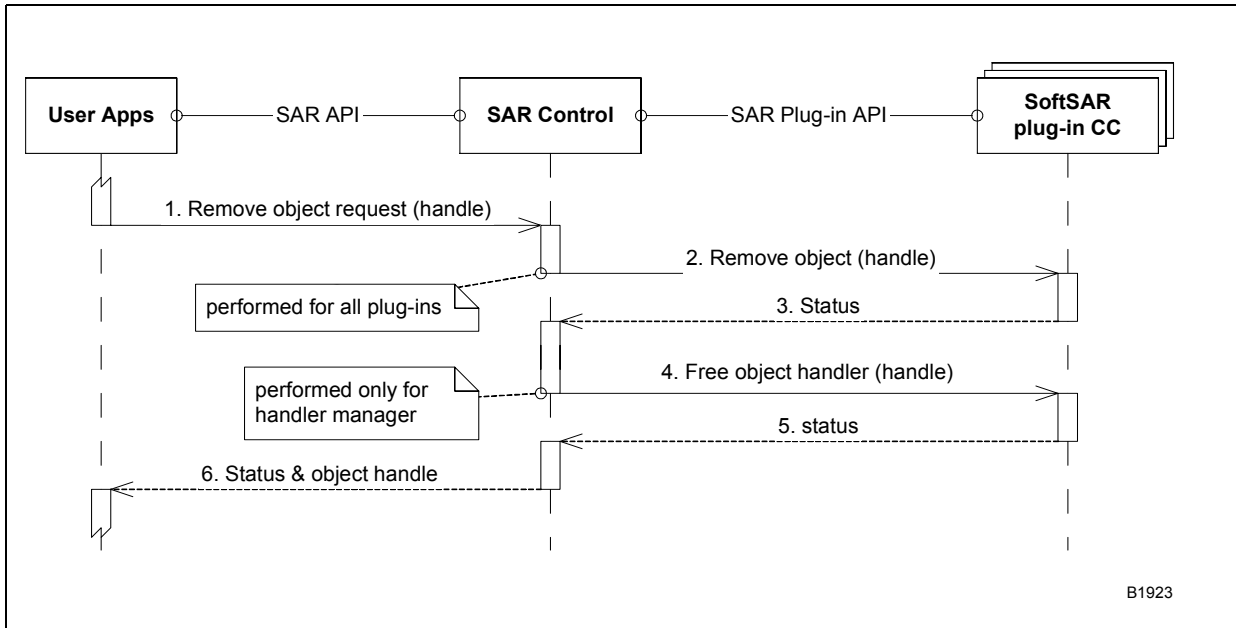
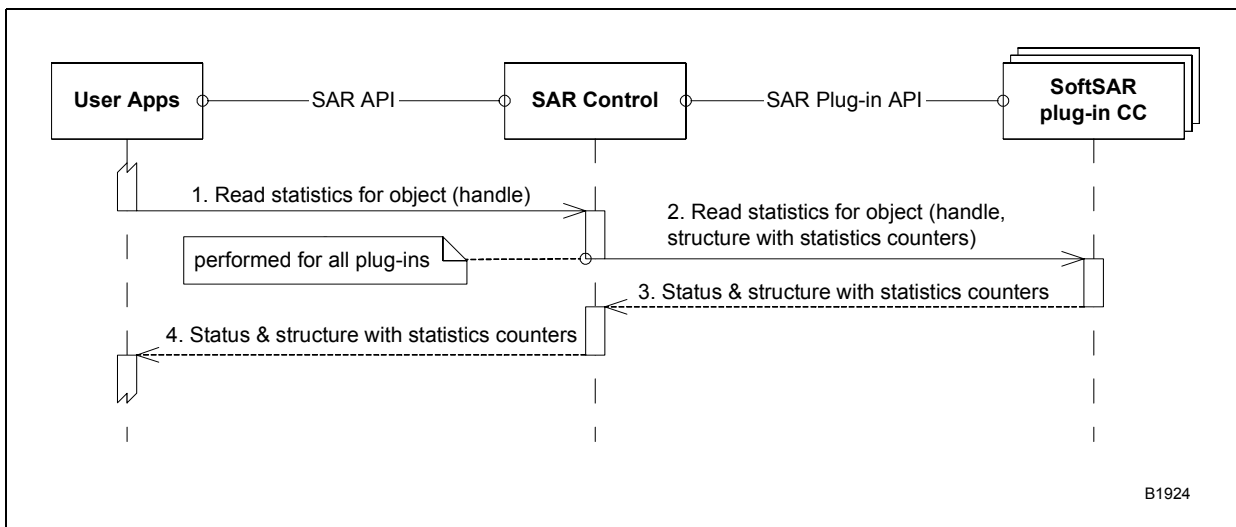


Figure 65-11. Read Statistics Operation Data Flow



65.7.2 SAR Control Main Core Component

65.7.2.1 Functionality

The SoftSAR core components provide the following functions:

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

65.7.2.2 Assumptions and Dependencies

65.7.2.2.1 Assumptions

The following is assumed:

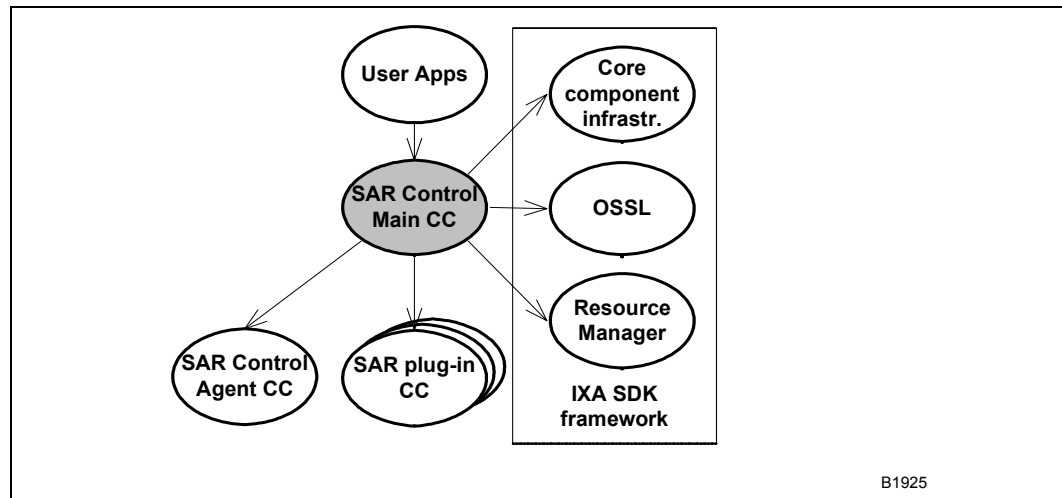
- Support for single and dual-IXP hardware configurations
- Compliance with the IXA SDK framework.

65.7.2.2.2 Dependencies

The SAR Control depends on the IXA SDK framework. It uses the services of Resource Manager (*Intel[®] Internet Exchange Architecture Portability Framework Developer's Manual*) for allocating and freeing DRAM/SRAM memory, and accessing system registry to retrieve static parameters (properties). The Core Components Infrastructure services (*Intel[®] Internet Exchange Architecture Portability Framework Reference Manual*) are used for message and packet handling between other core components. The Operating System Service Layer (OSSL) makes the core component code independent of the underlying OS.

The core component routes user data requests from user applications to the rest of SoftSAR core components (egress and ingress block), collects the result of operations and returns it to the calling application.

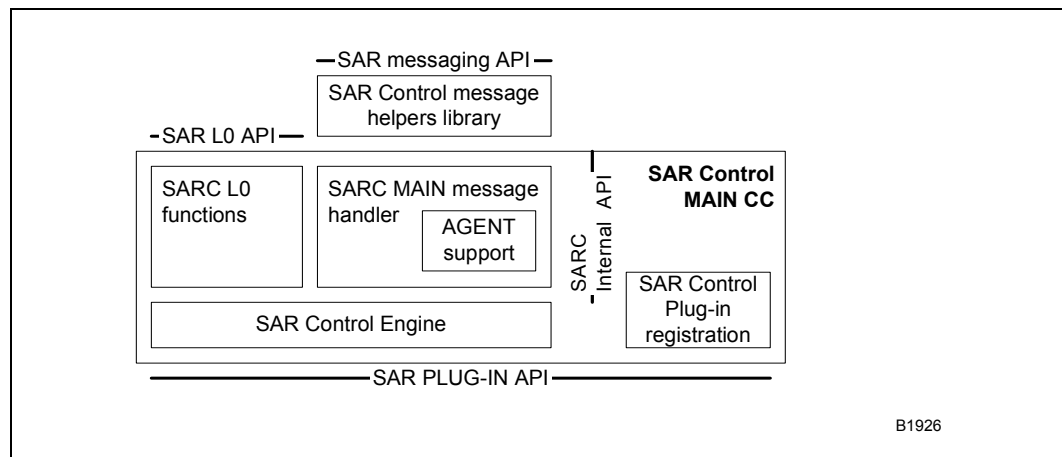
Figure 65-12. SoftSAR Main Core Components Dependencies



65.7.2.3 Decomposition

Figure 65-13 shows the SAR Control Main core component decomposition. The core component is accessible either via messaging API (with help of the message helpers library) or via the library API.

Figure 65-13. SAR Control MAIN Core Components Decomposition



65.7.2.4 Data and Control Flow

65.7.2.4.1 Dynamic Behavior

Each function from SAR API has a corresponding function in the SAR Plug-in API. When a calling/user application calls any function from the SAR API, the SAR Control calls all the required SoftSAR core components to perform the requested action. The calling order of the core components for each function is described in [Section 65.7.2.4.4, “The Calling Order of SoftSAR Core Components”](#) on page 938.

The following groups of actions are initiated from SAR API:

- Add—adding VC, CID or port operations. If any of the core components returns error status, the adding action is abandoned and the delete operation has to be performed for the added element.
- Update—updating VC, CID or port operations. If any of the core components returns error status, the update action is abandoned.
- Delete—deleting VC, CID or port operations. If any of the core components returns error status, the deleting operation must continue and the remaining SoftSAR core components must be called as well.
- Reading status and statistics—VC and port level.
- Exception and data packet handling—VC level.

65.7.2.4.2 The VC Queue Number Management

The pool of VC queue number is managed by the main sub-component of SAR Control. A VC queue number (VCQ#) is allocated at the beginning of the VC adding action. The VCQ# is subsequently used in all interactions with the rest of SoftSAR core components. During VC delete operation, the VCQ# is freed as the last action in the whole operation.

65.7.2.4.3 Synchronization

Each SoftSAR building blocks manages its own data. Egress and ingress part of SoftSAR are independent. But within a single processor (egress or ingress) the configuration activities should be done in specific order. The wrong sequence of configuration calls causes problems.

65.7.2.4.4 The Calling Order of SoftSAR Core Components

Figure 65-14 shows the convention defining an order of SoftSAR building blocks within a single data flow direction (on single or dual network processor unit).

Figure 65-14. Block Ordering Convention

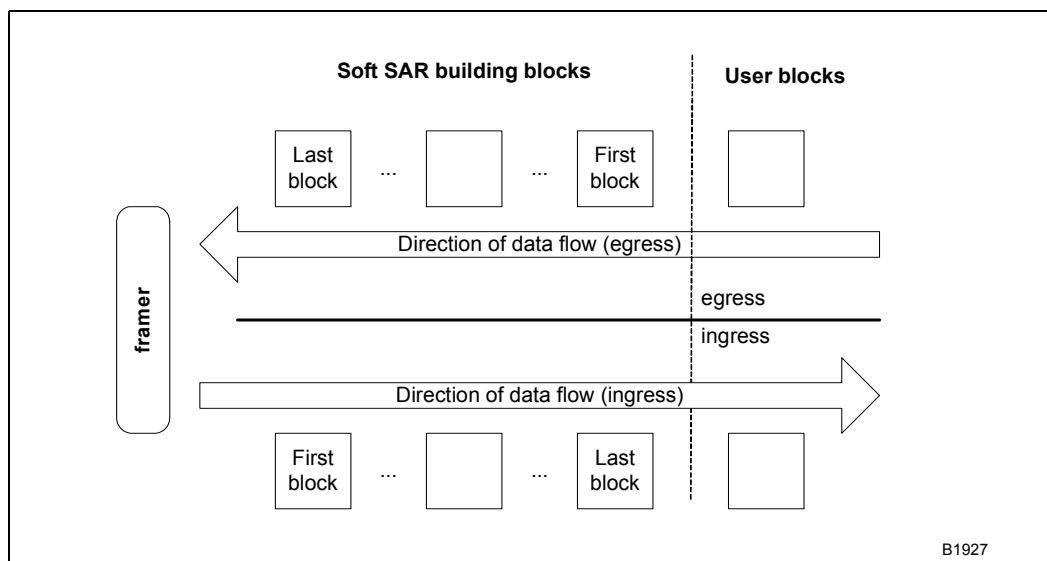


Table 65-1 shows the general rules for calling sequences for different groups of actions (see Section 65.7.2.4.1, “Dynamic Behavior” on page 937). The rules define the block in the whole path that should be called as first.

Table 65-1. General Rules for Calling Sequences

	Ingress path	Egress path
Deleting action	First	First
Adding action	Last	Last
Status reading	First	First

65.7.2.5 Configuration and Initialization

The SAR Control MAIN core component supports multiple SAR instances. The name of its configuration section in the system repository is “SAR_#instance”, where “#instance” is the instance number (for example, “SAR_0”). Table 65-2 shows the other configuration items.

Table 65-2. SAR Control MAIN Configuration Items

Property Name	Default value	Usage Description
AAL2_INCLUDED	0	The SAR supports (allows configuration) AAL2 traffic. The possible values are 0 (AAL2 not supported) and 1 (AAL2 supported).
MAIN_CC_LIST		The string variable contains list of core components that should be registered with SAR CONTROL MAIN. If any of the core components core components is not registered the SAR Control MAIN does not process a request.

65.7.3 External API

This section lists the external API for Queue Manager core component. For complete details, see Chapter 28, “SoftSAR” of the *IXA Software Building Blocks Reference Manual*.

65.7.3.1 Data Structures

Table 65-3 lists the data structures and the data type definitions supported by AAL types for VC.

Table 65-3. SAR Data Structures and Data Type Definitions

Data Structures and Data Types	Description
VC-related Data Structures	Defines VC related data type
<code>ix_cc_atmsar_aal_t</code>	Defines VC related enumeration
<code>ix_cc_atmsar_vc_spec_t</code>	Defines VC identification parameters
<code>ix_cc_atmsar_service_class_t</code>	Defines supported service classes at VC creation
<code>ix_cc_atmsar_traffic_param_t</code>	Defines traffic parameters used during add VC
<code>ix_cc_atmsar_vc_descr_t</code>	Defines all VC parameters required to add VC
<code>ix_cc_atmsar_update_data_t</code>	Defines all VC parameters required for VC update

Table 65-3. SAR Data Structures and Data Type Definitions (Continued)

Data Structures and Data Types	Description
<code>ix_cc_atmsar_vc_handle_t</code>	Defines VC identification that is used in the system
Port-Related Data Structures	Defines port related data structures
<code>ix_cc_atmsar_port_bandwidth_id_t</code>	Defines speed/bandwidth of port that can be set
<code>ix_cc_atmsar_port_descr_t</code>	Defines all port parameters required to add port

65.7.3.2 Core Component Infrastructure API

Table 65-4 lists the SAR core component supported Core Component Infrastructure APIs.:

Table 65-4. SAR Core Component Infrastructure API

Name	Description
<code>ix_cc_atmsar_init()</code>	Initialize the core component
<code>ix_cc_atmsar_fini()</code>	Terminate the core component
<code>ix_cc_atmsar_msg_handler()</code>	Message handler for processing rule add/remove requests
<code>ix_cc_atmsar_pkt_handler()</code>	Packet handler for processing exception packets

65.7.3.3 Messaging API

The Messaging API uses the Message Helper wrapper library that facilitates sending messages to the SoftSAR core component. One-to-one mapping exists between Message Helper APIs and the message types supported by a core component. All API functions are asynchronous; a core component reports operation status in a callback routine. [Table 65-5](#) lists the SAR Messaging API.

Table 65-5. SAR Messaging API

API Function	Description
<code>ix_cc_atmsar_async_vc_create()</code>	Adds VC
<code>ix_cc_atmsar_async_vc_update()</code>	Updates VC
<code>ix_cc_atmsar_async_vc_remove()</code>	Deletes VC
<code>ix_cc_atmsar_async_port_create()</code>	Adds port
<code>ix_cc_atmsar_async_port_remove()</code>	Deletes port
<code>ix_cc_atmsar_async_get_vc_stats()</code>	Gets VC statistics
<code>ix_cc_atmsar_async_get_port_stats()</code>	Gets port statistics

65.7.3.4 Library API

The Library API provides direct C-style calls to the SAR Control core component. The following library functions are provided. [Table 65-6](#) lists the SAR library APIs.

Table 65-6. SAR Library API

API Function	Description
<code>ix_cc_atmsar_vc_create()</code>	Add VC
<code>ix_cc_atmsar_vc_update()</code>	Update VC
<code>ix_cc_atmsar_vc_remove()</code>	Delete VC
<code>ix_cc_atmsar_port_create()</code>	Add port
<code>ix_cc_atmsar_port_remove()</code>	Delete port
<code>ix_cc_atmsar_get_vc_stats()</code>	Get VC statistics
<code>ix_cc_atmsar_get_port_stats()</code>	Get port statistics

65.7.3.5 Plug-in API

[Table 65-7](#) lists the APIs used internally in SAR core components to establish communication with the plug-in core components. All API functions are synchronous. On return, they report operation status in the return code.

Table 65-7. SAR Control Plug-in API

API Function	Description
<code>ix_cc_atmsar_plugin_get_cfg_params()</code>	Gets configuration parameters for the SAR instance
<code>ix_cc_atmsar_plugin_reg_vc_service()</code>	Registers plug-in for servicing VCs
<code>ix_cc_atmsar_plugin_reg_port_service()</code>	Registers plug-in for servicing ports
<code>ix_cc_atmsar_plugin_reg_vc_handle_service()</code>	Registers plug-in for servicing VC handles
<code>ix_cc_atmsar_plugin_reg_port_handle_service()</code>	Registers plug-in for servicing port handles
<code>ix_cc_atmsar_plugin_reg_done()</code>	Confirms that plug-in registering is completed

65.8 SAR Control Agent Core Component Design

The SAR Control Agent core component provides the following functionality:

- Utilization of SAR Plug-in API.
- Provides front-end SAR API for user modules/calling applications.

65.8.1 Assumptions and Dependencies

65.8.1.1 Assumptions

The following is assumed:

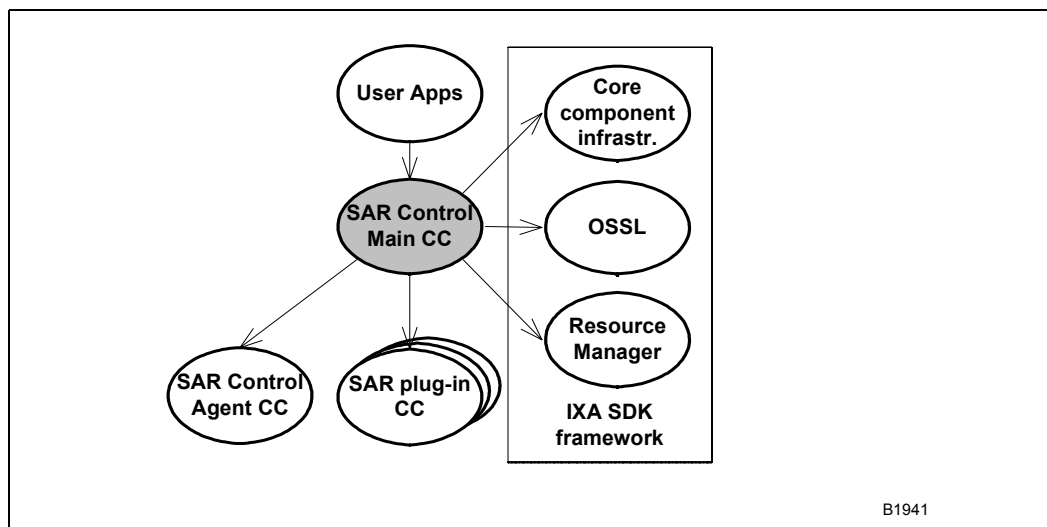
- Support for single- and dual-IXP HW configurations
- Compliance with the IXA Portability Framework.

65.8.1.2 Dependencies

The SAR Control agent depends on the IXA Portability Framework. It uses the services of Resource Manager (see *Intel® Internet Exchange Architecture Portability Framework Developer's Manual*) for allocating and freeing DRAM/SRAM memory, and accessing system registry to retrieve static parameters (properties). The Core Components Infrastructure services (*Intel® Internet Exchange Architecture Portability Framework Reference Manual*) are used for message and packet handling between other core components and the agent. The Operating System Service Layer (OSSL) makes the core component code independent of the underlying OS.

The core component routes calling application data requests from the calling applications to rest of SoftSAR core components (egress and ingress block), collects result of operations and returns it to user application.

Figure 65-15. SoftSAR Agent Core Components Dependencies

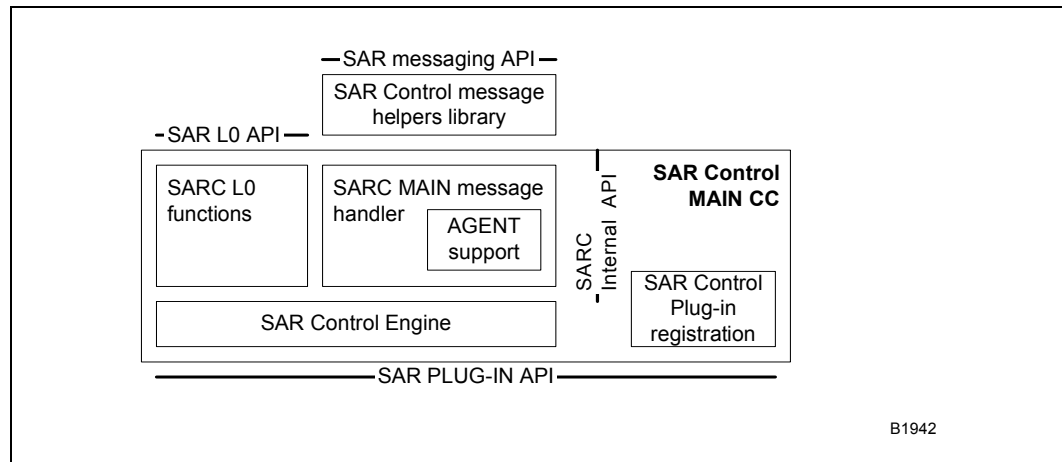


The SAR Control core component controls and services all SoftSAR core components. It provides front-end API for user applications and consumes API of core components that constitute SoftSAR functionality.

65.8.2 Decomposition

Figure 65-16 shows the SAR Control Agent core components decomposition. The core components is accessible remotely from SAR Control Main core components via SARC Internal API. The communication with plug-in core components is identical like in the case of SAR Control Main core components.

Figure 65-16. SAR Control Agent Core Component Decomposition



65.8.3 Data and Control Flow

See [Section 65.7.2.4, “Data and Control Flow”](#) on page 937.

65.8.4 Configuration and Initialization

The SAR Control agent core component supports multi instances. The name of its section in the system repository is “SAR_#instance”, where “#instance” is instance number. The example of name is “SAR_1”. This is the same as in SAR Control main core component ([Section 65.7.2.5, “Configuration and Initialization”](#) on page 939) and the agent core component shares few configuration variables with the main core component.

Table 65-8. Configuration Parameters

Property Name	Default value	Usage Description
AAL2_INCLUDED	0	The SAR supports (allows configuration) AAL2 traffic. The possible values are 0 (AAL2 not supported) and 1 (AAL2 supported). The variable is shared with SAR Control MAIN core component.
AGENT_CC_LIST		The string variable contains list of core components that should be registered into SAR control agent. If any of the core component is not registered the SAR Control agent does not process a request.

65.9 ATM RX Core Component

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

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

For more information on the ATM RX microblock, see [Chapter 6, “ATM AAL5 RX Microblock.”](#)

65.9.1 Assumptions and Dependencies

65.9.1.1 Assumptions

The ATM RX microblock sends exception packets to the ATM RX Core Component. The neighboring core components cannot send any packets to the ATM RX core component. The design of this core component defines one input and one output for data packets in the system file `bindings.h`. Bindings and registration of the packet handler are done by the system application for this component (see [Chapter 40, “System Application”](#)).

65.9.1.2 Dependencies

The ATM RX core component uses the services of the IXA Portability Framework for:

- allocating memory
- freeing memory
- patching symbols
- 64-bit counter support
- accessing the system registry to retrieve static parameters.

The ATM RX Core Component uses the Core Components Infrastructure services to register its message handler. The ATM RX core component passes messages to its message handler using the Message Support Library (see [Chapter 63, “Message Helper and Support Library”](#)).

If the system registry is used, it is implemented by using some global definitions so that when the registry is not included in the user's application, the ATM RX core component obtains this static data from compile-time defined values, such as a header file. Additionally, the SAR Control core components consumes the ATM RX core component provided API.

65.9.2 Shared Data Structures

The following lists the tables and variables shared between microblocks and core component. Refer to :

- VC primary hash table [Section 6.4.3, “Hash Table” on page 113 of Chapter 6, “ATM AAL5 RX Microblock”](#)
It is configurable table. The content of table is changed when any VC is added / deleted.
- VC secondary hash table [Section 6.4.3, “Hash Table” on page 113 Chapter 6, “ATM AAL5 RX Microblock”](#)
It is configurable table. The content of table is changed when any VC is added/deleted.

65.9.3 Data flow

65.9.3.1 Data Input and Output

SoftSAR Core component usage defines the operations supported by the component:

- Add Port—no action required
- Delete Port—no action required. In debug mode, it can check that hash table contains any VC for the port
- Add VC—add entry to VC hash table - It uses a new algorithm for hashing that is defined in [Section 65.9.4.2, “Hash Mechanism for VC Searching” on page 947](#)
- Delete VC—remove entry from VC hash table. It uses a new algorithm for hashing that is defined in [Section 65.9.4.2, “Hash Mechanism for VC Searching” on page 947](#). Deleting VC should be performed carefully and not disturb traffic for other VCs. The algorithm is shown in [Section 65.9.3.2, “Synchronization” on page 945](#)
- Get VC statistics—return statistics for port
- Get Port Statistics—return statistics for port

65.9.3.2 Synchronization

The algorithm for removing VC operation is shown:

```
Find position of removed VC in primary and secondary hash table;
If (found entry is not last in primary and secondary hash table)
{
    Find last entry in primary or secondary hash table;
    Move the last entry into freed (found) entry;
}
Clear last entry in primary or secondary hash table;
If (the last bucket in secondary hash table does not have VC entries)
{
    Update previous secondary bucket or update primary bucket.
    Free the unused bucket
}
```

The ATM RX core component uses API function exposed by Core Component Infrastructure to send out packets (see the *Intel® Internet Exchange Architecture Portability Framework Developer's Manual*).

65.9.4 Configuration and Initialization

The ATM RX Core Component defines dynamic and static data to be used during initialization. This data must be set by the property master and system registry or, if a system registry is not present, within a component's own header file. This data includes configuration parameters to be patched into the microblock.

65.9.4.1 Static Configuration Data

The ATM RX core component defines the static configuration data listed in [Table 65-9](#). The data is obtained from the system registry during component initialization. If the system registry is not present, then the default values defined in the component's header file is used. Some of the data are configuration parameters to be patched into the microblocks.

Table 65-9. ATM RX Core Component Static Configuration Properties

Parameter	Default	Description
ATM_RX_MEV2_NUMBER	0x0000001 (ME0)	Microengine numbers to be allocated to the ATM RX microblock. Only bit [0:7] are valid for ME0-ME7. This parameter is required only when ATM is included.

If a system registry is not included in the system, the ATM RX Core Component uses the compile-time defined values from its header files for these parameters. Values in the parenthesis indicate the default values defined in the header file.

During initialization, the ATM RX Core Component also allocates and patches memory and symbols into the microblock. these symbols and memory include:

Table 65-10. ATM RX Core Component Patch Symbols

Symbol	Description
ATM_RX_RXC_HASH_BASE	A pointer to the base of the hash table. This hash table is used to look up the receive context for a given virtual circuit. Memory allocation and initialization for this hash table is also performed.
ATM_RX_COUNTERS_BASE	The base of the counter table. Only one base if the counter table is patched in.

Memory is allocated for the counters listed below. This is on a per-virtual circuit basis. The maximum number of VCs (IX_MAX_NUMBER_ATM_VCS) is currently 64K.

- The number of erroneous cells received according to the error check for the error in RSW.
- The number of erroneous cells received according to the CRC error check.
- The number of packets received exceeding the maximum AAL5 size (of 64 KB).

65.9.4.2 Hash Mechanism for VC Searching

The hash table mechanism is used for translation from port, VPI and VCI into VCQ#. Specifically, SoftSAR building blocks do not use “hash mechanism” built in IXP2x00 processor. For hashing, The CRC32 mechanism is used because it is a less processor-power consuming mechanism than the original hash mechanism and it does not need to task-switch operation for its execution.

Table 65-11 defines the key for hash table for up to 16 ports (channels).

Table 65-11. IXP2400 16 ports—Key for Hash Table

LW	Bits	Size	Name	Description
0	31:28	4		Reserved. It has to be zero.
0	27:24	4	Port	Input port maintained from SPI4 port
0	23:16	8	VPI	Virtual Path Identifier
0	15:0	16	VCI	Virtual Channel Identifier

Table 65-12 defines the key for hash table for requests up to 2048 ports.

Table 65-12. 2048 ports—Key for Hash Table

LW	Bits	Size	Name	Description
0	31:24	8		Reserved. It has to be zero.
0	23:16	8	VPI	Virtual Path Identifier
0	15:0	16	VCI	Virtual Channel Identifier
1	31:11	21	Reserved	Reserved. It has to be zero.
1	10:0	11	port	Input port maintained from SPI4 port, and FPGA port

Address of bucket in primary hash table is computed using the following algorithm:

```
address = table_base_addr + (result_of_crc32 & 0xFFFF) << 4
```

Index of entry (result of CRC) is multiplied by 16, because a size of bucket in the primary hash table has size of 16.

65.9.5 Startup/Shutdown

The startup/shutdown complies with the scheme described in [Section 65.7, “SoftSAR Core Components”](#) on page 933.

65.9.6 External API

This section lists the external API for ATM RX core component. For complete details, see [Chapter 28, “SoftSAR”](#) of the *IXA Software Building Blocks Reference Manual*.

65.9.6.1 Core Component Infrastructure API

Table 65-13 lists the Core Component Infrastructure API provided by ATM RX core components.

Table 65-13. Core Component Infrastructure API

API	Description
<code>ix_cc_atmrx_init()</code>	Initializes the core component
<code>ix_cc_atmrx_fini()</code>	Terminates the core component
<code>ix_cc_atmrx_msg_handler()</code>	Handles messages for interface state and statistics
<code>ix_cc_atmrx_pkt_handler()</code>	Packet handler for processing exception packets

65.10 ATM TX Core Component

The ATM TX Core Component performs the following functions:

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

There are distinct microblocks for MPHY-16 and for SPHY and MPHY-4. For information on these microblocks, as well as the ATM TX microblock, see [Chapter 9, “Packet TX for MPHY-16,”](#) [Chapter 8, “Packet TX for SPHY and MPHY-4,”](#) and [Chapter 11, “ATM AAL5 TX Microblock.”](#) For external APIs see SoftSAR Core Components of the *IXA Software Building Blocks Reference Manual*.

65.10.1 Data flow

SoftSAR Core component usage defines the operations supported by the component:

- Add Port—no action required
- Delete Port—no action required
- Add VC—Not supported in this release
- Update VC—Not supported in this release
- Delete VC—Not supported in this release
- Get VC statistics—return statistics for port
- Get Port Statistics—return statistics for port

65.10.2 Assumptions and Dependencies

65.10.3 Assumptions

The following assumptions are made:

- The ATM TX Core Component assumes that the ATM TX microblock does not send any exception packets and messages to the core.
- The neighboring core components cannot send any packets to the ATM core component for transmitting the ATM TX framer. If the neighboring core components need to transmit packets to the framer, they must be sent to the Queue Manager.
Because it neither receives nor sends packets, the ATM core component does not define any data input and output in the system file `bindings.h`.
- The ATM TX core component provides a configuration API for ATM TX interface media-related functions, such as enabling and disabling the ATM interface. The initialization of the ATM framer is also done by ATM TX core component.

65.10.4 Dependencies

The ATM TX core component uses the services of the IXA Portability Framework for:

- allocating memory
- freeing memory
- patching symbols
- 64-bit counter support
- accessing the system registry to retrieve static parameters.
- registering the message handler

The ATM TX core component uses the Core Components Infrastructure services for to register its message handler. The ATM/POS TX Core Component passes messages to its message handler using the Message Support Library (see [Chapter 63, “Message Helper and Support Library”](#)).

The ATM TX core component uses IXA Portability Framework's system repository to retrieve static parameters. Use of the system repository is implemented by using global definitions (such as `#define IX_INCLUDE_REGISTRY 1`) so that when the repository is not included in user's application, the ATM TX core component can obtain the static data from compile time defined values, such as a header file.

The ATM TX core component utilizes the API provided by the framer device driver to initialize and configure the IXF6048 ATM TX framer device based on the static configuration data obtained from either the system repository or the component's global header file, `ix_cc_atm_pos_tx.h`.

65.10.5 Configuration and Initialization

The ATM TX core component defines dynamic and static data to be during initialization. For dynamic properties where the ATM TX core component is the master, the data must be set by the System Application. For other data, the data must be set using the system repository or if the system registry is not present then, for static data, by one of the core component's header files. This data includes configuration parameters to be patched into the microblock.

65.10.5.1 Dynamic Configuration Data

The dynamic property shown in [Table 65-14](#) can be set at run time:

Table 65-14. ATM/POS TX Core Component Dynamic Configuration Data

Data	Default	Property Master	Description
Interface State	UP	ATM TX	Interface Up or Down

The ATM or POS interface state can be changed at run time by the Stack Driver.

65.10.6 Static Configuration Data

[Table 65-15](#) shows the The ATM TX core component defined static configuration data. This data is obtained from the system repository during the core component's initialization. If the system repository is not present, then the default values defined in the component's global header file, `ix_cc_atm_pos_tx.h`, are used. This data includes configuration parameters to be patched into the microblocks.

Table 65-15. ATM Interface TX Configuration Parameters

Data	Default	Description
ATM_TX_MEV2_NUMBER	0x0000060 (ME5, ME6)	Microengine numbers to be allocated to ATM TX microblock. Only bit [0:7] are valid for ME0 - ME7. This data is required only when ATM mode is selected.

65.10.7 Startup/Shutdown

The startup/shutdown complies with the scheme described in [Section 65.7, “SoftSAR Core Components”](#) on page 933.

65.10.8 External API

This section lists the external API for ATM TX core component. For complete details, see [Chapter 28, “SoftSAR”](#) of the *IXA Software Building Blocks Reference Manual*.

65.10.8.1 Core Component Infrastructure API

[Table 65-16](#) lists the Core Component Infrastructure API provided by the ATM TX core component.

Table 65-16. ATM TX Core Component Infrastructure API

API	Description
<code>ix_cc_atmtx_init()</code>	Initializes the core component
<code>ix_cc_atmtx_fini()</code>	Terminates the core component

65.11 TM4.1 Core Component

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

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

For information on the TM4.1 microblock, see [Chapter 23, “TM4.1 Shaper and Scheduler Microblock”](#). For external APIs see SoftSAR Core Components of the *IXA Software Building Blocks Reference Manual*.

65.11.1 Assumptions and Dependencies

The TM4.1 core component provides API that is consumed by SAR Control core component.

65.11.2 Shared Data Structures

[Table 65-17](#) shows a list of tables and variables shared between microblocks and core component. Each of them are described with focus on changes of format, configuration and initialization.

Table 65-17. List of Shared Tables

Tables	Description
VC Queue Descriptor	It is configurable table
	see Chapter 23, “TM4.1 Shaper and Scheduler Microblock” .
	The initialization and symbol patching for the table is design in current design.
High bit rate (HBR) VC table	see Chapter 23, “TM4.1 Shaper and Scheduler Microblock”]
	It is configurable table. The table is changed when HBR VC is added / deleted.
Port shaping table	see Chapter 23, “TM4.1 Shaper and Scheduler Microblock”
	It is a new configurable table.
	It requires initialization and symbol patching.
	The table should be prepared / filled with special algorithm that guarantees even flow off of traffic for each port. The issue is discussed in separate section.
Port Info table	see Chapter 23, “TM4.1 Shaper and Scheduler Microblock”
	It is a new configurable table.
	It requires adding design for initialization and symbols patching.
Departure queues table	see Chapter 23, “TM4.1 Shaper and Scheduler Microblock”
	It is a run-time table. It does not require any configuration activities after initialization.
	It requires adding design for initialization and symbol patching.

Table 65-17. List of Shared Tables (Continued)

Tables	Description
Real-time time queues (RTTQ) table	see Chapter 23, “TM4.1 Shaper and Scheduler Microblock”
	It is a run-time table. It does not require any configuration activities after initialization.
	The initialization and symbol patching for the table is design in current design.
Non real-time queue (NRTTQ) table	see Chapter 23, “TM4.1 Shaper and Scheduler Microblock”
	It is a run-time table. It does not require any configuration activities after initialization.
	The initialization and symbol patching for the table is design in current design.
UBR time queue table	see Chapter 23, “TM4.1 Shaper and Scheduler Microblock”
	It is a run-time table. It does not require any configuration activities after initialization.
	The initialization and symbol patching for the table is design in current design.

65.11.3 Data and Control Flow

Dynamic behavior (configuration issues) and data flow is unchanged.

65.11.3.1 Support for Port Shaping Table

General rules for the table preparation are described in [Section 6.1.5, “Design Overview” on page 106 of Chapter 6, “ATM AAL5 RX Microblock”](#). The base requirement for inserting entries for a particular port is that the step between all entries must be the same. Multiplying the step and the number of entries for port must be equal to the size of port shaping table. The table size has a limitation and is equal to 2^n where “n” is an integer. The same limitation applies to the number of entries for port—the number has the same characteristics, 2^n .

This section describes an algorithm for setting entries in the table. The algorithm takes advantage of the correlation between first position for setting and entries already inserted into table for previous ports. To find first free position, algorithm swaps bits of number of entries already put into table. The number of bits swapped is equal to “n” where “n” is exponent for 2 from the size of port shaping table. The algorithm details are presented as pseudo C code

```

// Exponent for port shaping table
#define PORT_SHAPING_TABLE_EXP 11

// 2048 - port shaping table size
#define PORT_SHAPING_TABLE_SIZE (2 << PORT_SHAPING_TABLE_EXP)
int portShaperTab[PORT_SHAPING_TABLE_SIZE];

// local variables for the algorithm.
int i, j, k, pos, entriesCount=0, entries;

// clear port shaper table
memset(portShaperTab, 0, sizeof(portShaperTab));

// process list of ports. The list of ports is sorted with decreasing,
// where key is number of entries for port.
for(i=0; sortedPortList[i] != 0; i++)
{
    // Find first position for setting new port into port shaping table
    entries=sortedPortList[i].entries;
    for(j=0, pos = 0, k=entriesCount; j< PORT_SHAPING_TABLE_EXP; j++)
    {
        pos <<=1;
        if((k & 1) != 0) pos++;
        k >>=1;
    }

    // find step between entries in port shaping table
    k= PORT_SHAPING_TABLE_SIZE/entries;

    // fill the table
    for(j=0; j<entries; j++, pos+=k)
    {
        pos%=PORT_SHAPING_TABLE_SIZE;
        if(portShaperTab[pos] != 0)
            error - exit from algorithm;
        portShaperTab[pos]= sortedPortList[i].portNumber;
    }
    entriesCount+=entries;
}

```

The algorithm works only if the application fills an empty port shaping table during initialization. If dynamic run-time port add/delete are required, the core component needs additional algorithms for servicing pools of entries for reuse.

65.11.3.2 Support for HBR VC

The scheduler core component must fill the HBR TQ table but it has to take into consideration the influence of the portShaping table on the HBR TQ table. (See [Section 23.8, “2048 Ports Hierarchical Port-rate Shaping”](#) on page 384 of [Chapter 23, “TM4.1 Shaper and Scheduler Microblock”](#)).

The HBR TQ table has size 128 entries. The portShaping table has size 2048. The HBR TQ table maps into portShaping table 16 times. For allowing setting any HBR VC the port must have at least 16 entries set into portShaping table. For synchronization, HBR TQ and portShaping tables issue are taken first 128 entries of portShaping table, for the rest of entries of port shaping table the result of synchronization is propagated. The number of entries for the particular port is defined as 2^n , where “n” is an integer (it is result of limitations described in [Section 65.11.3.1, “Support for Port Shaping Table”](#) on page 952). The number of entries that can be use in HBR TQ table is equal entries for port in port shaping table divided by 16. And, the entries for the port in HBR TQ match to entries for port at first 128 entries of port shaping table.

For controlling the synchronization, the core component uses the same algorithm as in [Section 65.11.3.1, “Support for Port Shaping Table”](#) on page 952.

65.11.4 Configuration and Initialization

The TM4.1 core components does the following actions:

- Allocates DRAM/SRAM memory for the hash table and initializes it with empty entries (all zeros).
- Patches TM4.1 microcode with imported variables. The function gets the variable values from the system repository properties.
- Registers a packet handler `ix_cc_atmtm4.1_pkt_handler()` for inputs:
- `IX_CC_ATMSAR_TM41_EXEPTION_PKT_INPUT` (single source mode)
- `IX_CC_ATMSAR_TM41_LOCAL_PKT_INPUT` (single source mode)
- Registers a message handler `ix_cc_atmsar_tm41_msg_handler()` for inputs:
- `IX_CC_ATMSAR_TM41_MSG_INPUT` (multiple sources mode)
- Allocates a control block, and returns a pointer to this block in `arg_ppContext`.

65.11.5 Startup/Shutdown Discussion

The startup/shutdown complies with the scheme described in [Section 65.7, “SoftSAR Core Components”](#) on page 933.

65.11.6 External API

This section lists the external API for TM4.1 core component. For complete details, see Chapter 28, “SoftSAR” of the *IXA Software Building Blocks Reference Manual*.

65.11.6.1 Core Component Infrastructure API

Table 65-18 lists the TM4.1 core component supported the Core Component Infrastructure APIs

Table 65-18. TM4.1 Core Component supported Core Component Infrastructure API

API function	Description
<code>ix_cc_atmtm41_init()</code>	Initializes the core component
<code>ix_cc_atmtm41_fini()</code>	Terminates the core component
<code>ix_cc_atmtm41_msg_handler()</code>	Handles messages for processing rule add/remove requests
<code>ix_cc_atmtm41_pkt_handler()</code>	Handles packets for processing exception packets

MPLS Components

The following chapter is included in this section:

- [Chapter 66, “MPLS Forwarder Core Component”](#)

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

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

- Initializes and maintains the data structures for the ILM Forwarder and FTN Forwarder microblocks
- Configures the ILM Forwarder and FTN Forwarder microblocks (static configuration)
- Provides message and packet handlers to receive messages from the control plane and packets from the ILM Forwarder and FTN Forwarder microblocks
- Handles fragmentation of MPLS packets
- Implements "slow path" forwarding for fragmented MPLS packets
- Handles packets with special MPLS label values

MPLS Forwarder Core Component 66

66.1 Overview

The Multiprotocol Label Switching (MPLS) Forwarder core component works on behalf of the ILM Forwarder and FTN Forwarder microblocks.

It performs the following functions:

- Initializes and maintains the data structures for the ILM Forwarder and FTN Forwarder microblocks
- Configures the ILM Forwarder and FTN Forwarder microblocks (static configuration)
- Provides message and packet handlers to receive messages from the control plane and packets from the ILM Forwarder and FTN Forwarder microblocks
- Handles fragmentation of MPLS packets
- Implements “slow path” forwarding for fragmented MPLS packets
- Handles packets with special MPLS label values

For complete details on the MPLS Microblocks, see [Chapter 35, “FTN Forwarder Microblock”](#) and [Chapter 36, “ILM Forwarder Microblock”](#). For external APIs, see [Chapter 29, “MPLS Forwarder”](#) in the *IXA Software Building Blocks Reference Manual*.

66.2 Data Flow

The MPLS forwarder core component defines a single packet input for receiving exception packets from the ILM Forwarder and FTN Forwarder microblocks. The input id is defined in the system file, `bindings.h` as follows:

```
#define IX_CC_MPLS_PKT_INPUT
```

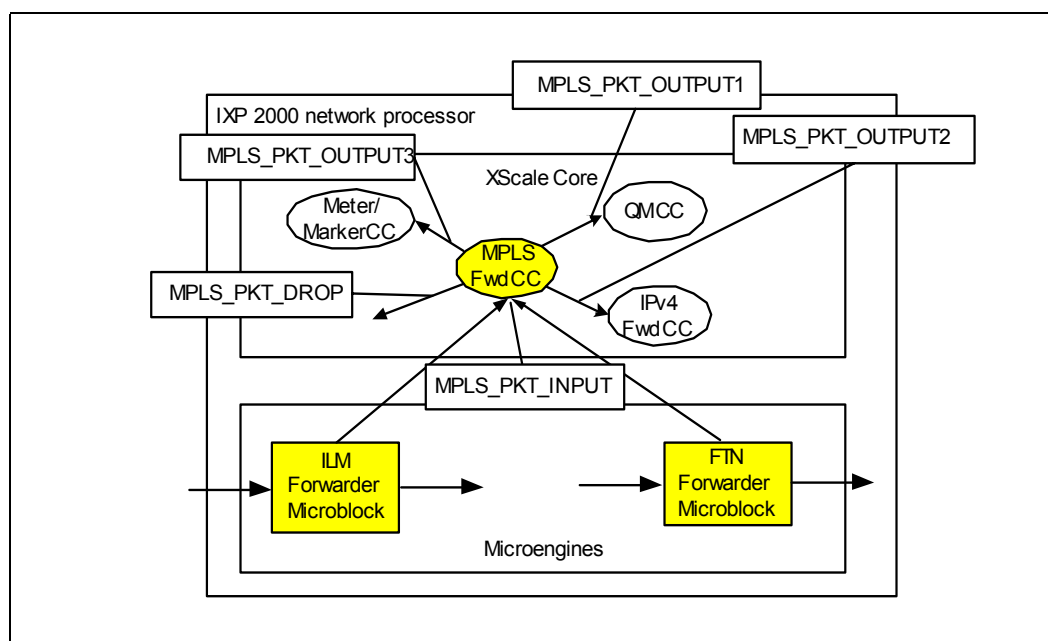
The MPLS core component defines four outputs for sending packets to the Queue Manager core component, IPv4 Forwarder core component, Meter / Marker core component, and for dropping packets. The output identifiers for outgoing packets are defined in the system’s `bindings.h` as:

```
#define IX_CC_MPLS_PKT_OUTPUT1/* QM output */
#define IX_CC_MPLS_PKT_OUTPUT2/* IPv4 output */
#define IX_CC_MPLS_PKT_OUTPUT3/* Meter/Marker output */
#define IX_CC_MPLS_PKT_DROP
```

The MPLS core component uses API functions exposed by the Core Component Infrastructure to send packets out.

Figure 66-1 illustrates the packet data flow for the MPLS Forwarder core component.

Figure 66-1. MPLS Forwarder Core Component Packet Data Flow



The MPLS core component defines one input (`IX_CC_MPLS_PKT_INPUT`) for receiving exception data packets from the ILM Classifier and FTN Forwarder microblocks.

According to the results of an exception packet processing, the MPLS core component can send packets out through one of the following outputs:

- `IX_CC_MPLS_PKT_OUTPUT1`—if a packet is further forwarded as an MPLS packet, this output should be bound to the Queue Manager core component input. This output is also used for ICMP error messages generated by the MPLS core component.
- `IX_CC_MPLS_PKT_OUTPUT2`—if, after popping all labels, a packet is passed to the IPv4 forwarder core component for further forwarding.
- `IX_CC_MPLS_PKT_OUTPUT3`—if an MPLS packet, after processing in the MPLS core component, needs to be further processed by the Meter/Marker core component.
- `IX_CC_MPLS_PKT_DROP`—if a packet cannot be forwarded (for example, its length exceeds the output port `maxLabPktSize` parameter value and it cannot be fragmented).

66.3 Assumptions and Dependencies

66.3.1 Assumptions

The implementation of the MPLS Forwarder Core Component assumes the following:

- The MPLS Forwarder Core Component operates in the environment defined by the Core Component Infrastructure. This infrastructure allocates individual core components to certain execution engines, also called threads. To avoid synchronization problems in accessing shared data structures, the MPLS Forwarder Core Component should be placed in the same execution

engine as the IP Forwarder Core Component, because it assumes sharing the IP Routing table with the IPv4 Forwarder.

- The MPLS application design assumes an MPLS-aware IPv4 routing table. The MPLS-awareness means that IPv4 routing table entries can be populated both by the IPv4 Forwarder Core Component and the MPLS Forwarder Core Component. In case of a match, the routing table lookup result specifies both the next block to execute and a next hop ID value. If the next block to be executed is the MPLS Forwarder, the next hop ID has meaning specific to MPLS.
- This version of the MPLS application cooperates only with the IPv4 Forwarder Core Component and microblocks.

66.3.2 Component Dependencies

The MPLS core component uses the services of the IXA Portability Framework for allocating memory, freeing memory, patching symbols, 64-bit counter support and optional system registry to retrieve static parameters. The Core Components Infrastructure services are used for event handling, message handling, and packet handling between the MPLS core component and ILM Forwarder and MPLS Marker microblocks. For more information on the IXA Portability Framework and the Core Component Infrastructure, refer to the *Intel® Internet Exchange Architecture Portability Framework Developer's Manual*.

66.3.3 IPv4 and MPLS Core Component Cooperation

An MPLS ingress LER forwarder assigns incoming packets to forwarding equivalence classes (FECs) based on their IP addresses. To avoid double lookup, it cooperates with the IPv4 Forwarder, which performs the longest prefix match (LPM) on an incoming packet, and according to the lookup result, either processes the packet or passes it to the MPLS Forwarder.

The above implies that the IPv4 Core Component exposes the API allowing the MPLS Core Component to modify the IPv4 route table and next hop database (NHD). Due to placing the IPv4 and MPLS core components in the same execution engine, the MPLS Core Component can use the IPv4 Route Table Manager Library functions for maintaining the next hop information and IP routes belonging to MPLS.

The RTM Library accepts the MPLS core component as its second client, in addition to the IPv4 Core Component. First, the IPv4 Core Component performs the RTM initialization, and then the MPLS Core Component obtains the RTM handle by calling the `ix_cc_ipv4_get_rtm_handle()` function.

To minimize changes to the IPv4 forwarder, the LPM for both IP-to-IP domain and IP-to-MPLS domain routes returns a handle to a NHD entry. The NHD is IP-specific - it cannot directly accommodate the information needed for MPLS forwarding. Therefore, the MPLS Core Component uses the `l2Index` field of an NHD entry as an index into the MPLS NHLFE table containing all MPLS forwarding data. The IPv4 forwarder discriminates IP- and MPLS-originated NHD entries by the value of `l2IndexType` field.

A generic MPLS ingress LER injects IP packets into an MPLS domain based on two tables—the FTN (FEC to NHLFE map), and the NHLFE table. This implementation of the MPLS forwarder operates on a “conceptual” FTN table, using the IPv4 route table and IPv4 NHD for this purpose.

The trade-off of this solution is one extra memory access during ingress LSP lookup—the sequence LPM-NHD-NHLFE could be reduced to LPM-NHLFE, if the IPv4 route table supported different types of `nextHopID` values.

66.3.4 Configuration and Initialization

The MPLS forwarder Core Component defines some static and dynamic data to be used during initialization time. These data should be set properly by the control plane and the framework's registry, or a component own header file if the registry is not present in the system. Some of these data are configuration parameters to be patched into the microblock.

66.3.4.1 Static Configuration Data

Table 66-1 shows the static configuration data defined by the MPLS Forwarder Core Component.:

Table 66-1. Static Configuration Data

Data	Default	Description
MPLS_MAX_NHLFE_ENTRIES	65536	The maximum number of NHLFE table entries.
MPLS_MIN_LABEL	16	The minimum incoming label value.
MPLS_MAX_LABEL	65536	The maximum incoming label value.
MPLS_PER_PLATFORM_LABEL_SPACE	TRUE	The per-platform label space.
MPLS_PER_INTERFACE_LABEL_SPACE	FALSE	The per-interface label space. This is mutually exclusive with the MPLS_PER_PLATFORM_LABEL_SPACE constant.
MPLS_SET_TABLE_SIZE	65536	The maximum number of entries in the ILM and NHLFE set tables.
MPLS_MAX_NHLFE_IN_SET	4	The maximum number of NHLFE entries in an NHLFE set.
MPLS_MAX_LOCAL_PORTS	4	The maximum number of ports per blade.
MPLS_MAX_PHB_NUM	4	The maximum number of PHB to EXP mappings.
MPLS_MAX_EXP_NUM	8	The maximum number of EXP to PHB mappings.

The static configuration data is retrieved either from the framework's system registry, if it is present, or from one of the component header files.

66.3.4.2 Dynamic Configuration Data

The MPLS Forwarder core component does not have any configuration data that can be changed at run time.

66.3.4.3 Patching Symbols

Table 66-2 shows the symbols and memory base addresses used by the MPLS Forwarder Core Component to patch the ILM Forwarder microblock code.

Table 66-2. Patching Symbols for ILM Forwarder Microblock

Symbols	Default	Description
MPLS_ILM_TABLE_BASE	-	The base address of the ILM table.
MPLS_ILM_STAT_BASE	-	The base address of the ILM statistics table (in Seg counters).
MPLS_ILM_NHLFE_SET_TABLE_BASE	-	The base address of the ILM_NHLFE set table.

Table 66-2. Patching Symbols for ILM Forwarder Microblock (Continued)

Symbols	Default	Description
MPLS_ILM_NHLFE_SET_STAT_BASE	-	The base address of the ILM_NHLFE set statistics table (inSeg counters).
MPLS_INPUT_PORT_STAT_BASE	-	The base address of the MPLS Ports Statistics table.
MPLS_PARAMS_BASE	-	The base address of the MPLS Parameters table.
MPLS_EXP2PHB_TABLE_BASE	-	The base address of the MPLS Exp2Phb table.
MPLS_PHB2EXP_TABLE_BASE	-	The base address of the MPLS Phb2Exp table.

Table 66-3 shows the symbols and memory base addresses used by the MPLS Forwarder Core Component to patch the FTN forwarder microblock code.

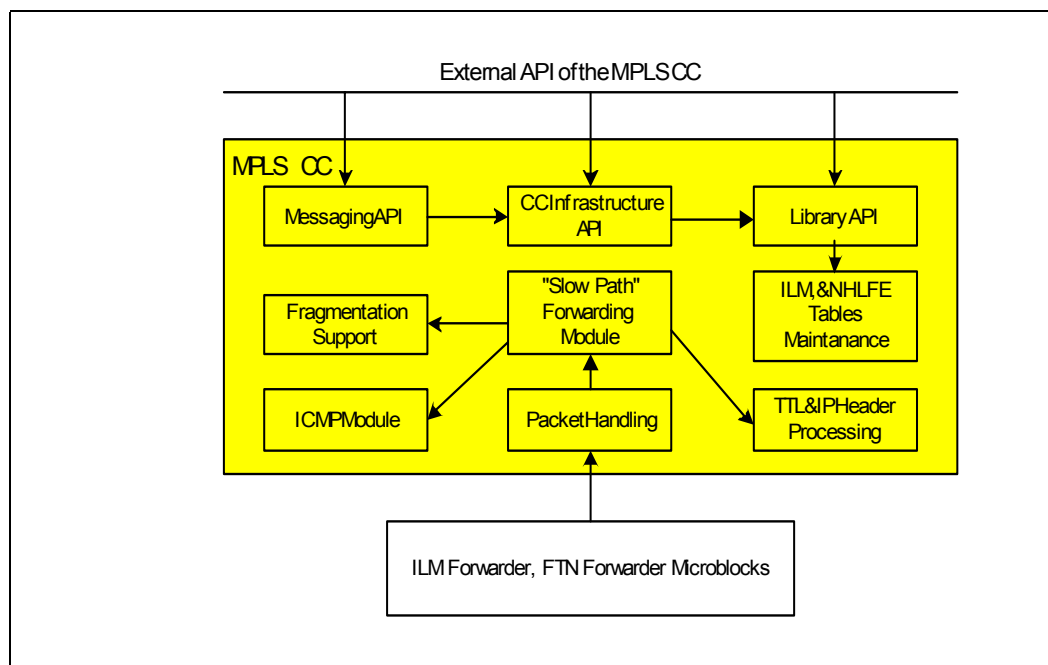
Table 66-3. Patching Symbols for FTN Forwarder Microblock

Symbols	Default	Description
MPLS_NHLFE_TABLE_BASE	-	The base address of the NHLFE table
MPLS_FTN_STAT_BASE	-	The base address of the FTN statistics table.
MPLS_NHLFE_SET_TABLE_BASE	-	The base address of the NHLFE set table.
MPLS_NHLFE_SET_STAT_BASE	-	The base address of the NHLFE set table.
MPLS_PARAMS_BASE	-	The base address of the MPLS Parameters table.
MPLS_EXP2PHB_TABLE_BASE	-	The base address of the MPLS Exp2Phb table.
MPLS_PHB2EXP_TABLE_BASE	-	The base address of the MPLS Phb2Exp table.

66.4 Modularity

Figure 66-2 illustrates the MPLS Core Component logical sub-modules and their interactions.

Figure 66-2. MPLS Core Component Logical Sub-modules and Interactions



- The Messaging API is implemented in the core component message helper library. It interfaces with the message handler defined in the core component Infrastructure API indirectly through the Message Support Library. The message handler function uses the Library API for retrieving data. Both the Infrastructure API and Library API use services from level-0 framework.
- The Packet Handling Module processes exception packets received from the ILM and FTN Forwarder microblocks. The microblocks send exception packets to the MPLS core component in the following circumstances:
 - The top label in the MPLS packet is the “Router Alert Label” (has the value of 1)
 - The MPLS packet's length exceeds the “Effective Maximum Frame Payload Size for Labeled Packets” of the output port
 - The MPLS packet's outgoing TTL value is lower than 1.
 - There are more than 3 consecutive label POP operations on a given MPLS packet.

The exception packets are passed to the “Slow Path Forwarding” module, which processes them as follows:

Packet Handling Module Process	Slow Path Forwarding Module Process
The top label in the MPLS packet is “Router Alert Label”	The packet is processed in the way described in [RFC3032].
The MPLS packet's length exceeds the “Effective Maximum Frame Payload Size for Labeled Packets” of the output port	If the MPLS payload is other than IP, the packet is dropped. Otherwise, the Packet Handling module executes the algorithms specified in [RFC3032]. As a result, an ICMP message “Destination Unreachable because fragmentation needed and DF set” may be generated and sent with the help of the ICMP module.
The MPLS packet's outgoing TTL value is lower than 1	An ICMP message “Time exceeded” is generated and sent with the help of the ICMP module.
There are more than 3 consecutive label POP operations on a given MPLS packet.	The packet forwarding is performed by the core component in order not to slow down the “fast path” in microblocks.

- The ICMP Module supports generation and sending of the “Destination Unreachable because fragmentation needed and DF set”, and “Time exceeded” ICMP messages. The MPLS Core Component forwards those messages as MPLS packets to the MPLS destination address, as specified in [RFC3032].
- The Fragmentation Support module performs fragmentation of IP packets on the request of the Packet Handling module. This module could be eliminated, if the IPv4 Core Component fragmentation support was available as a library function.
- The TTL&IP Header Processing module is used by the Slow Path Forwarding module for performing the necessary IP header modifications in case of penultimate hop popping.
- The ILM, NHLFE and MPLS Ports Table Maintenance module implements the functions operating on the ILM table and the NHLFE table. It uses the IPv4 core component API to implement IP FECs to NHLFE mappings by adding appropriate IPv4 FIB and Next Hop Table entries.

66.5 External API

This section lists the external API for MPLS Forwarder Core Component. For complete details, see [Chapter 29, “MPLS Forwarder”](#) in the *IXA Software Building Blocks Reference Manual*

66.5.1 Data Structures and Types

These structures are shared between the MPLS core component, the ILM Forwarder and the MPLS Marker microblocks. Additionally, the MPLS Forwarder core component operates on several specific data types. Table 66-4 lists the data structures and types described in the following sections.

Table 66-4. Data Types and Structures

Data Types and Structures	Description
<code>ix_cc_mpls_fec</code>	Specifies an MPLS Forwarding Equivalence Class
<code>ix_cc_mpls_label</code>	Specifies the control-plane level view of an MPLS label
<code>ix_cc_mpls_ilm</code>	Specifies the ILM key
<code>ix_cc_mpls_params</code>	Specifies the control-plane level view of MPLS parameters
<code>ix_cc_mpls_nhlfe</code>	Specifies the control-plane level view of an NHLFE table entry
<code>ix_cc_mpls_nhlfe_handle</code>	Defines the layout of an NHLFE handle
<code>ix_cc_mpls_cc_nhlfe</code>	Defines the layout of the CC_NHLFE table entry
<code>ix_cc_mpls_nhlfe_stats</code>	Specifies the control-plane view of per-NHLFE table entry counters
<code>ix_cc_mpls_cc_lsp</code>	Defines the layout of a CC_LSP table entry
<code>ix_cc_mpls_lsp_param</code>	Specifies the control-plane level view of LSP parameters
<code>ix_cc_mpls_lsp</code>	Specifies the control-plane level view of an LSP
<code>ix_cc_mpls_lsp_stats</code>	Specifies the control-plane view of a per-LSP statistics
<code>ix_cc_mpls_port_stats</code>	Specifies the control-plane view of an MPLS Ports Statistics Table entry

66.5.2 Core Component Infrastructure API

Table 66-5 lists the MPLS core component infrastructure API.

Table 66-5. MPLS Core Component Infrastructure API

API	Description
<code>ix_cc_mpls_init()</code>	Initialize the core component
<code>ix_cc_mpls_fini()</code>	Terminate the core component
<code>ix_cc_mpls_msg_handler()</code>	Message handler for forwarding tables maintenance, MPLS ports management, and statistics
<code>ix_cc_mpls_microblock_pkt_handler()</code>	Packet handler for processing exception packets from MPLS microblocks

66.5.3 Message Helper API

Table 66-6 lists the Message Helper API provided by the MPLS core component.

Table 66-6. MPLS Message Helper API

API	Description
<code>ix_cc_mpls_cb_general()</code>	Generic callback function
<code>ix_cc_mpls_async_lsp_create()</code>	Creates an LSP
<code>ix_cc_mpls_async_lsp_delete()</code>	Deletes an LSP
<code>ix_cc_mpls_async_lsp_modify()</code>	Modifies an LSP
<code>ix_cc_mpls_async_lsp_query()</code>	Reads an LSP table entry
<code>ix_cc_mpls_async_lsp_stats_query()</code>	Reads LSP statistics
<code>ix_cc_mpls_async_lsp_purge()</code>	Deletes all LSPs
<code>ix_cc_mpls_async_nhlfe_create()</code>	Creates an NHLFE
<code>ix_cc_mpls_async_nhlfe_delete()</code>	Deletes an NHLFE
<code>ix_cc_mpls_async_nhlfe_query()</code>	Reads an NHLFE table entry
<code>ix_cc_mpls_async_nhlfe_stats_query()</code>	Reads NHLFE statistics
<code>ix_cc_mpls_async_nhlfe_purge()</code>	Deletes all NHLFEs
<code>ix_cc_mpls_async_nhlfe_set_create()</code>	Creates an NHLFE set
<code>ix_cc_mpls_async_nhlfe_set_delete()</code>	Deletes an NHLFE set
<code>ix_cc_mpls_async_nhlfe_set_modify()</code>	Modifies an NHLFE set
<code>ix_cc_mpls_async_nhlfe_set_query()</code>	Reads an NHLFE set
<code>ix_cc_mpls_async_param_query()</code>	Reads MPLS parameters area
<code>ix_cc_mpls_async_port_stats_query()</code>	Reads MPLS port statistics

66.5.4 Library API

Table 66-7 lists the Library API provided by the MPLS core component.

Table 66-7. MPLS Library API

API	Description
<code>ix_cc_mpls_lsp_create()</code>	Creates an LSP
<code>ix_cc_mpls_lsp_delete()</code>	Deletes an LSP
<code>ix_cc_mpls_lsp_modify()</code>	Modifies an LSP
<code>ix_cc_mpls_lsp_query()</code>	Reads an LSP table entry
<code>ix_cc_mpls_lsp_stats_query()</code>	Reads LSP statistics
<code>ix_cc_mpls_lsp_purge()</code>	Deletes all LSPs
<code>ix_cc_mpls_nhlfe_create()</code>	Creates an NHLFE

Table 66-7. MPLS Library API (Continued)

API	Description
<code>ix_cc_mpls_nhlfe_delete()</code>	Deletes an NHLFE
<code>ix_cc_mpls_nhlfe_query()</code>	Reads an NHLFE
<code>ix_cc_mpls_nhlfe_stats_query()</code>	Reads NHLFE statistics
<code>ix_cc_mpls_nhlfe_purge()</code>	Deletes all NHLFEs
<code>ix_cc_mpls_nhlfe_set_create()</code>	Creates an NHLFE set
<code>ix_cc_mpls_nhlfe_set_delete()</code>	Deletes an NHLFE set
<code>ix_cc_mpls_nhlfe_set_modify()</code>	Modifies an NHLFE set
<code>ix_cc_mpls_nhlfe_set_query()</code>	Reads an NHLFE set
<code>ix_cc_mpls_param_query()</code>	Reads MPLS parameters
<code>ix_cc_mpls_port_stats_query()</code>	Reads MPLS port statistics

This appendix describes the Network Simulator DLL (NetSim) which is used in the SDK for generating IPv6 and MPLS test traffic on the simulator.

A.1 Overview

NetSim is a DLL that works in conjunction with the IXP Workbench simulation environment. It provides a mechanism for developers to generate test network traffic for their IXP simulation. It provides a flexible and extensible environment for developing input streams for a variety of protocols. It also allows the specification of tables that can be used for the generation of traffic.

On the IXA SDK 3.1, NetSim is used primarily for generating and verifying IPv6 and MPLS streams.

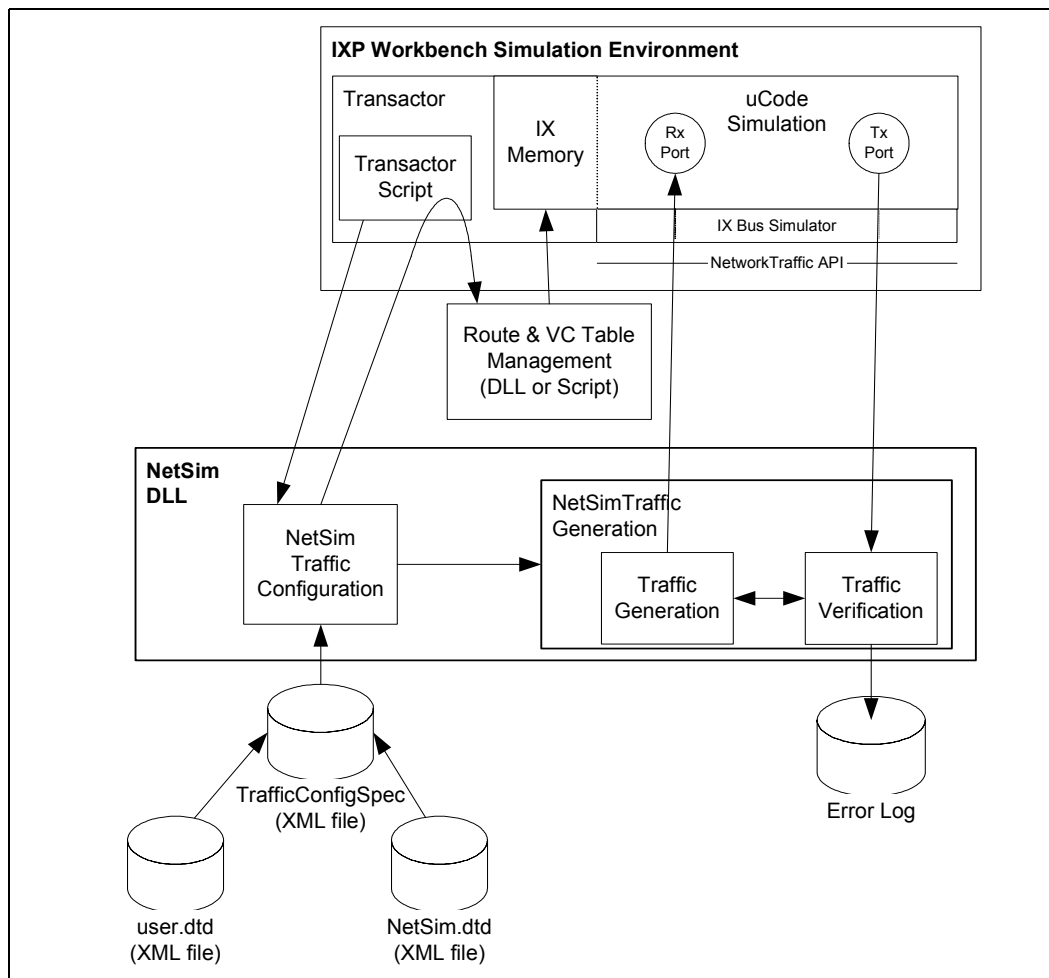
A.2 Architecture

The components that comprise NetSim are as follows:

- Traffic Configuration Specification (TCS)—this is a text file that acts as a central repository for all configuration data. This file is implemented as an XML schema and allows the specification of Table Entries, Stream definitions, and Port assignments. Within this file, data streams may be specified with user-defined addresses, or optionally, with addresses corresponding to the table entries.
- Netsim.dtd—as a general rule, a single NetSim.dtd file is included within every TCS file. This is an XML Data Type Definition file and is analogous to a C header file. It defines the parameters and structure of the TCS file.
- User-created XML Data Type Definition file (DTD)—this is an extension to Netsim.dtd that the user may create in order to add new table types and protocol header formats.
- NetSim DLL—this is a foreign model DLL that implements the IXP Workbench's Network Traffic Generation API. It parses the TCS file; interfaces with Script Functions or other Local Foreign Model DLL components to configure tables, and uses the Traffic Generation library to create input traffic.

Figure A-1 illustrates the Netsim framework architecture.

Figure A-1. NetSim Framework Architecture



A.2.1 Connecting NetSim to the IXP Workbench

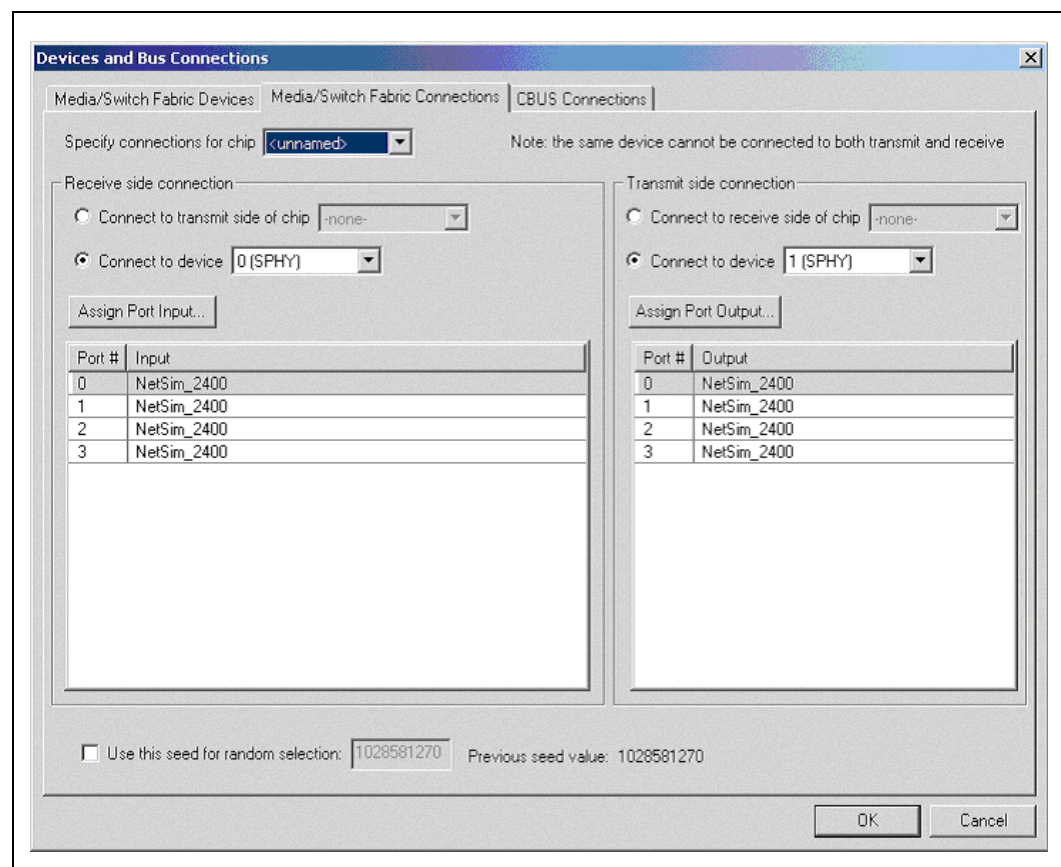
A.2.2 Assigning NetSim to an IXP Simulation Port

The NetSim DLL implements an API required by the IXP Workbench for simulating traffic on network ports.

To assign NetSim to an IXP simulation port:

- Select “Devices and Bus Connections”
- Select “Media/Switch Fabric Connections” to illustrate the dialog shown in [Figure A-2](#).
NetSim is allocated to all receive and transmit ports.

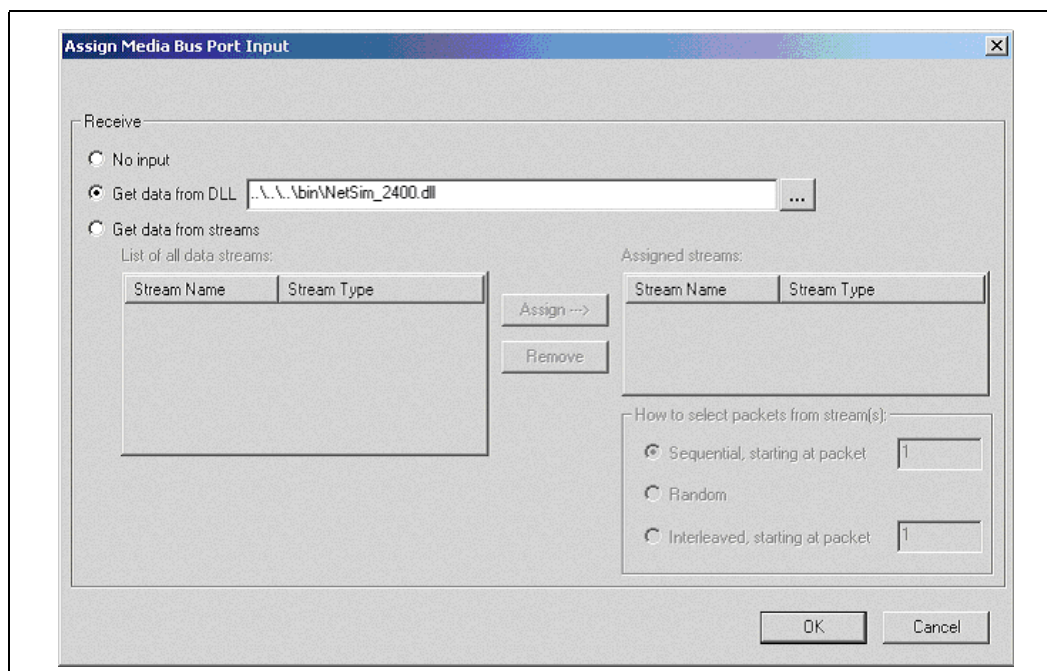
Figure A-2. Devices and Bus Connections—Media/Switch Fabric Connections



If NetSim is not allocated:

- Select the desired port to attach the NetSim DLL
- Click the “Assign Port Input” or “Assign Port Output” button in the upper left corner. The next dialog illustrated in [Figure A-3](#) is shown.

Figure A-3. Assign Media Bus Port Input



- Select “Get Data from DLL”
- Add the path to the NetSim DLL.

This connection enables the NetSim to generate and pass data streams to the workbench.

The Transmit side connection is similar. When assigned, this connects the NetSim to capture and verify frames when specified in the TCS file.

If it is desirable to have a project also to be able to get the port data from a Workbench Stream file, the NetSim may also be registered as a Foreign Model DLL. This avoids having to remove the NetSim function calls from the Script file when only the Stream file input is used.

A.3 Initializing NetSim

When the workbench is first started, NetSim is instantiated when the workbench queries its first port via the interface. The TCS file is loaded from the workbench startup script. If the TCS file uses tables defined within the scripts, make sure that those tables are loaded and executed before NetSim is asked to load a TCS file.

A typical loading of a TCS via the script file

```
// The DLL object is already instantiated, now load a custom tcs file
netsim_load_tcs("../tools\netsim\I5Lib\samples\example01.tcs");
// This puts NetSim in a ready state waiting for the proper WB events
netsim_start();
```


A.4 NetSim Workbench Script Reference

These are the functions that are called from a Workbench Script to initialize NetSim load NetSim TCS files.

A.4.1 `void netsim_verbose(int errorlevel)`

This sets the verbosity level of messages that NetSim generates in the Workbench Command window during the loading of TCS files. The values of errorlevel are:

- 0 == No output
- 1 == Detail initialize output
- 2 == Detail stream output
- 3 == Detail data table output
- 4 == Detail function output
- 5 == Detail shaping output (Future option)
- 6 == Detail police output (Future option)
- 7 == Detail shape/police clock (Future option)
- 8 and higher == All of the above outputs

A.4.2 `void netsim_load_tcs(char* tcs_filename)`

Initiates the loading of a TCS file. The parameter indicates the filename of the TCS file to load. This function may be called multiple times with different TCS filenames.

A.4.3 `void netsim_start(void)`

This function is called once after all TCS files have been loaded to start the simulation.

A.5 XML Basics

The Traffic Configuration Specification (TCS) files are the means with which a user can specify data to be input into an IXP simulation. The TCS file is implemented as an XML document. The NetSim DLL uses an XML parser to read and interpret the contents of a TCS file. In-depth knowledge of XML is not necessary to write a TCS file, but a basic understanding of the XML features that are used by NetSim makes it easier to comprehend.

The two types of XML files that are used by NetSim are Documents (*.TCS) and Document Type Definitions (*.DTD). A TCS file is an XML document. This TCS file specifies the protocols and data setup for a simulation run. In addition NetSim also uses XML Document Type Definition (DTD) files. The DTD file defines the structure of the XML document and can also be used to specify tables, defaults, and/or constant data types. NetSim can be used with other DTD files to allow the user to add new data table, function table, and custom frame types.

A.5.1 XML Documents

The first line of every TCS file should be the following statement:

```
<?xml version="1.0"?>
```

This is a directive telling the XML parser which version of XML to use.

Comments in XML start with `<!--` and end with `-->`:

```
<!-- This is a comment... -->
```

XML allows data to be easily organized in a tree structure comprised of node elements, node attributes, and empty nodes. Each node element may have attributes associated with it, and optionally contain data or other elements.

A.5.2 Node Element

A node element is delimited using the syntax `<...>` to indicate the start of a node and `</...>` to indicate the end of a node. Between these delimiters other nodes or raw data may be specified. For example, an element `foo` might contain the element `bar`, that in turn contains raw data, as shown in the example below.

Example of an Element

```
<?xml version="1.0"?>
<foo>
    <!-- this is an example XML Document -->
    <bar "data1" </bar><!-- bar is a sub element of foo -->
</foo>
```

A.5.3 Node Attributes

A node may also have attributes associated with it. When attributes are present, they are specified within the first delimiter for a node. For example each `bar` element above may have an attribute `id` associated with it as shown in the example below.

Foo Example1: Node Elements and Node Attributes

```
<?xml version="1.0"?>
<foo>
<bar id="1"> "data1" </bar> <!-- id is and attribute of "bar" -->
<bar id="2"> "data2" </bar>
</foo>
```

A.5.4 Empty Node

Sometimes an XML document may be designed such that a node does not contain any sub-nodes or data. Instead, the node consists only of attributes. Such a node is referred to as an empty node. Shorthand allows these to be defined by placing the forward slash “/” at the end of the starting delimiter like this:

Example: Shorthand Definition of an Empty Node

```
<nodeid attrib foo="1" />
```

Continuing with the foo example, suppose the designer of this document had designed the “bar” node to contain “data” as an attribute of the node. Now the document could be written as follows:

Foo Example2: an Empty Node

```
<?xml version="1.0"?>
<foo>
  <!-- both id and data are attributes of bar -->
    <bar id="1" data="data1" />
    <bar id="2" data="data2" />
</foo>
```

Whether a document uses attributes or sub-nodes are purely a matter of design and the needs of the application.

Notice that all data associated with an attribute in these examples is specified within quotes. All data assignments, either numeric or character, should be quoted in an XML document.

A.5.5 Document Type Definition (*.DTD)

An XML Document Type Definition (DTD) describes the layout of the document. For each element in an XML document, the DTD describes what attributes are associated with the element and what sub-elements may be contained within the element. The DTD enables the XML parser to detect typo's and syntax errors in the document. The DTD can also be used to define default values for node attributes.

A.5.5.1. Defining Elements in the DTD

<!DOCTYPE> Directive

The DTD is specified in a <!DOCTYPE> element that is placed between the opening <?XML> node and the start of the document itself. The format of the <!DOCTYPE> directive is shown below.

Format of <!DOCTYPE> Directive

```
<!DOCTYPE docname [DTD definition]>
```

<!ELEMENT> Directive

The <!ELEMENT> directive defines the name and contents of the node. The basic form of the directive is as shown below.

Format of <!Element> Directive

```
<!ELEMENT nodename nodedatatype>
```

The [Foo Example1: Node Elements and Node Attributes](#) and [Foo Example2: an Empty Node](#) may have a DTD specified as follows:

DTD Example 1: Example With One Element

```
<?xml version="1.0"?>
<!DOCTYPE foo [
    <!ELEMENT foo (bar)><!-- foo contains a single "bar" element -->
    <!ELEMENT bar (#PCDATA)><!-- bar contains character data -->
]>
<foo>
    <bar> "data1" </bar>
</foo>
```

This example has the “foo” element containing one, and only one, “bar” element—any other variation generates a parser error.

It may be desirable to allow some elements to contain 0, 1 or n sub-elements. Similarly, it may be desirable to allow the “foo” element to contain other sub-elements, or one of a selection of sub-elements. [Table A-1](#) shows examples of qualifiers and syntax that allow this type of control.

Table A-1. DTD Qualifiers and Syntax

Syntax Element	Usage	Meaning
?	<!ELEMENT foo (bar?)>	The element foo may contain zero or one instance of bar.
+	<!ELEMENT foo (bar+)>	The element foo contains one or more instances of bar.
*	<!ELEMENT foo (bar*)>	The element foo contains zero or more instances of bar.
	<!ELEMENT foo (foo bar)>	The element foo may contain either another instance of foo or an instance of bar.
and *	<!ELEMENT foo (foo bar)*>	These constructs may be mixed where appropriate. This example allows one or more instances of either the foo or the bar element to be sub-elements of foo.
#PCDATA	<!ELEMENT foo (#PCDATA)>	The element contains character data.
ANY	<!ELEMENT foo ANY>	The element foo may contain a single instance of any node declared within the DOCTYPE.
EMPTY	<!ELEMENT foo EMPTY>	The element foo may not contain sub-elements. This is typically used on element that only have attributes.

A.5.5.2. Defining Attributes in the DTD

Element attributes may be defined in the DTD using the `<!ATTLIST>` directive. The basic format of this directive is:

<!ATTLIST> Directive

```
<!ATTLIST node_name attribute_name CDATA value_description>
```

Table A-2 shows one of the value description field.

Table A-2. DTD: Value Description

Value Description	Usage	Meaning
Quoted (") data	<code><!ATTLIST bar id CDATA "0"></code>	Assigns the value in quotes as the default value for the attribute if it is not explicitly defined in the document.
#IMPLIED	<code><!ATTLIST bar id CDATA #IMPLIED></code>	Indicates that this is an "optional" attribute.
#REQUIRED	<code><!ATTLIST bar id CDATA #REQUIRED></code>	Indicates a required attribute. The XML parser reports an error if a document uses the element without specifying this attribute.
#FIXED	<code><!ATTLIST bar id CDATA #FIXED "value"></code>	Indicates that this attribute is not to be set directly in the document. It always evaluates to "value" when read by the parser.

In the example shown earlier—[DTD Example 1: Example With One Element](#), the user may now require to specify a value for the element `bar`'s `id` field as shown below.

DTD ATTLIST Example 1: Using Various Parameters

```
<?xml version="1.0"?>
<!DOCTYPE foo [
  <!ELEMENT foo (bar*)>
  <!ELEMENT bar EMPTY>
  <!ATTLIST bar id CDATA #REQUIRED> <!-- the id attribute is required -->
]>
<foo>
  <!-- not specifying an id value here would result in a parser error. -->
  <bar id="1"/>
</foo>
```

The `CDATA` parameter indicates that the parameter contains Character Data. As with element definitions, a range of values may be specified for a given attribute by listing the valid options in parentheses separated by "|". For example we could add a "type" option with legal values of `a`, `b`, or `c`, to the "bar" element as follows:

DTD ATTLIST Example 2: Using the Type Directive

```

<?xml version="1.0"?>
<!DOCTYPE foo [
    <!ELEMENT foo (bar*)>
    <!ELEMENT bar EMPTY>
<!--
The type here can have values of either a, b, or c.
If no type is specified the default is "a".
-->
    <!ATTLIST bar
id CDATA #REQUIRED
type (a | b | c) "a" >
]>
<foo>
    <bar id="1" type="c" />
    <bar id="2" /> <!-- no type specified default of "a" is used -->
</foo>

```

A.5.5.3. Importing DTDs from an External File

Typically it is desirable to share Element definitions in a DTD among several documents. This can be done in the !DOCTYPE declaration by using the SYSTEM keyword and specifying the name of the file containing the !ELEMENT and !ATTLIST declarations. For example, given a file named foo.dtd ([Example: foo.dtd](#)) may be re-written as shown in [Example: Importing a DTD file](#).

Example: foo.dtd

```

<!ELEMENT foo (bar*)>
<!ELEMENT bar (#PCDATA)>
<!ATTLIST bar id CDATA #REQUIRED type (a | b | c) "a" >

```

Example: Importing a DTD file

```

<?xml version="1.0"?>
<!DOCTYPE foo SYSTEM "..\lib\foo.dtd">
<foo>
    <bar id="1" type="c"/>
    <bar id="2" />
</foo>

```

A.5.5.4. Importing DTDs from Multiple Files

DTDs can be created by combining element definitions that are dispersed across multiple files. This is done using the `!ENTITY` declaration. An Entity in XML is a token that is defined in the DTD and can be included anywhere in the document using the name of the entity preceded by `'%'` and terminated by `"`;". Thus, if we have two files `foo.dtd` and `bar.dtd` as follows:

Example: `foo.dtd`

```
<!ELEMENT foo (bar*)>
```

Example: `bar.dtd`

```
<!ELEMENT bar EMPTY>
<!ATTLIST bar
  id CDATA #REQUIRED
  type (a | b | c) "a"
>
```

Example: `bar.dtd --re-written`

```
<?xml version="1.0"?>
<!DOCTYPE foo SYSTEM "..\lib\foo.dtd" [
  <!ENTITY % bar SYSTEM "bar.dtd">
  %bar;
]>
<foo>
  <bar id="1" type="c"/>
  <bar id="2"/>
</foo>
```

A.5.5.5. Defining Constant Values Using ENTITY Definitions

The `!ENTITY` declaration is also useful for declaring “constant” declarations that may be used within the XML file. For example, if in the previous declaration we wanted to define a constant called `MY_TYPE` that has the value “c” we could declare and use the `ENTITY` as follows:

Example: Defining Constant Values Using ENTITY Definitions

```
<?xml version="1.0"?>
<!DOCTYPE foo SYSTEM "..\lib\foo.dtd" [
  <!ENTITY % bar SYSTEM "bar.dtd">
  %bar;
  <!ENTITY MY_TYPE "c">
]>
<foo>
  <bar id="1" type="%MY_TYPE;" />
  <bar id="2"/>
</foo>
```

Every time `%MY_TYPE` appears in the XML file it expands to the value assigned in the `ENTITY` - “c” in above example.

A.6 NetSim TCS and DTD Configuration

A.6.1 TCS File Structure for NetSim

NetSim uses a DTD file (Netsim.dtd) to define the basic structure of the TCS file. This file typically resides in the same directory as the DLL and should not be modified. Thus, every TCS file starts with the following two elements (although the path to the DTD may need to be modified to point to the correct directory):

Elements at the beginning of a TCS File

```
<?xml version="1.0"?>
<!DOCTYPE netsim SYSTEM "..\lib\netsim.dtd">
```

A.6.1.1. Sections of TCS File

There are six major sections to a TCS file: `type_definitions`, `table_entries`, `streams`, `policies`, `rx_ports`, and `tx_ports`. Thus, the basic structure of a TCS file is as follows:

Basic Structure of a TCS File

```
<?xml version="1.0"?>
<!DOCTYPE netsim SYSTEM "..\lib\netsim.dtd">

<netsim>
  < type_definitions>
    ...
  </type_definitions>

  <streams>
    ...
  </streams>

  <policies>
    ...
  </policies>

  < rx_ports >
    ...
  </rx_ports >

  < tx_ports >
    ...
  </tx_ports >
</netsim>
```

The very simplest TCS files may not use the `type_definitions` or `table_entries` nodes. These entries may be omitted if not used.

A.6.2 Creating and Assigning Streams to a Device/Port

NetSim allows the user to define and assign <streams> to a Device/Port. A Stream is a collection of cells/frames sharing the same protocol stack and address information. Streams are created and assigned to the IXP receive ports via a `stream_set`. A `stream_set` is a collection of one or more streams that may be referenced as a single entity.

A `stream_set` is defined within the <streams> node. Each <stream_set> node has a <protocols> section that defines the protocol stack (and possibly addresses) to be used. A `stream_set` is assigned to a receive port via the <assign_stream_set> statement in the <rx_ports> section of the TCS file.

A.6.2.1. The Device and Port Attributes

Before you can connect a stream to a port, you must know the configuration of the workbench IXP device and ports. Using the device/port setup in [Section A.2.1, “Connecting NetSim to the IXP Workbench” on page 971](#), the two devices are configured.

- Device 0 is specified under the label “Receive side connection” next to the “Connect to device” assignment. The assignment is “0 (SPHY)” indicates “0” is the device.
- Device 1 is specified under the label “Transmit side connection” next to the “Connect to device” assignment. The assignment “1 (SPHY)” indicates “1” as the device.

The following code demonstrates how to create a `stream_set` with a single stream definition, and assign it to a device and port. The stream sends data into an Ethernet port.

Stream Example 1: Simple ethernet stream_set

```
<netsim>
<!-- ***** Stream Definitions ***** -->
<streams>
  <stream_set id = "my_ethernet_stream">
    <protocols>
      <ethernet
        src_addr      = "0xE5D000000000"
        dest_addr     = "0xEDA000000000"
        protocol_type = "0x05DD" />
      <ipv4_hdr
        src_addr      = "1.1.1.1"
        dest_addr     = "2.2.2.2" />
      <fixed_payload
        size          = "16"
        iterations    = "8" />
    </protocols>
  </stream_set>
</streams>
<!-- ***** RXPorts ***** -->
<rx_ports>
  <rx_port device = "0" port = "0">
    <assign_stream_set id="my_ethernet_stream" />
  </rx_port>
</rx_ports>
<!-- ***** TxPorts ***** -->
<tx_ports>
</tx_ports>
```

In [Stream Example 1: Simple ethernet stream_set](#), the `stream_set` has an `id` attribute with the value of “my_ethernet_stream”. This `id` is used in the `rx_port` section to assign the `stream_set` to an `rx_port` where the device and port are “0” in the IXP simulation. Also, note that the `<protocols>` node defines the protocol stack that is generated. The protocol headers are assigned to the generated frames in the same order, and with the specified addressing information.

The [Stream Example 1: Simple ethernet stream_set](#) may be interpreted as follows:

- From [Stream Example 1: Simple ethernet stream_set](#), only device 0 or port 0 are setup to receive streams from NetSim. Since there are no `tx_port` defined in the `<tx_ports>` section, no transmit ports are set to capture data.
- A single stream is created that contains a raw payload of 16 bytes of data (size = “16”). This payload is encapsulated with the ipv4 header, then again encapsulated with the ethernet header (ETHERNET | IPV4 | PL).
- When the ipv4 protocol encapsulates the raw payload, the header field (by definition is 20 bytes in length) is set initially to “0x450000000000000040600000000000000000”. Then the `src_addr`, and `dest_addr` translate the dot notation address into hex format and overwrite only those fields in the initial header definition.
- When the Ethernet protocol (the datalink level) encapsulates its payload, by default the header field for ethernet (by definition is 14 bytes in length) is initialized to “0x00000000000000000000000000000000”. Then the `src_addr`, `dest_addr`, and `protocol_type` fields are translated from a string hex notation in to only those specific fields in the initial header definition. At that point the Ethernet trailer is appended to its payload.
- This single stream is sent out to the `rx_port` when the workbench requests data from that device/port. When the stream is complete, another stream is constructed using the same process described above. A total of 8 streams are sent because the payload was defined to perform 8 iterations (iterations=”8”).
- A final observation is that by definition of the ipv4 header, a checksum and length field must be computed. These fields automatically computed and written when the payload is constructed. The Ethernet protocol also applies the same automation and generations the CRC field for the trailer.

[Stream Example 2: Simple Ipv6 Stream Set](#) illustrates some of the fields for the protocols defined above. Changing these fields can be designated as hex strings, integer string, char string, and so on.

Another code demonstration is sending an IPv6 stream to an Ethernet port.

Stream Example 2: Simple Ipv6 Stream Set can be interpreted as follows:

- From the example above, only one device/port is setup to receive streams. Since there are no tx_port is defined in the <tx_ports> section, no transmit ports are setup to capture data.
- A single stream is created containing a starting raw payload of ten bytes of data (start_size= "10"). This raw payload is encapsulated with the IPv6 header and the Ethernet header (Ethernet | IPv6 | PL).
- This single stream is sent out to the rx_port when the workbench requests data from that device/port. When the stream is complete, another stream is constructed using the same process described above. However in this example, the payload was designated to be constructed using the increment_payload protocol. Therefore, the start_size is incremented. A new stream is then constructed with a payload of 11. This continues until the payload exceeds end_size= "19". A total of 20 streams are sent. A single increment_payload statement generates 10 streams, this is repeated twice due to the cycles= "2" notation in the payload.

As you can see, a simple stream definition can generate a wide variety of data very quickly.

A.7 Miscellaneous Features

A.7.1 Inducing Frame/Cell Errors

There is a feature within NetSim that can generate bit errors up to a byte in length. The following example is the directive that can be assigned to an rx_port to generate random error. The directive is as follows:

Directive to generate random error

```
<frame_bit_error
    error_rate      = "0.001"
    max_burst_length = "8" />
```

The above directive tells the NetSim to generate a set of bit errors for every 1000 bits. The actual length is from 1 to 8 bits that is selected randomly. If not specified, by default the error_rate = 0 and the max_burst_length value is ignored.


```
00400030dddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd
dddddddddddddddddddddddddd
00400032dddddddddddddddddddddddddddddddddddddddddddddddddddddeeeeeeeeeeee
eeeeeeeeeeee0000007b193307a1
00400030aaaa030000000800450000b0000000004068a300b0b0b020b0b0b0108090202dddd
dddddddddddddddddddddd
00400030dddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd
dddddddddddddddddddddd
00400030aaaa030000000800450000b0000000004068a300b0b0b020b0b0b0108090104dddd
dddddddddddddddddd
00400030dddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd
dddddddddddddddddd
00400030dddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd
dddddddddddddd
00400032dddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd
dddddddddd000000b8d839ea30
00400030dddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd
dddddddddddddd
00400032dddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd
dddddddddd000000b803734eed
00400030aaaa030000000800450000b0000000004068a300b0b0b020b0b0b0108090203dddd
dddddddddddddd
00400030dddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd
dddddddddddddd
00400030dddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd
dddddddddd
00400032dddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd
dddddddddd000000b89cde9f40
00400030aaaa030000000800450000b0000000004068a300b0b0b020b0b0b0108090301dddd
dddddddddddddd
00400030dddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd
dddddddddd
00400030aaaa030000000800450000b0000000004068a300b0b0b020b0b0b0108090204dddd
dddddddddddddd
00400030dddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd
dddddddddd
00400030dddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd
dddddddddd
00400032dddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd
dddddddddd000000b8d5e7030b
00400030dddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd
dddddddddd
00400032dddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd
dddddddddd000000b846abc8a7
00400030aaaa03000000080045000073000000004068a6d0b0b0b020b0b0b0108090302dddd
dddddddddddddd
00400030dddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd
dddddddddd
00400032dddddddddddddddddddddddddddddddddddddddddddddddddddddeeeeeeeeeeeee
eeeeeeeeeeee0000007bddcc445c
00400030aaaa03000000080045000050000000004068a900b0b0b020b0b0b0108090303dddd
dddddddddddddd
00400032dddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd
dddddddddd00000058eaf61a4d
00400032aaaa03000000080045000020000000004068ac00b0b0b020b0b0b0108090304dddd
dddddddddd00000028f1ff7173
```

When the `eventlog_dir` directive is specified for the `rx_ports`, one log is generated into a file named `RXEventsN.txt`. This directive logs the events of all of the cells as passed to the receive port.

Note: The log—[Example of the RXEvent log: Events for TX port](#) was generated by the TCS setup for the Stream Distribution Example 4.

Example of the RXEvent log: Events for TX port

```

Device 1 Port 9 [w] Port streaming started
Device 1 Port 9 Stream 1 Pdu 1 [s] Cell 1
Device 1 Port 9 Stream 1 Pdu 1 [s] Pdu contained 1 cell(s)
Device 1 Port 9 Stream 1 Pdu 2 [s] Cell 1
Device 1 Port 9 Stream 2 Pdu 1 [s] Cell 1
Device 1 Port 9 Stream 2 Pdu 1 [s] Cell 2
Device 1 Port 9 Stream 1 Pdu 2 [s] Cell 2
Device 1 Port 9 Stream 1 Pdu 2 [s] Pdu contained 4 cell(s)
Device 1 Port 9 Stream 1 Pdu 3 [s] Cell 1
Device 1 Port 9 Stream 2 Pdu 1 [s] Cell 3
Device 1 Port 9 Stream 2 Pdu 1 [s] Cell 4
Device 1 Port 9 Stream 2 Pdu 1 [s] Pdu contained 3 cell(s)
Device 1 Port 9 Stream 1 Pdu 3 [s] Cell 2
Device 1 Port 9 Stream 1 Pdu 3 [s] Cell 3
Device 1 Port 9 Stream 1 Pdu 3 [s] Pdu contained 2 cell(s)
Device 1 Port 9 Stream 2 Pdu 2 [s] Cell 1
Device 1 Port 9 Stream 2 Pdu 2 [s] Cell 2
Device 1 Port 9 Stream 1 Pdu 4 [s] Cell 1
Device 1 Port 9 Stream 1 Pdu 4 [s] Cell 2
Device 1 Port 9 Stream 2 Pdu 2 [s] Cell 3
Device 1 Port 9 Stream 2 Pdu 2 [s] Cell 4
Device 1 Port 9 Stream 2 Pdu 2 [s] Pdu contained 6 cell(s)
Device 1 Port 9 Stream 1 Pdu 4 [s] Cell 3
Device 1 Port 9 Stream 1 Pdu 4 [s] Cell 4
Device 1 Port 9 Stream 1 Pdu 4 [s] Pdu contained 2 cell(s)
Device 1 Port 9 Stream 2 Pdu 3 [s] Cell 1
Device 1 Port 9 Stream 2 Pdu 3 [s] Cell 2
Device 1 Port 9 Stream 2 Pdu 3 [s] Cell 3
Device 1 Port 9 Stream 2 Pdu 3 [s] Cell 4
Device 1 Port 9 Stream 2 Pdu 3 [s] Pdu contained 4 cell(s)
Device 1 Port 9 Stream 3 Pdu 1 [s] Cell 1
Device 1 Port 9 Stream 3 Pdu 1 [s] Cell 2
Device 1 Port 9 Stream 2 Pdu 4 [s] Cell 1
Device 1 Port 9 Stream 2 Pdu 4 [s] Cell 2
Device 1 Port 9 Stream 3 Pdu 1 [s] Cell 3
Device 1 Port 9 Stream 3 Pdu 1 [s] Cell 4
Device 1 Port 9 Stream 3 Pdu 1 [s] Pdu contained 6 cell(s)
Device 1 Port 9 Stream 2 Pdu 4 [s] Cell 3
Device 1 Port 9 Stream 2 Pdu 4 [s] Cell 4
Device 1 Port 9 Stream 2 Pdu 4 [s] Pdu contained 2 cell(s)
Device 1 Port 9 Stream 3 Pdu 2 [s] Cell 1
Device 1 Port 9 Stream 3 Pdu 2 [s] Cell 2
Device 1 Port 9 Stream 3 Pdu 2 [s] Cell 3
Device 1 Port 9 Stream 3 Pdu 2 [s] Pdu contained 3 cell(s)
Device 1 Port 9 Stream 3 Pdu 3 [s] Cell 1
Device 1 Port 9 Stream 3 Pdu 3 [s] Cell 2
Device 1 Port 9 Stream 3 Pdu 3 [s] Pdu contained 2 cell(s)
Device 1 Port 9 Stream 3 Pdu 4 [s] Cell 1
Device 1 Port 9 Stream 3 Pdu 4 [s] Pdu contained 1 cell(s)
Generation stopped:
totalPDUs      processed 12
totalFrames/Cells processed 36

```

As shown in [Example of the RXEvent log: Events for TX port](#), it is much easier to walk through the stream-shaping algorithm for the example. It gives an indication of the Pdu and Stream from which the cell originated.

These same set of files are also generated for data captured via the transmit ports. However, all the details of origination may not be available.

Adding these attributes slows down the processing of the workbench, however, they are handy when trying to setup the TCS files.

A.8 Traffic Capture with Validation

NetSim allows the user to define and assign a `<protocols>` stack to a tx port for data capture for validation. This assignment is very similar to the assignment of protocols to a rx port, however, not all protocol parameters are used, therefore, may be ignored.

A.8.1 TCS Configuration for Capture with Validation

The [Capture Example: Simple IPv6 Capture Stream](#) illustrates how to configure a TCS file for data capture for data verification purposes. The example is exactly the same from [Stream Example 2: Simple Ipv6 Stream Set](#), however the `tx_ports` node now contains definitions. Also, a stream is specified and assigned to the `tx_port` with a type specification. It is also required that the `NETSIM.DLL` is assigned to the IXP Simulation transmit port on the Workbench.

Capture Example: Simple IPv6 Capture Stream

```
<netsim>
<!-- ***** Stream Definitions ***** -->
<streams>
<!-- This stream will match the 16 bit prefix -->
<stream_set id = "ipv6_stream_16">
<protocols>
<ethernet
    src_addr      = "0x1111111111102"
    dest_addr     = "0x000101010203"
    protocol_type = "0x86DD"
/>
<ipv6
    class        = "0x33"
    flow_label= "0x44444"
    next_header= "0x77"
    hop_limit= "0x89"
    source_addr  = "0x20000000000000000000000000000016"
    dest_addr    = "0x3FFE1D34567890123456789012345678"
/>
<increment_payload
    start_size = "&PAYLOAD_MIN;"
    end_size   = "&PAYLOAD_MAX;"
    cycles     = "&NUM_CYCLES;"
/>
```



```

</protocols>
</stream_set>

<stream_set id= "capture_protocol_stack" type= "capture">
    <protocols>
        <ethernet validate_level= "2"/>
        <ipv6 validate_level= "2"/>
        <fixed_payload />
    </protocols>
</stream_set>
</streams>

<!-- ***** RXPorts ***** -->
<rx_ports>
    <rx_port device = "0" port = "0">
        <assign_stream_set id= "my_atm_stream" />
    </rx_port>
</rx_ports>

<!-- ***** TxPorts ***** -->
<tx_ports>
    <tx_port device = "0" port = "0">
        <assign_stream_set id= "capture_protocol_stack"/>
    </tx_port>
</tx_ports>

```

Capture Example: Simple IPv6 Capture Stream can be interpreted as follows:

- From the example above, only device/port are setup to capture streams for verification. It is assumed that the NetSim DLL has also been assigned to the same port in the workbench.
- When the TCS file is loaded, one device/port is expected to receive data that follows the protocol stack of the assigned stream. The protocol stack is Ethernet | IPv6 | PL for the above example.
- When the TX port receives a frame from the workbench, the frame is captured and validated based on the datalink protocol assigned to the `tx_port`. The datalink protocol is the first protocol specified on the protocol stack. In [Capture Example: Simple IPv6 Capture Stream](#), this is the “Ethernet” protocol. At this time, all of the header and trailer are removed thus leaving the PDU. The PDU is then validated further based on the entire protocol stack. All validation statistics are logged in a directory specified by the `event_dir` parameter. The filename convention is based on the device/port number assigned in the `rx_port` section. The filename for the log above is `c:\temp\TxEvents08.txt` and original frames are in the `TxFrames08.txt` file.

As the workbench runs through the TCS files, it counts the number of frames and PDUs that have come to the port. When the workbench is removed from debugging mode, the totals are logged into the output log.

It is an advantage to assign NetSim to every transmit port even if no data is expected to be transmitted. If data is received from a port that has not been assigned a protocol stack, an error is logged into its predefined file.

PDU validation is dependent on the inter knowledge of the protocol. The most common type validation is to verify that the data pointers and size passed are within reason. If a pointer is received, NetSim insures that the header and trailer (if any) of that protocol fit within the given size. If these checks pass, then the payload size is verified by first making sure it falls within the min/max payload size specified by the protocol. If this passes, we make sure that any internal size held within the header or trailer equals to the actual payload size. Not all protocols contain internal sizes, therefore, it is assumed that the size passed is correct.

If protocol format is violated, the verification outputs may make no sense. For example, in the project for Ethernet, the CRC may be done by the hardware. The implementation of this project may completely eliminates the CRC field that NetSim expects to be present. The Ethernet protocol removes its header and trailer and passes the result to the IP protocol. In this case, an error occurs at the IP level as part of its information was removed.

AAL-2 Receive Microblock

B

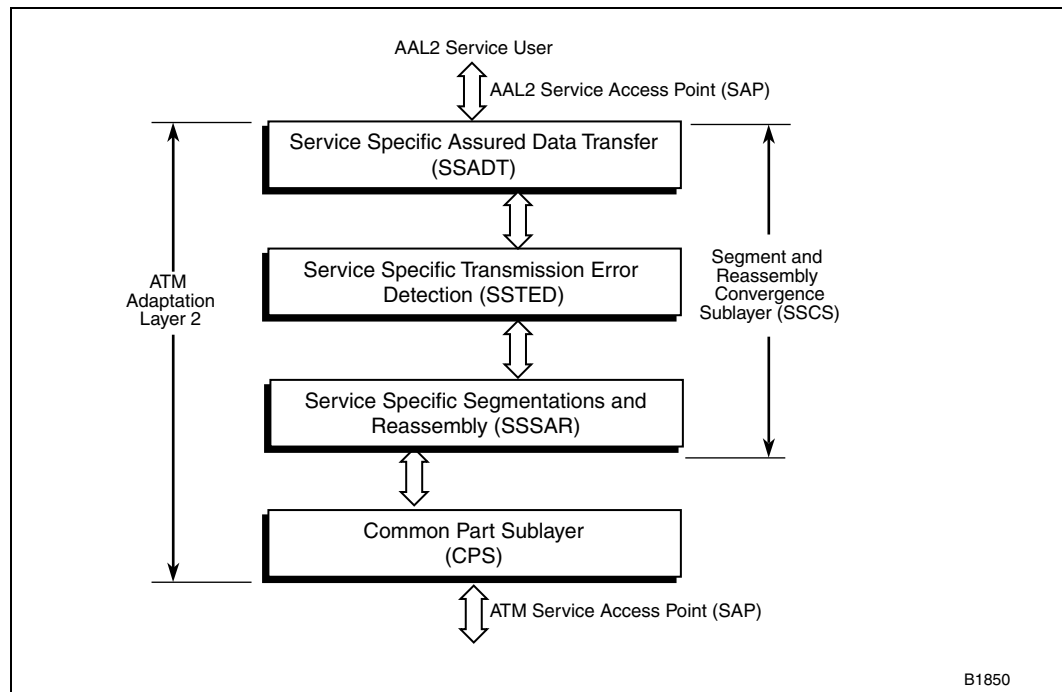
Note: The AAL-2 Receive microblock has a limited level of support in the Intel® Internet Exchange Architecture Software Development Kit (Intel® IXA SDK). Contact your Intel sales representative for more information.

Microblock	Description	Usage	Cycle Budget
ATM AAL-2 Receive	For ATM interface. Supports AAL-2 Receive (CPS and SSCS) processing. May be combined with AAL-5 microblock	Supports OC-12 data rates with 1 microengine on the IXP2400	420 cycles per microengine for OC-12 on the IXP2400

B.1 Overview

This section introduces the ATM Adaptation Layer (AAL) 2 terminology. Readers familiar with AAL-2 protocol layer [I.363.2] may skip this section. The AAL-2 provides efficient voice-over-ATM services. The AAL-2 structure is divided into two sublayers—the Common Part Sublayer (CPS) and the Service Specific Convergence Sublayer (SSCS). [Figure B-4](#) provides a graphical overview of AAL-2 and its sublayers.

Figure B-4. AAL-2 Sub-layers



B.1.1 AAL-2 CPS

The AAL-2 CPS provides the users a basic structure for identifying assembling/disassembling the variable payload associated with individual users, error correction, and the relationship with the SSCS. AAL-2 CPS has the following characteristics:

- Defined on an end-to-end basis as a concentration of AAL-2 channels
- Each AAL-2 channel is a bi-directional virtual channel, with the same channel identifier value used for both directions.
- Established AAL-2 channels over an ATM layer Permanent Virtual Circuit (PVC), Soft Permanent Virtual Circuit (SPVC) or Switched Virtual Circuit (SVC).

The multiplexing function in the CPS merges several streams of CPS packets (individual voice circuits) onto a single ATM connection.

B.1.2 AAL-2 SSCS

The AAL-2 SSCS is defined in [I.363.2] as the link between the AAL-2 CPS and the higher layer applications of the individual AAL-2 users. Two SSCS recommendations have been developed in conjunction with SSCS of which this design is only concerned with [I.366.1]. According to [I.366.1], the Service Specific Segmentation and Reassembly (SSSAR) sublayer for the AAL type 2 provides an SSCS for AAL-2 to handle segmentation and reassembly for data users. Additionally, the Service Specific Transmission Error Detection (SSTED) and the Service Specific Assured Data Transfer (SSADT) sublayers provide an error detection and delivery assurance for AAL-2. This design only takes into consideration the SSSAR sublayer for SSCS.

B.1.3 Design Overview

The AAL-2 RX microblock utilizes the “pool of threads” method to maintain the cell ordering. The AAL-2 Receive microblock group provides the AAL-2 sublayer processing for incoming ATM cells, which carry AAL-2 traffic.

B.1.3.1. Packet Sequencing through Thread Synchronization

The AAL-2 RX microblock uses the “pool of threads” method, and manages the thread synchronization using local memory and the CAM on each VC. When it receives a cell for AAL-2 processing, it performs the CAM lookup to find out whether any other thread has locked this VC flow. A match indicates that another thread has the VC locked and is processing data. In this scenario, where a VC is locked, a new entry from the Local Memory Queue freelist is retrieved and the `rbuf_offset`, which holds the cell data, is added to the queue of the thread belonging to the locked VC. If the VC is not already locked (that is, no other threads are currently processing data for the VC), it locks the VC by adding the VC to the CAM at the index equal to the current thread, retrieves the 4-byte VC and the 16-byte CID reassembly contexts from the SRAM.

B.1.3.2. Buffer Freelist

The CPS Receive component allocates the appropriate buffer after extracting a full CPS packet. Additionally, a new buffer may be required if there is not enough space in the existing buffer for the SSSAR block to write the packet, as the of SSSAR SDUs has a only a maximum capacity of 64k-bytes. The SSSAR block allocates a buffer from the freelist and writes the SRAM buffer

descriptor address in the first long word of the previous buffer descriptor, which corresponds to the SW next pointer. This process allocates any number of buffers required for holding the SSSAR SDU.

B.1.3.3. CPS and SSSAR Processing

The AAL-2 RX extracts CPS packets from ATM cells. This includes reassembling those CPS packets, which are split across two ATM cells. When a CPS packet has been completely extracted, the subtype variable is checked to determine if the current layer is the final sublayer of processing. If so, a buffer descriptor is allocated, which contains the CPS packet information (CID, UI, and length) and a handle to the packet payload is passed to the next block. Otherwise, if this is not the final sublayer, a check is made to see if the subtype is set to SSSAR. If this is the case, an empty buffer descriptor is passed to the SSSAR RX macro along with the CPS packet payload and the size of the CPS packet.

The AAL-2 RX outputs two 32-bit buffer handles. The buffer handles are used to access AAL-2 SDU packet descriptors stored in SRAM (Start of packet (SOP) and end of packet (EOP) packet descriptors).

B.2 Assumptions

The following assumptions are made:

- AAL-2 RX microblock only implements the AAL-2 CPS [I.363.2.] and AAL-2 SSSAR [I.366.1] functionality.
- AAL-2 RX Microblock uses pool of thread mechanism for thread ordering. This mechanism has its own drawback when there are less than 8 VCs in the system. At any time, only one thread would be processing packets for one VC. If the traffic is only for the VC being processed, then all other threads enqueue transmission requests for the VC into the local memory queue for the thread. This thread performs all CPS and SSSAR processing for the VC and the rest of the threads enqueue the transmission request to the thread processing the VC. This results in inefficient usage of the parallel processing capabilities of the ME.

B.3 Configuration Options

Table B-3 shows the configuration switches supported by the AAL-2 Receive Microblock.

Table B-3. Configuration Switches

Switch	Description
AAL2_RX_CID_SIMPLE_LOOKUP	Used for simple look-up to find the CID reassembly context. If this flag is defined, the 16-bit VC flow id and 8-bit CID is used to get the index to the CID reassembly context table. If this flag is not defined, the AAL-2 RX uses hash lookup to find the CID reassembly context.
AAL2_SINGLE_CPS_PER_CELL	Set to disable interleaving of CPS packets in ATM cells by the CPS sublayer functions.
AAL2_COUNTERS	Defined to enable the statistic counters

B.4 Data Structures

B.4.1 AAL-2 CPS Receive Context (CPS RXC) Table

The CPS receive maintains the CPS Receive Context—a per VC reassembly context. The reassembly context is 32 bits in size. The following table lists the fields in this data structure and their positions:

Table B-4. CPS RXC Data Structure

LW	Bits	Size	Fields	Description
0	7:0	8	CID	Last CID processed
0	8:8	1	Seq_num	Sequence num of the last cell of the VC
0	15:9	7	Expct	Bytes expected to complete last CPS packet
	17:16	2	Split	Used if partial header of last CPS packet is received
	31:18	14	cid_reassembly_ctx	Pointer to the last CID reassembly context processed

In order to support 64K VCCs, the AAL-2 CPS reassembly context table requires 256K-bytes.

B.4.2 CID Receive Context (CID RXC) Table

The CPS and SSSAR receive blocks maintain the CID Receive Context—a per CID reassembly context. The reassembly context is of 16-bytes and is described below.

Table B-5. CID RXC Data Structure

LW	Bits	Size	Fields	Description
0	31:0	32	Last LW0	Incomplete SSSAR packet. Used for-byte alignment
1	7:0	8	Old_len	Length of partially received packet
	31:8	24	Last LW1	Bits bits 8:31 of lword1 - bits 0:7 of lword1 has to be zero or will not be used. Used for-byte alignment.
2	23:0	24	Ph_buffer	Holds complete or partial 2-byte CPS packet header.
3	31:0	32	INFO_Buffer	Buffer of the last packet

If AAL2_RX_CID_SIMPLE_LOOKUP flag is defined, then the design uses lookup based on VC flow Id and CID to get the index of this table. If this flag is not defined, then a hash table is maintained in SRAM to get the index of the CID receive context table.

B.4.3 Hash Tables

The hash lookup is used to find the pointer to the CID reassembly context if simple lookup is not used. For this purpose, two tables—primary and secondary are maintained in SRAM. Each entry in the primary hash table (Table B-6) maintains two keys—combination of VC flow id and CID. It also contains two CID reassembly context pointers corresponding to the two keys stored in the

entry. To take care of the collision cases a pointer to the secondary hash table is stored in the entry. Each entry in the secondary table ([Table B-7](#)) can store up to four keys and a pointer to the next hash bucket in the link list.

Table B-6. Primary Hash Lookup Table

LW	Bits	Size	Fields	Description
0	31:0	32	Key1	First Key (VC Flow Id + CID).
1	31:16	16	CID_reass_ptr2	Index of the CID reassembly context in CID reassembly table for key 2, range[0, 64K -1].
	15:0	16	CID_reass_ptr1	Index of the CID reassembly context in CID reassembly table for key 1, range[0, 64K-1].
2	31:0	32	Key2	Second Key (VC Flow Id + CID).
3	31:16	16	Reserved	Reserved
	15:0	16	Sec_bkt_index	Index of the secondary bucket.

Table B-7. Secondary Hash Lookup Table

LW	Bits	Size	Fields	Description
0	31:0	32	Key1	First Key (VC Flow Id + CID).
1	31:16	16	CID_reass_ptr2	Index of the CID reassembly context in CID reassembly table for key 2, range[0, 64K -1].
	15:0	16	CID_reass_ptr1	Index of the CID reassembly context in CID reassembly table for key 1, range[0, 64K-1].
2	31:0	32	Key2	Second Key (VC Flow Id + CID).
3	31:0	32	Key3	Third Key (VC Flow Id + CID).
4	31:16	16	CID_reass_ptr4	Index of the CID reassembly context in CID reassembly table for key 4, range[0, 64K -1].
	15:0	16	CID_reass_ptr3	Index of the CID reassembly context in CID reassembly table for key 3, range[0, 64K-1].
5	31:0	32	Key4	Fourth Key (VC Flow Id + CID).
6	31:0	32	Reserved	Reserved.
7	31:16	16	Reserved	Reserved.
	15:0	16	Next_bkt_index	Index of the next secondary bucket in the chain.

B.4.4 CRC-5 Tables

The AAL-2 Receive Microblock uses two lookup tables to compute CRC-5 on the 19 bits of the CPS packet header. The lookup tables are stored in SRAM by the Xscale core and are read into the local memory at startup by the AAL-2 Receive Microblock as part of initialization. [Table B-8](#) shows the tables in SRAM.

Table B-8. AAL-2 Receive Microblocks CRC-5 SRAM Tables

able Name	Size	Description
CRC-5 8 bit lookup table	256-bytes	CRC-5 table used for CRC result using 8 bits of CRC data.
CRC-5 3 bit lookup table	32-bytes	CRC-5 table used for CRC result using 3 bits of CRC data.

B.4.5 Counters

[Table B-9](#) shows the counters available via the AAL2_COUNTERS build switch. Each VC queue is associated with a block of four 32-bit counters.

Table B-9. AAL-2 RX Counters

AAL2_NUM_RX_CELLS	Number of ATM cells received.
AAL2_NUM_RX_PKTS	Number of CPS packets received.
AAL2_NUM_RX_OFFSET_ERR	Number of AAL-2 CPS packets received with error in offset field.
AAL2_NUM_RX_SEQ_ERR	Number of AAL-2 CPS packets received with sequence number error.
AAL2_NUM_RX_OVRSIZE_ERR	Number of AAL-2 CPS packets received with length field error.
AAL2_NUM_RX_PARITY_ERR	Number of AAL-2 CPS packets received with parity error.
AAL2_NUM_RX_HEC_ERR	Number of AAL-2 CPS packets received with HEC error
AAL2_NUM_RX_DISCARD	Number of AAL-2 CPS/SSSAR packets discarded due to unavailability of buffers.

B.4.6 Local Memory Queue

Local memory queues are maintained to hold incoming RBUF elements for each thread. Each thread has its own queue of RBUF elements to process. [Figure B-6](#) illustrates the queue maintained in the local memory by each thread. For each thread, the head and tail of the queue is stored in local memory location. Each RBUF elements is stored in one longword and such a location is called as a local memory buffer. A queue for a thread is a link list of such buffer.

Initially, a freelist of local memory buffers is created. Each free buffer contains address of the next free buffer. The head of the local memory buffer freelist is stored in a GPR, @rbuf_element_freelist. Each time a RBUF element is required to queue a thread queue, the RBUF element is stored at @rbuf_element_freelist position and @rbuf_element_freelist is modified to reflect the new the next free position. Each thread has a corresponding local memory location which stores the head (bits 0:15) and the tail (bits 16:31) of the queue corresponding to the thread. Queue head and tail are the local memory address for the buffers where the start and end RBUF elements of the queue are stored. Each buffer stores RBUF element number (bits 16:31) and the next element (local memory address) (bits 0:15) in the queue. There are a total of 128 entries available in the local memory queue, which is enough to accommodate the full set of RBUF elements.

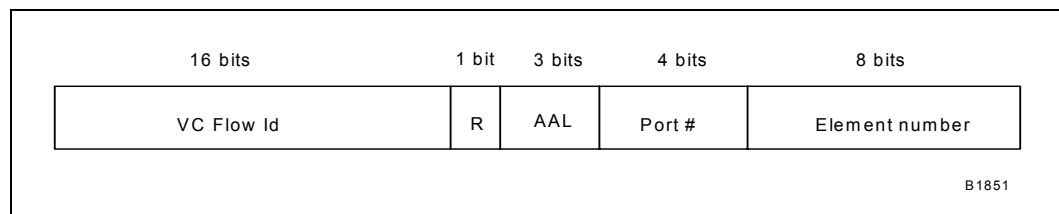
B.5 Algorithm

AAL-2 RX Microblock provides the AAL-2 sublayer processing. The various functions performed by this Microblock are as follow:

B.5.1 Cell RX

The AAL-2 RX Microblock receives data indication from ATM RX Microblock. The interface between ATM RX Microblock and AAL-2 RX Microblock is through the next neighbor registers. The data indication is 1 LW and [Figure B-5](#) shows its format.

Figure B-5. ATM-AAL-2 Data Indication

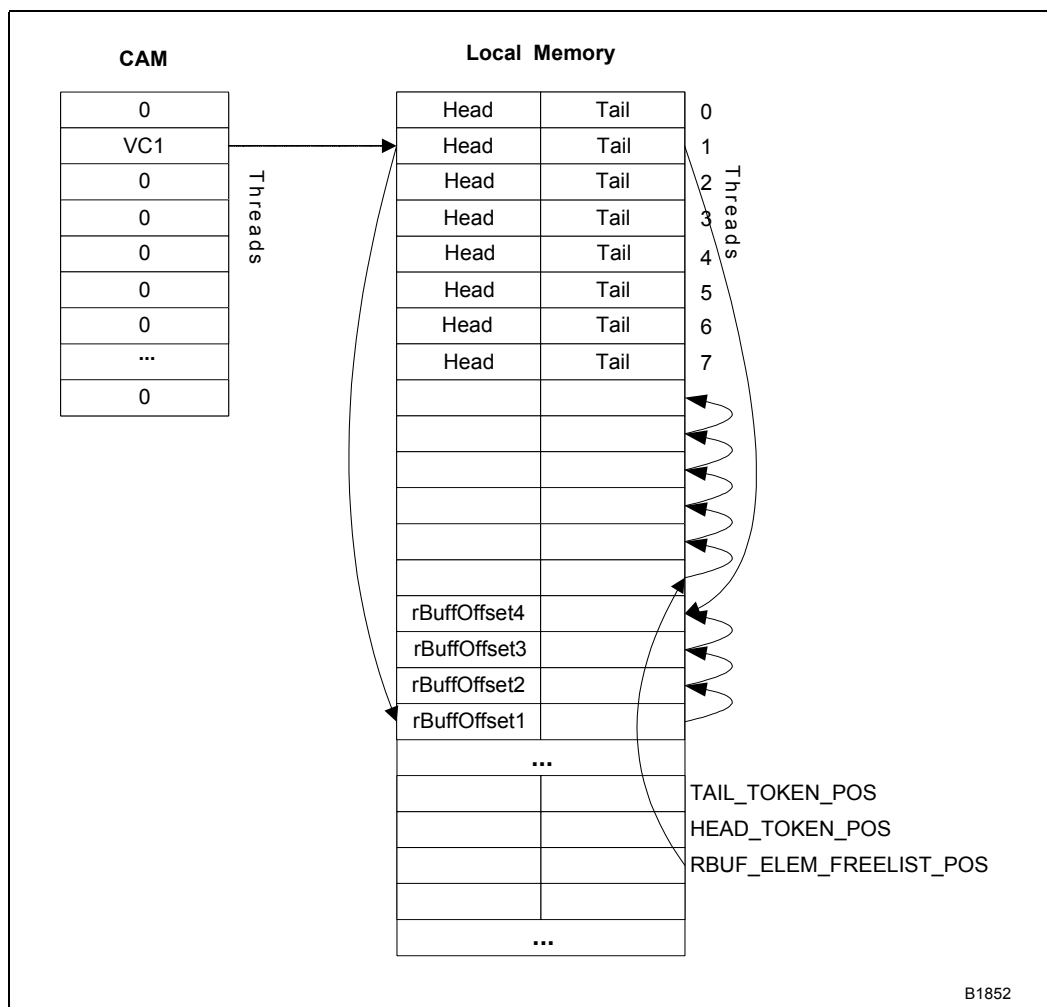


Element number indicates the RBUF element number. AAL type is ATM Adaptation Layer configured for this VC. The possible values are AAL-2 or AAL0. The cell is dropped if AAL type value is other than the AAL-2 or AAL0.

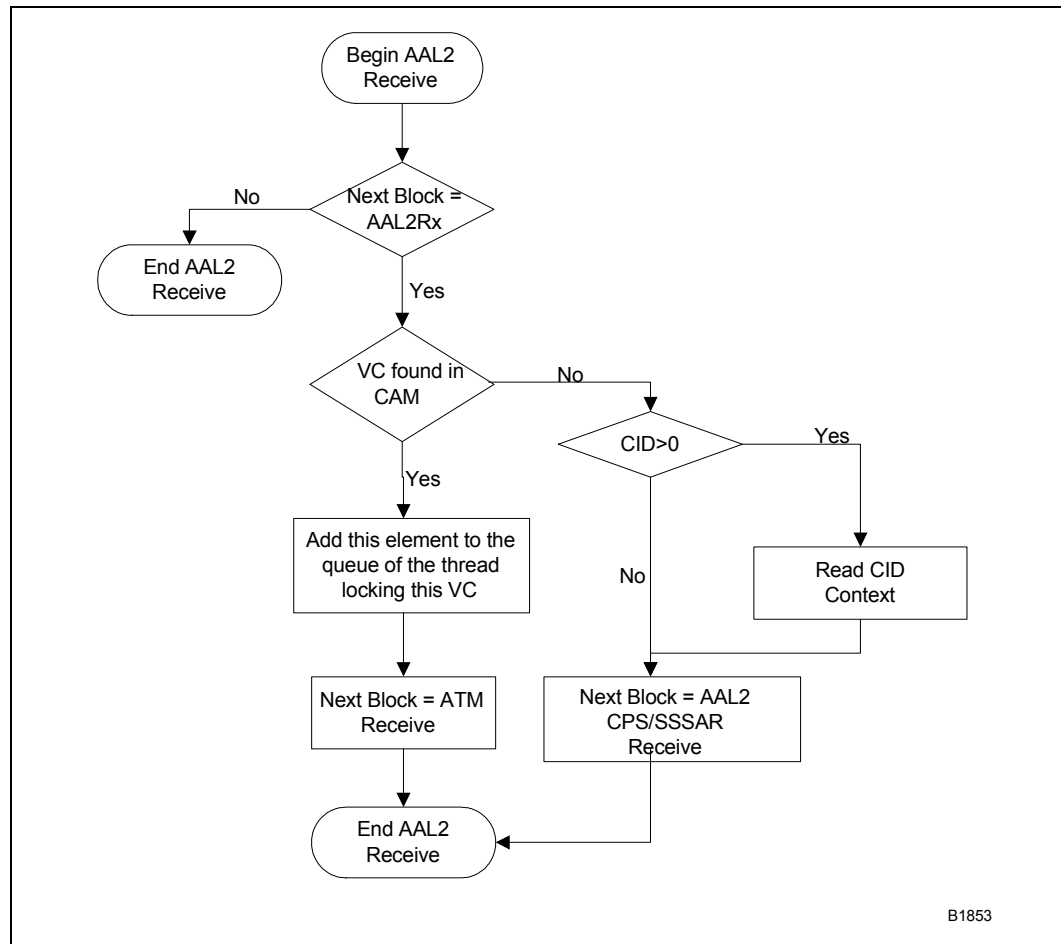
B.5.2 Thread Locking and Cell Queuing

The AAL-2 RX Microblock reads the cell from MSF and queues the cell into the thread queue maintained in local memory. As mentioned earlier, the AAL-2 RX Microblock manages the thread synchronization using local memory and the CAM for every VC flow id. First it searches the CAM for a match on the VC flow id. A match found indicates that another thread has the VC locked and is currently processing data for that VC. In this case, a new entry from the Local Memory Queue freelist is retrieved and the `rbuf_offset`, which holds the cell data, is added to the thread queue that has the VC locked. If the VC is not locked (that is, no other threads are currently processing data for the VC), the microblock locks the VC by adding the VC to the CAM at the index equal to the current thread. To help illustrate this thread synchronization mechanism, refer to [Figure B-6](#), which provides an example where thread 1 has VC1 locked and currently has 4 packets to process. The value at `RBUF_ELEM_FREELIST_POS` points to the next available element in the freelist. This is used and updated by threads when queuing and de-queuing elements from either its own or another thread's queue

Figure B-6. Local Memory Queues



Once it is determined that no other threads are currently processing a VC and it has been locked in the CAM by the current thread, then it retrieves the 4-byte VC and the 16-byte CID reassembly contexts from SRAM. [Figure B-7](#) provides an overview of the processing flows for this part of the AAL-2 RX microblock.

Figure B-7. Algorithm—AAL-2 Queue Cell Microblock


B.5.3 CPS/SSSAR Processing

The next step for AAL-2 RX Microblock is to extract CPS packets from within ATM cells. This includes reassembling those CPS packets, which are split across two ATM cells.

If the CPS packet has reached the CPS length, also the UUI field tells that there is “more” data in the SSSAR SDU to come (UUI == 27) then the SSSAR processing is completed. Otherwise, the CPS packet is written directly to DRAM. Figures B-8, B-9 and B-10 provide an overview of the processing flow through the AAL-2 CPS and SSSAR RX processing paths.

Figure B-8. AAL-2 CPS-SSAR Microblock Flowchart (page 1 of 3)

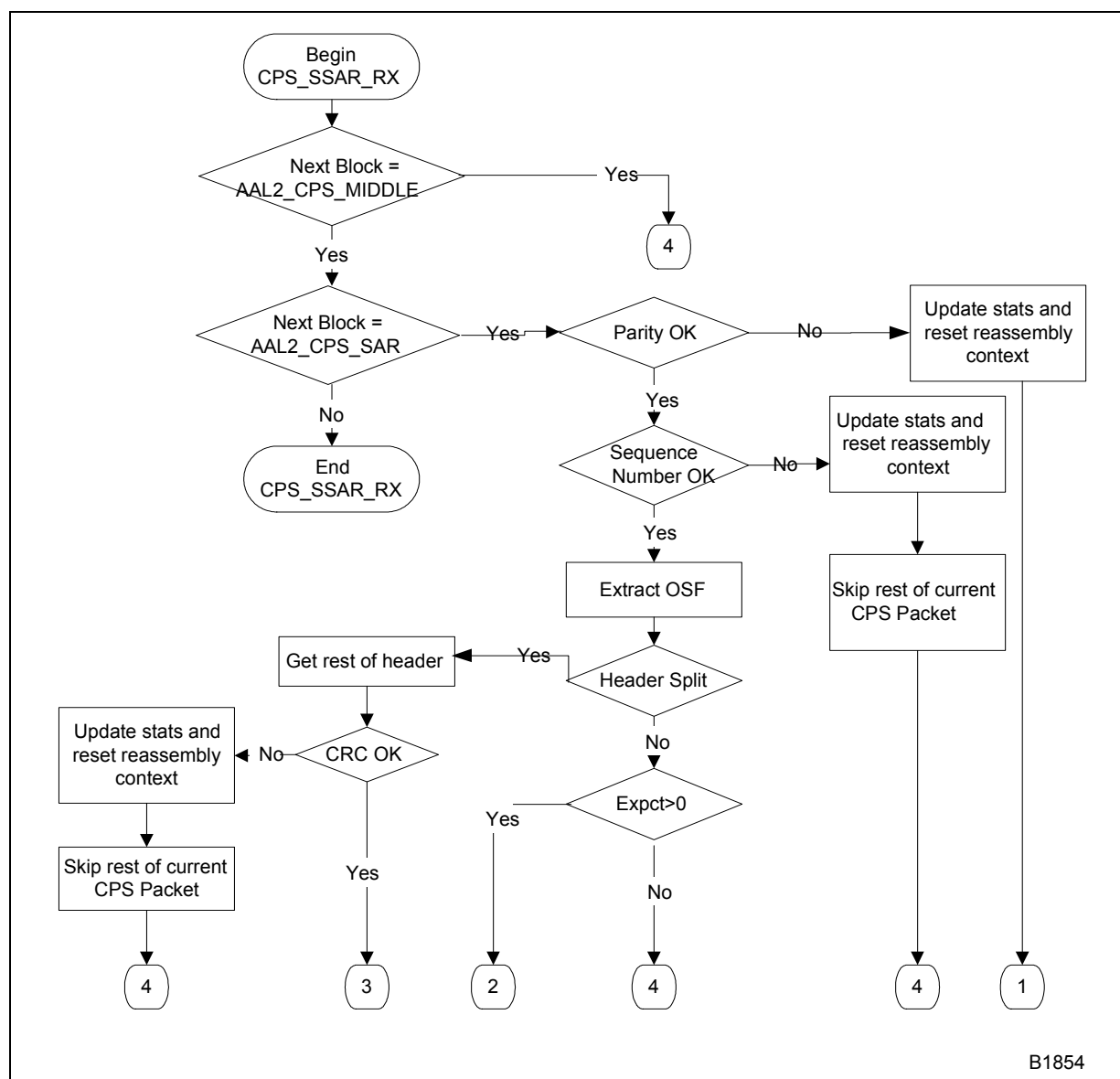
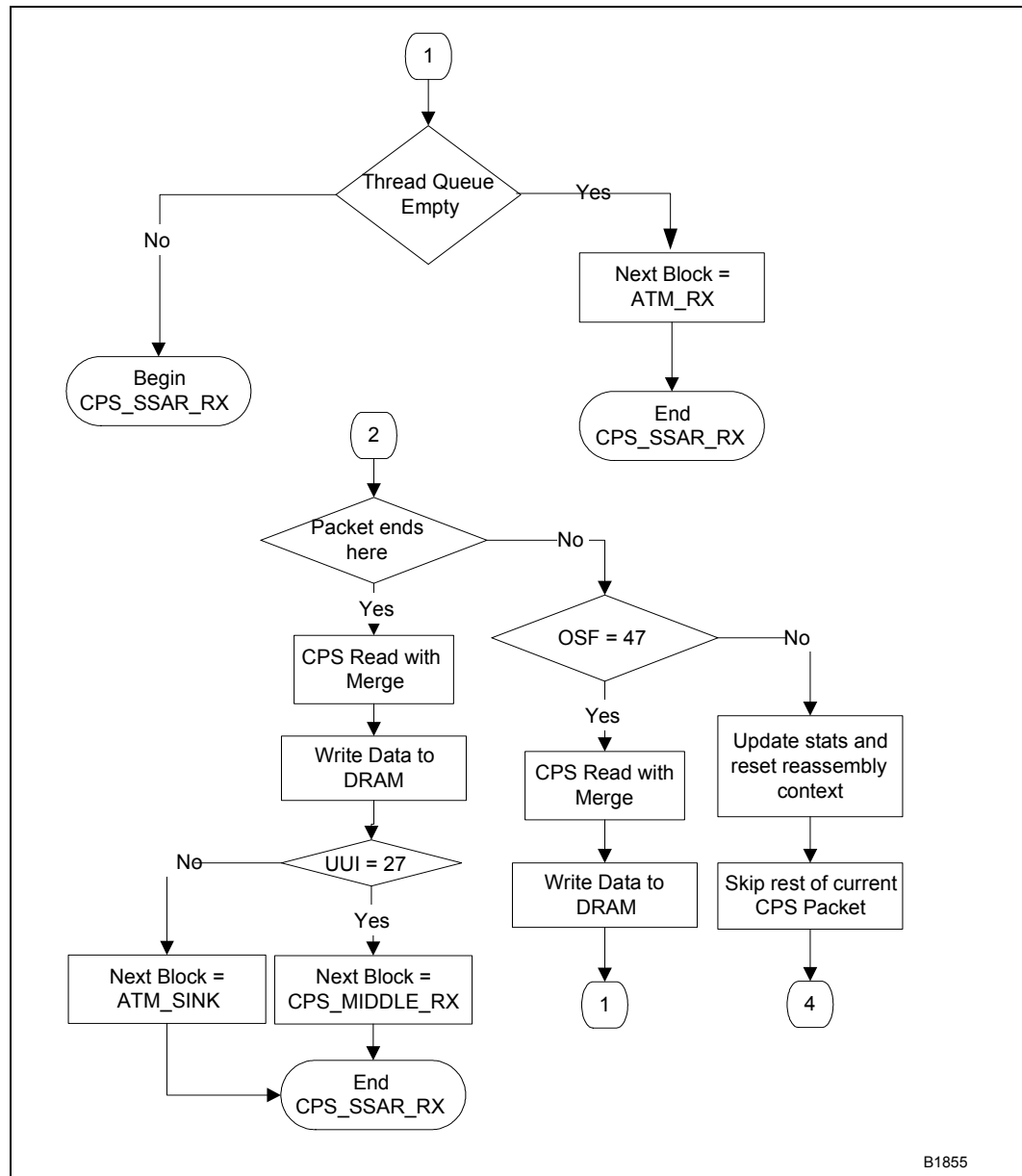
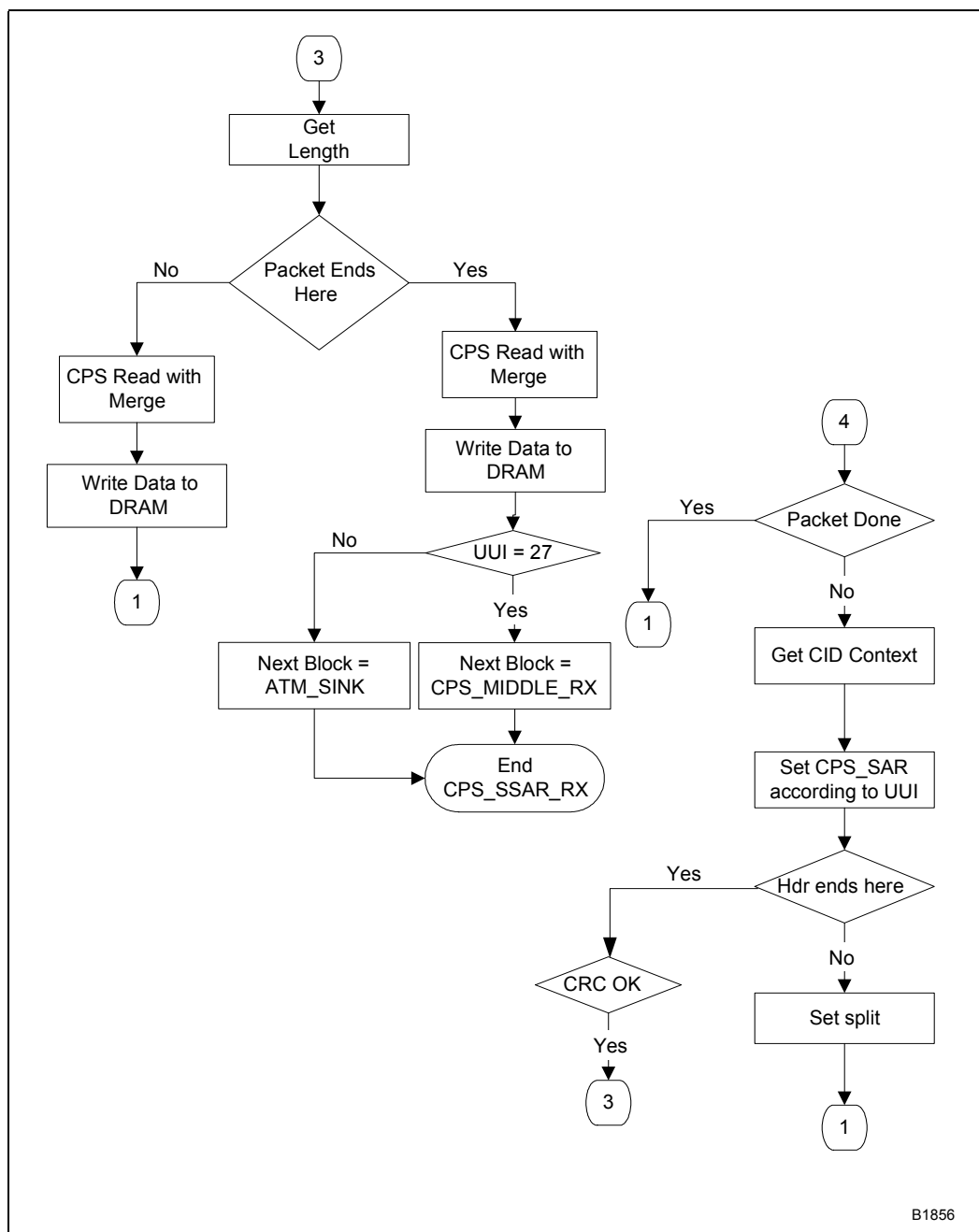


Figure B-9. AAL-2 CPS-SSAR Microblock Flowchart (page 2 of 3)



B1855

Figure B-10. AAL-2 CPS-SSAR Microblock Flowchart (page 3 of 3)



The AAL-2 RX Microblock outputs a 32-bit buffer handle, which can be used to access the buffer descriptor in SRAM, which holds information representing a single AAL-2 SDU.

B.5.4 Performance Analysis

Table B-10 shows the cycle counts and I/O latencies for processing 44 byte CPS packets.

Table B-10. Cycle Counts and I/O Latencies

Scenario	Cycle count /thread	I/O Latency
CPS packet processing (VC locked in CAM)	267	396
CPS packet processing (VC not locked)	270	712
SSSAR processing on first/middle CPS packet	264	598
SSSAR processing on last CPS packet	292	281

Note: The control space usage is 1909.

Note: The AAL-2 Receive microblock has a limited level of support in the Intel® Internet Exchange Architecture Software Development Kit (Intel® IXA SDK). Contact your Intel sales representative for more information.

Microblock	Description	Usage	Cycle Budget
AAL2_TX	For ATM interface. Supports AAL-2 cells. May be combined with AAL-5 microblock	Supports OC-12 data rates with 1 microengine on the IXP2400	420 cycles per microengine for OC-12 on the IXP2400

C.1 Overview

ATM Adaptation Layer (AAL) 2 provides bandwidth-efficient transmission of low-rate, short and variable length packets in delay sensitive applications. A single ATM connection supports more than one AAL-2 user information streams. The AAL-2 makes use of the service provided by the underlying ATM layer. Multiple AAL connections may be associated with a single ATM layer connection, allowing multiplexing at the AAL—multiplexing in the AAL-2 occurs in the Common Part Sublayer (CPS).

Figure C-11. CPS and SSCS Interaction

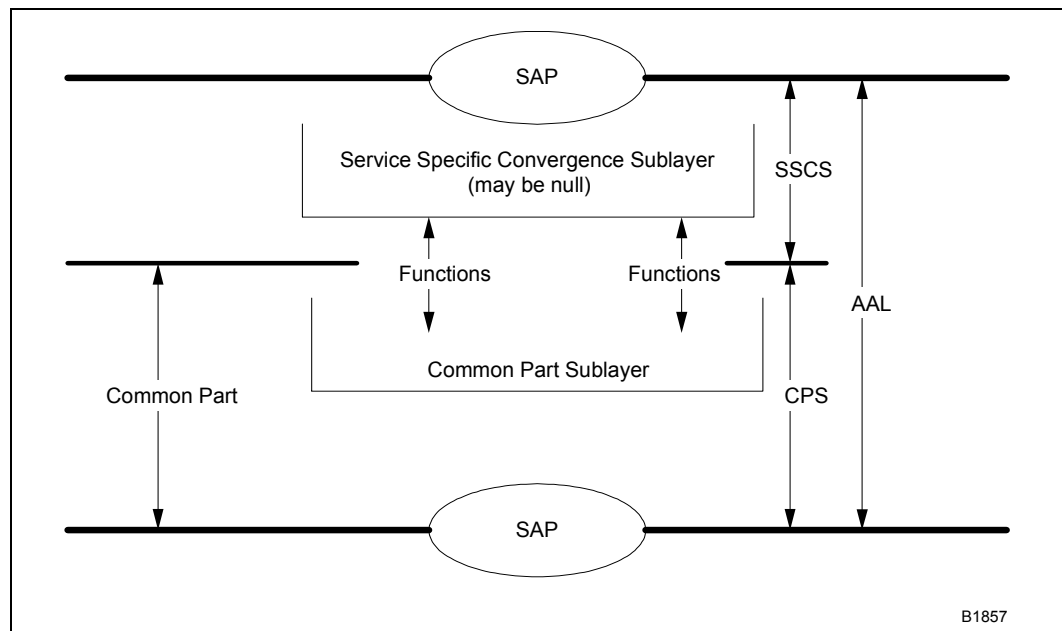


Figure C-11 shows the interaction between the CPS and Service Specific Convergence Sublayer (SSCS). The AAL-2 CPS provides the capabilities to transfer CPS-SDUs from one CPS user to one other CPS user through an ATM network. The service offers the following peer-to-peer operation:

- data transfer of CPS-SDUs of up to 45 (default) or 64 octets

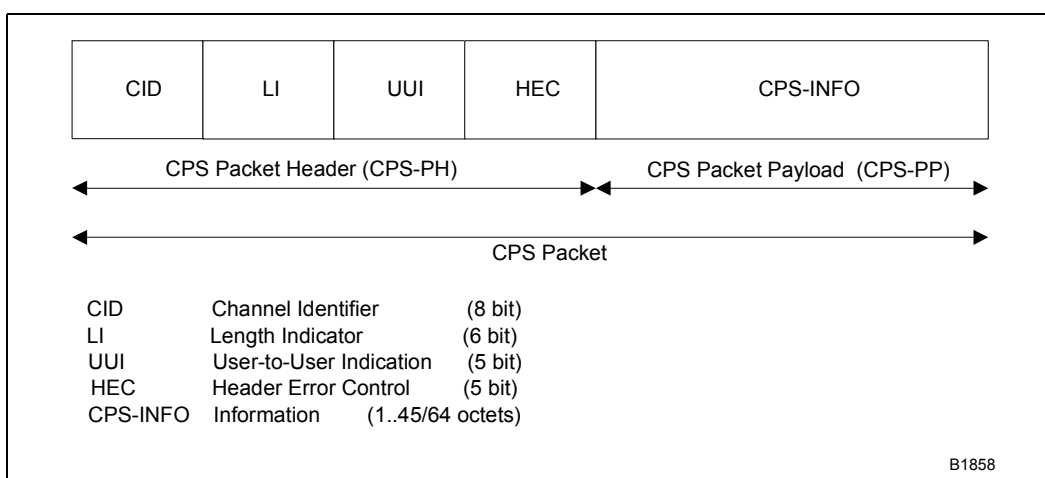
- multiplexing and demultiplexing of multiple AAL-2 channels
- CPS-SDU sequence integrity is maintained on each AAL-2 channel

The AAL-2 CPS has the following characteristics:

- The AAL-2 CPS connection is defined on an end-to-end basis as a concatenation of AAL-2 channels.
- The AAL-2 channel is a bidirectional virtual channel. The same value of channel identifier value shall be used for both directions.
- The AAL-2 channels are established over an ATM layer Permanent Virtual Circuit (PVC) or Switched Virtual Circuit (SVC).

Figure C-12 shows the format and coding of the CPS packet.

Figure C-12. Format of AAL-2 CPS-Packet



The Segmentation and Reassembly Service Specific Convergence sublayer applied for a Service Specific Convergence sublayer for the AAL-2 makes it possible to transport a packet size of more than the maximum length specified in the CPS and also to multiplex with low-rate and short length packets in delay sensitive application. The AAL-2 Service Specific Segmentation and Reassembly (SSSAR) provides the capabilities to transfer SSSAR SDUs from one SSSAR user to one other SSSAR user through the Common Part Sublayer (CPS).

The service offers peer-to-peer operation:

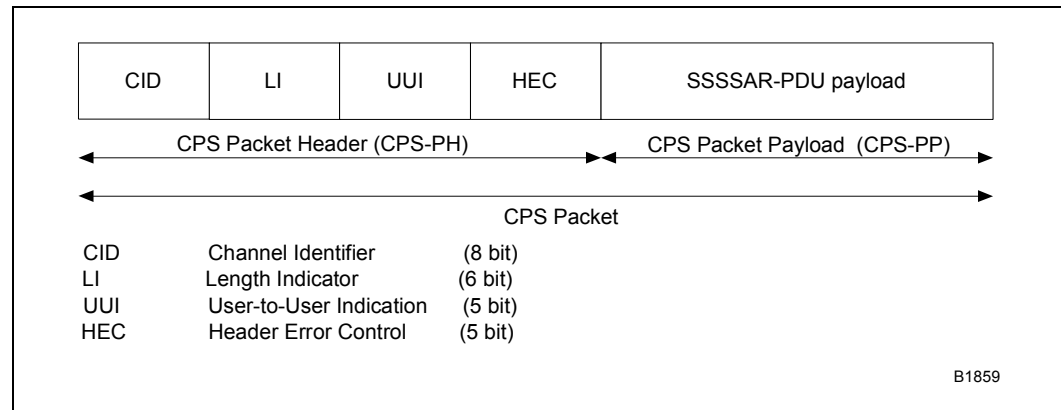
- Data transfer of SSSAR SDUs of up to 65 568 octets.
- SSSAR SDU sequence integrity is inherited from the Common Part Sublayer of the AAL-2.

The AAL-2 SSSAR connection utilizes the connections of the AAL-2 CPS; therefore, they inherit their characteristics; in particular, the AAL-2 SSSAR connection is a bi-directional virtual channel.

The AAL-2 TX microblock provides the CPS and SSSAR sublayers. The AAL-2 TX microblock receives the CPS-SDU or the SSSAR-SDU. The SSSAR-SDU is processed by the SSSAR sublayer functions of the AAL-2 TX microblock to create CPS SDU. The CPS sublayer functions of the AAL-2 TX microblock create the CPS packet header for the CPS-SDU and multiplex the CPS packet into ATM cells.

Table C-13 shows the format of the SSSAR PDU. The CPS User-to-User Indication (UII) is used to implement a “More Data” bit (M). A CPS UII value of “27” indicates that more data is required to complete the reassembly of an SSSAR SDU. Any other value, that is, between “0” and “26”, indicates the receipt of the final data of an SSSAR SDU.

Figure C-13. Format of SSSAR-PDU



C.1.1 Design Overview

The AAL-2 transmit microblock is implemented as a pool of seven SSSAR/CPS protocol processing threads performing the SSSAR and CPS sublayer functions on one microengine with one thread in the microengine dedicated to provide timer services to implement the `Timer_CU` functions.

C.1.1.1. Packet Sequencing through Thread Synchronization

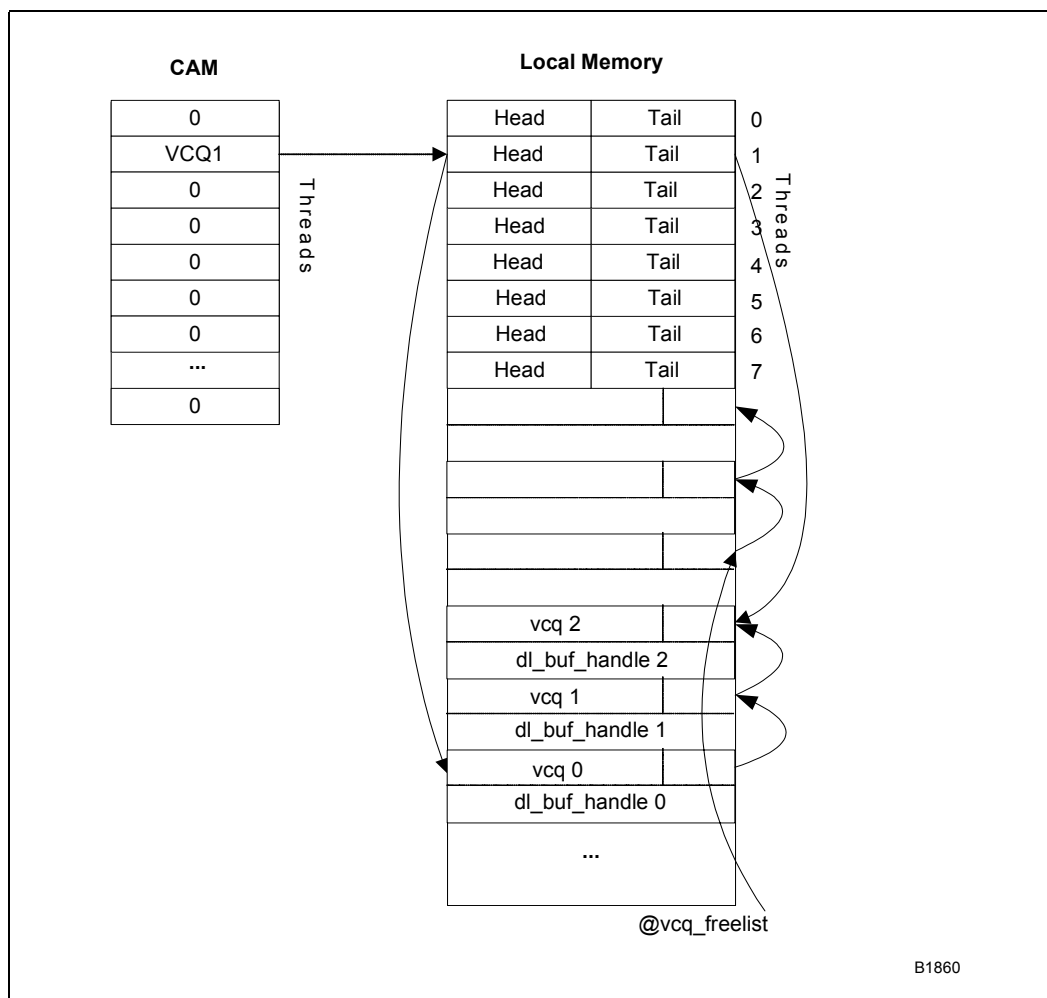
In order to ensure packet sequencing, a Virtual Circuit (VC) is serviced by only one of the seven processing threads at any given time, that is, no two threads work on a VC simultaneously. The synchronization between the threads is performed using CAM and local memory for each VC. Each processing thread maintains a queue in local memory to store the received data transmission requests to be processed by that thread.

The AAL-2 transmit microblock receives transmit requests on its input scratch ring. On receiving a transmit request for a packet, the thread handling the transmit request searches through the CAM to determine if any other thread is handling the same VC. If a different thread is already handling the VC, the received data transmission request is placed in the local memory queue of that thread and the threads goes back to process the next transmit request from the input scratch ring.

If no other thread is processing the VC for which a transmit request is received, the thread receiving the transmit request creates an entry in the CAM for that VC and locks the VC for processing.

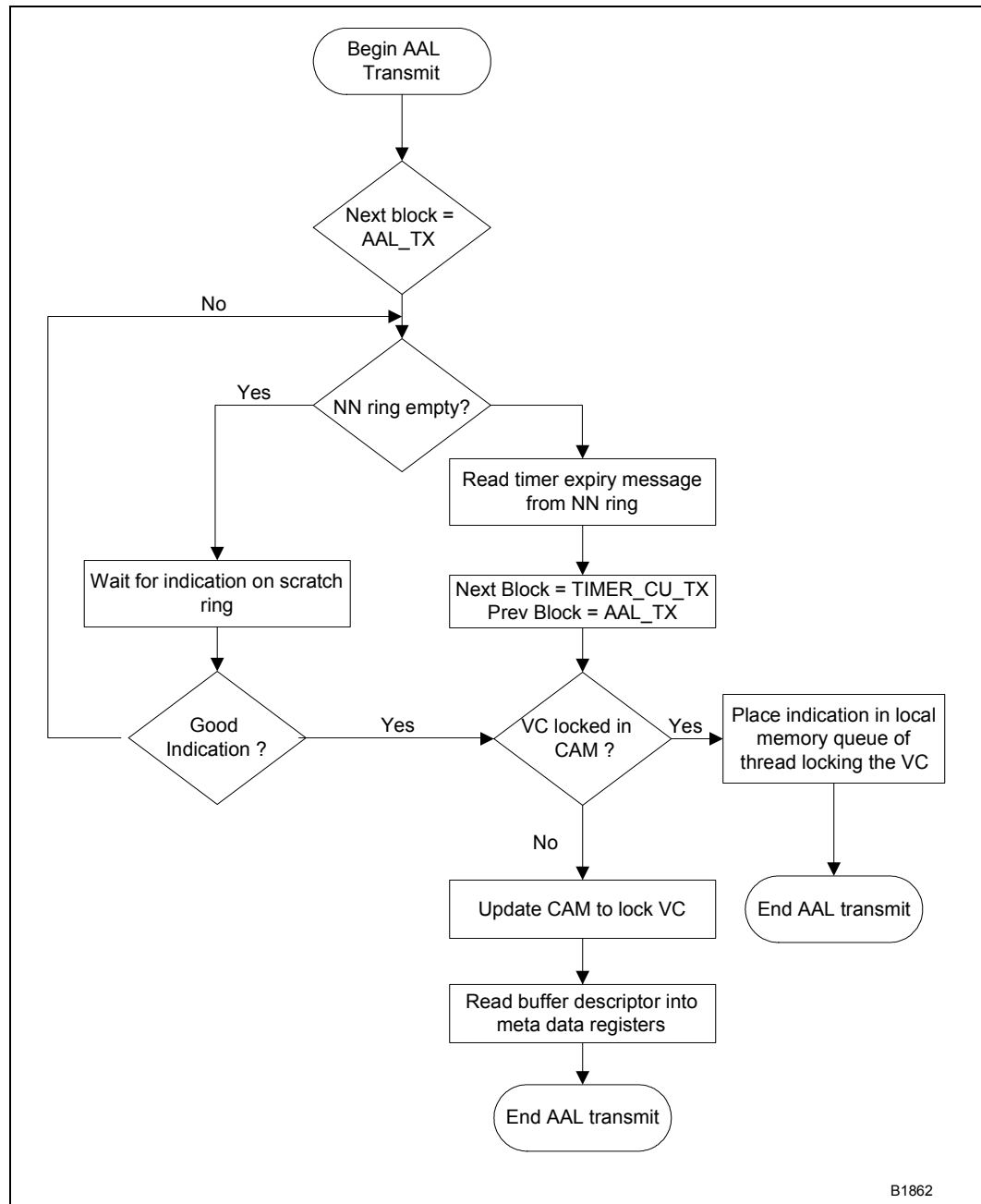
Table C-14 illustrates this thread synchronization mechanism with an example in which thread one has VC1 locked and it currently has three packets to process. The value at `vcq_freelist` points to the next available element in the freelist. This is used and updated by threads when queuing and dequeuing elements from either its own or another thread's queue.

Figure C-14. Pool of Threads Synchronization With CAM/LM



Once it is determined that no other threads are currently processing a VCQ and it has been locked in the CAM by the current thread, the transmit reassembly context for the VC and the Meta data for the buffer handle just received is retrieved from SRAM. [Table C-15](#) shows the flowchart for sourcing transmit request from input scratch ring and performing the thread synchronization.

Figure C-15. AAL-2 TX Transmit Request Sourcing and Thread Synchronization



B1862

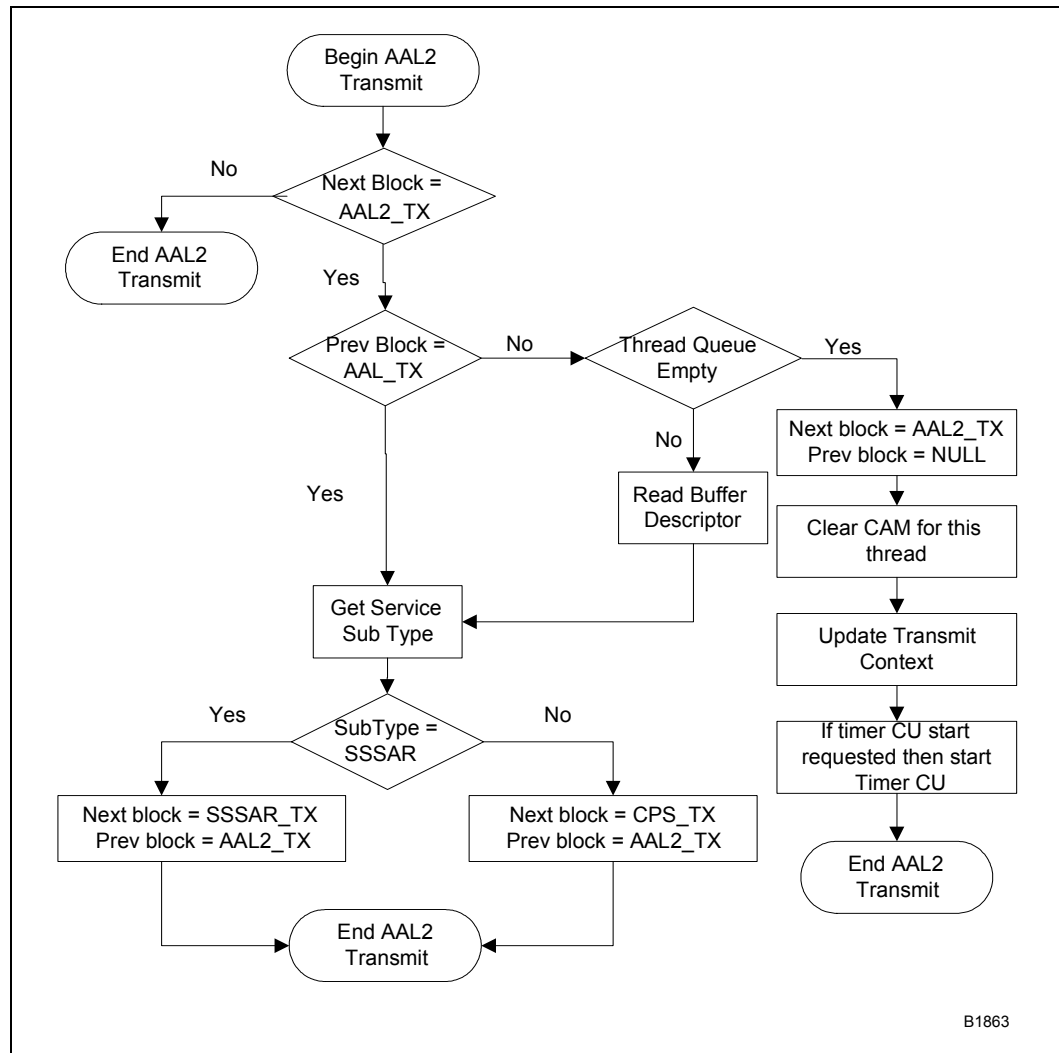
C.1.1.2. SSSAR/CPS SDU Processing

The data packet indication received from the scratch ring or from the local memory thread queue (placed by some other thread) is processed based on the AAL parameter provided in the buffer descriptor associated with it. The AAL parameter contains the following information:

- Service subtype—indicates whether the data is to be processed as a CPS SDU (single CPS packet) or SSSAR SDU (multiple CPS packet).
- Maximum PDU length—indicates maximum length of the PDU for both CPS and SSSAR. For CPS, there are only two possible values for maximum PDU length, 45 or 64. However, SSSAR PDU lengths can range from 0 to 63.
- Channel Identifier (CID)—the channel on which the CPS packet is to be transmitted.
- User to User Indication (UUI)—the UUI value to be filled in the CPS packet header. When SSSAR processing is to be done on the data packet the UUI is set to “27” for all CPS packets except the last CPS packet in which the UUI value indicated in the AAL parameter is updated.
- Timer CU value—the duration of the `timer_cu` to be started if the current CPS packet does not fill the ATM cell completely. The duration is specified in 0.1 ms units and can range from 0.1 ms to 10 ms. Timer CU value of 0 indicates that timer CU should not be started and the cell transmitted immediately.
- Loss priority—indicates the submitted loss priority to be used for generating the data request to the ATM layer.

Figure C-16 shows the flow chart for processing the data packet indication received over the input scratch ring or extracted from the threads local memory queue.

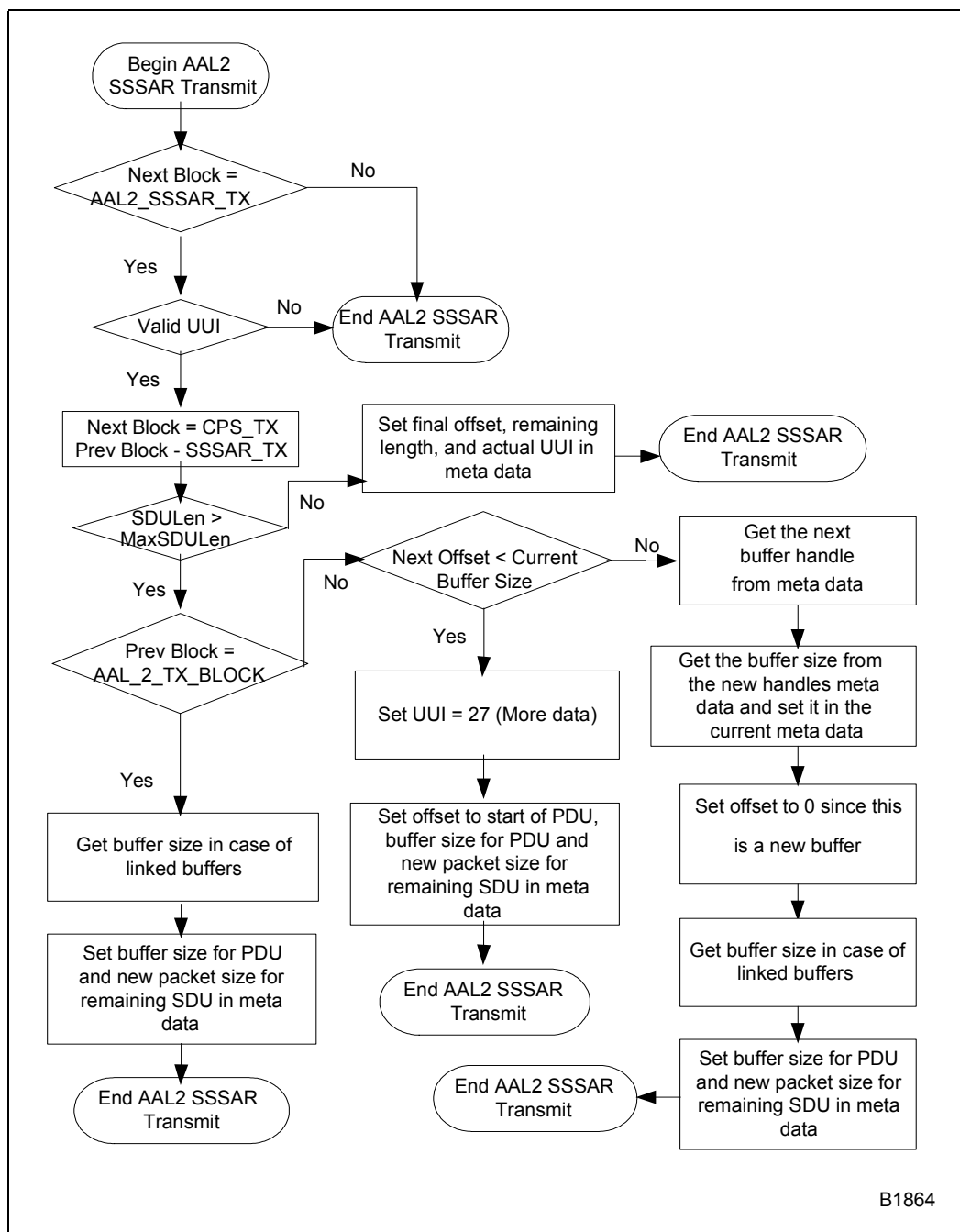
Figure C-16. AAL-2 Sublayer Selection and Sourcing from Local Memory Thread Queue



C.1.1.3. SSSAR Sublayer Functions

If the service subtype indicates that the received data packet is a SSSAR SDU, the SSSAR sublayer functions of the AAL-2 Transmit microblock are invoked to process the packet. As long as there is data in the SDU, the SSSAR sublayer function will segment off “Maximum PDU Length” packets and pass a pointer (existing packet descriptor with appropriate offset and size) to the packet segment to CPS sublayer functions. Additionally, the UUI for the CPS packet is determined based on which segment of the SDU is being processed. For the first and all intermediate packet segments, UUI is passed as 27 to the CPS sublayer functions. However, for the final packet segment, the original UUI from the packet descriptor is passed to the CPS sublayer functions. Figure C-17 shows the flowchart for the AAL-2 SSSAR sublayer function.

Figure C-17. AAL-2 Transmit SSSAR Sublayer Functions



C.1.1.4. CPS Sublayer Functions

The CPS sublayer functions provide multiplexing of CPS packets of various CIDs destined for a specific ATM channel (VPI/VCI/port) into ATM cells. When a new CPS packet is received from a packet descriptor, the AAL parameter passed in the meta data is read to determine the CID, UUI,

and length indicator required to build the CPS packet header. Additionally, a CRC5 algorithm is applied to the first 19-bits (CID, UI, LI) of the CPS packet header in order to compute the HEC which is then appended to the end of the packet header. The CPS transmit block continues to process by determining where the new data needs to be packed into the ATM cell. Several of the following cases are considered:

- The current ATM cell buffer is empty and this is the first CPS packet to add to the buffer. In this case, the ATM cell header is added to the cell buffer. Additionally, the AAL2 start field is constructed using the start field offset and sequence number, which are stored in the transmit context for each VC, as well as a parity bit. Finally, the CPS packet header and packet data are byte aligned into the cell buffer.
- The current ATM cell buffer already contains CPS packet data. In this scenario, the CPS packet header is byte aligned with the existing data in the ATM cell buffer. This is accomplished by merging the last 4-bytes written to the cell buffer, which is stored on a per VC basis in SRAM, with the CPS packet header. The remainder of the CPS packet header (if all 3 bytes were not aligned with the last long word written to the cell buffer) and the packet data are byte aligned and added to the cell buffer.
- The current ATM cell buffer already contains CPS packet data and the entire new CPS packet will not fit into the current cell buffer. In this case, the processing in step 2 above is followed for as much of the new packet data, which will fit into the current cell buffer. Once the cell buffer is filled and stored in DRAM, the cell descriptor is queued to the next stage for transmission. Next, a new cell buffer is allocated and the processing returns to step one for the remainder of the data in the CPS packet.

At the end of CPS packet processing, if the ATM cell buffer is not complete, the `Timer_CU` duration provided in the AAL parameter is checked to determine if the `Timer_CU` is enabled. If `Timer_CU` is not enabled, the ATM cell is transmitted immediately. However if a non zero `Timer_CU` duration is indicated in the AAL parameter, the CPS sublayer functions start the `Timer_CU` for that VC and store the buffer descriptor of the partially filled ATM cell in the transmit context for that VC. The following two cases are considered:

- A new CPS packet is received for transmission, which causes the partially filled ATM cell to be filled completely. In this case, the `Timer_CU` started for this VC is cancelled and the ATM cell is queued to the next stage for transmission.
- The `Timer_CU` expires and a timer expiry notification is received from the timer thread. On receiving this notification, the unfilled portions of the partially filled ATM cell are padded with zero and the ATM cell buffer is enqueued to the next stage for transmission.

Figure C-18. AAL-2 CPS Transmit Sublayer (1 of 3)

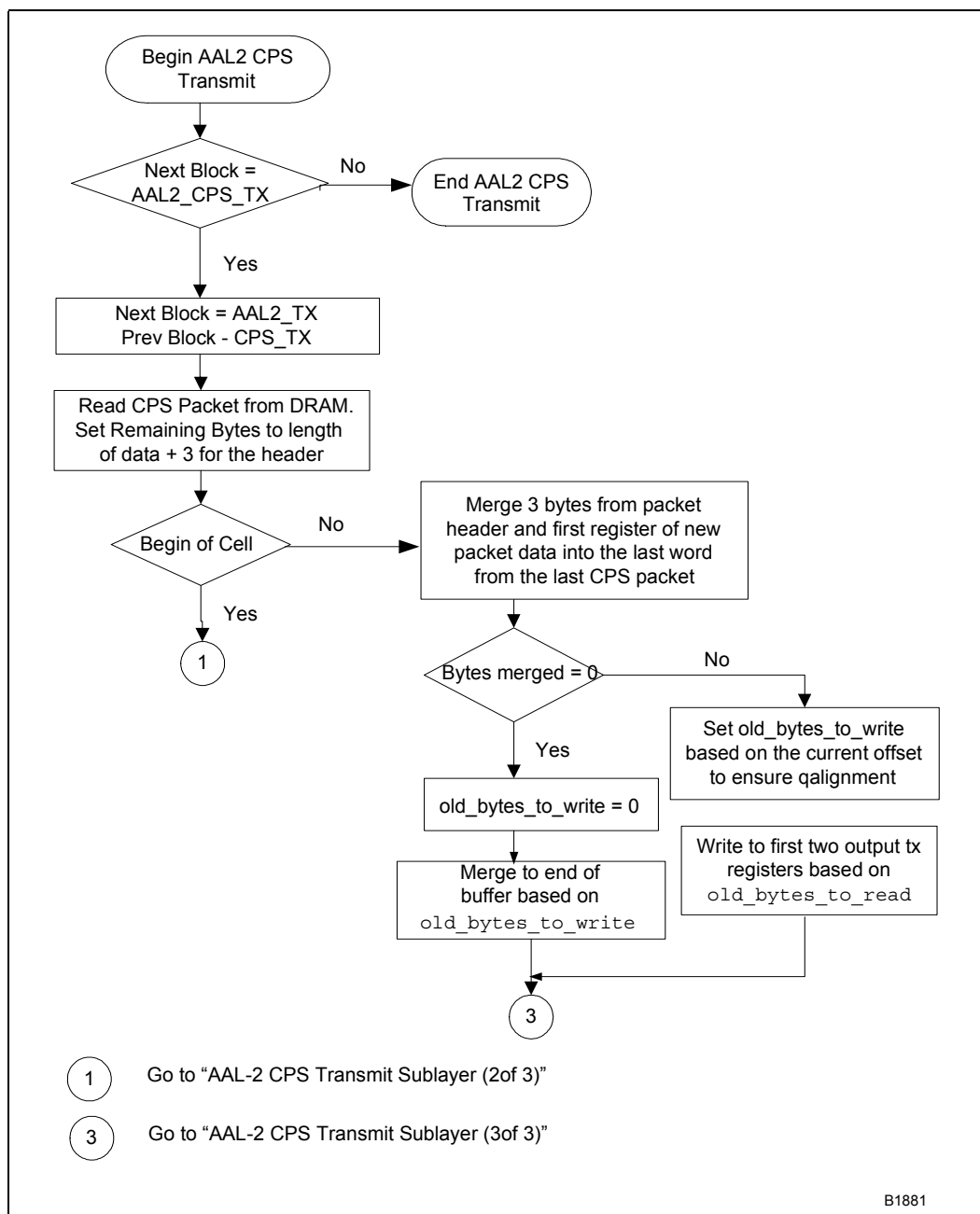


Figure C-19. AAL-2 CPS Transmit Sublayer (2 of 3)

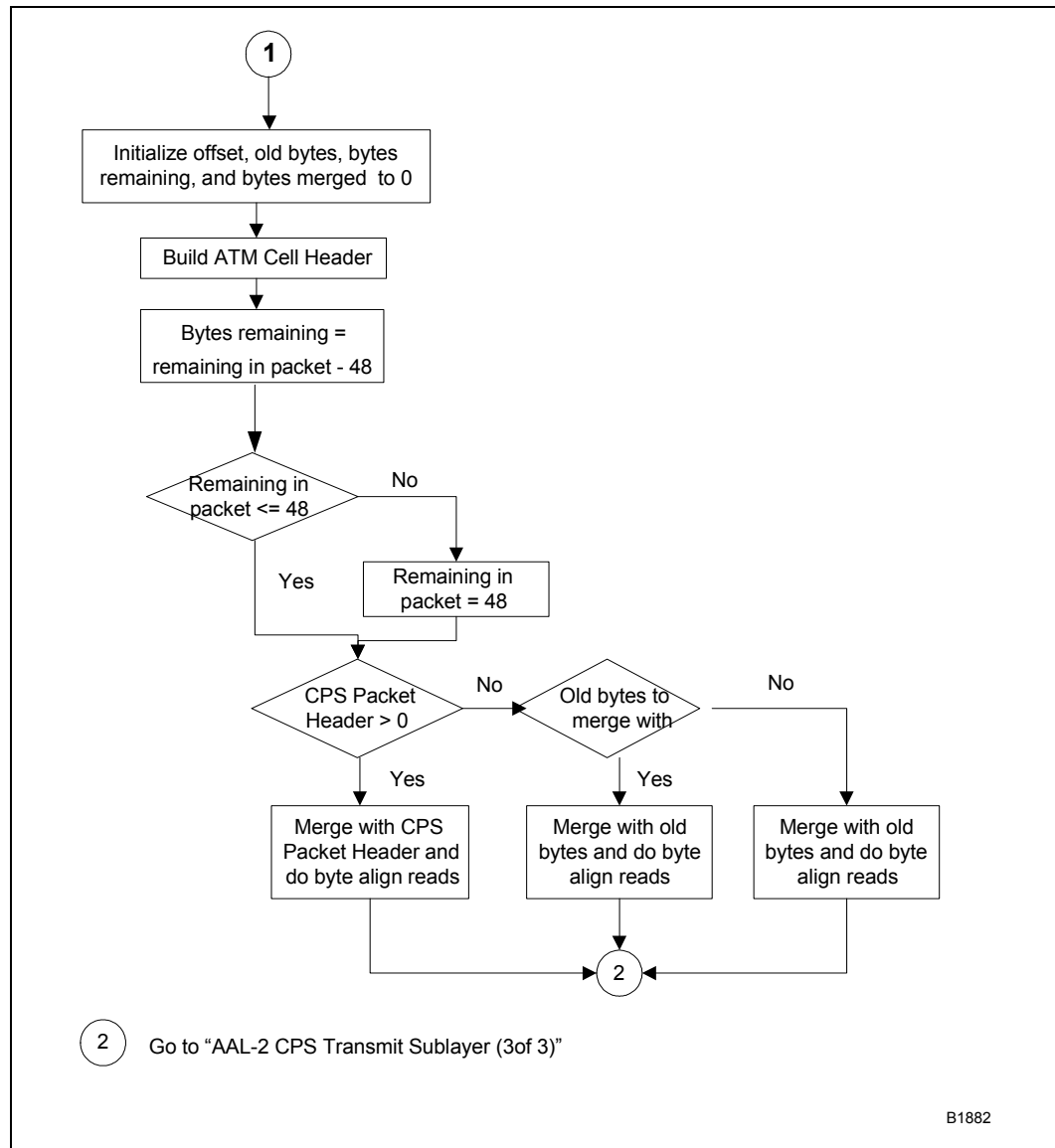
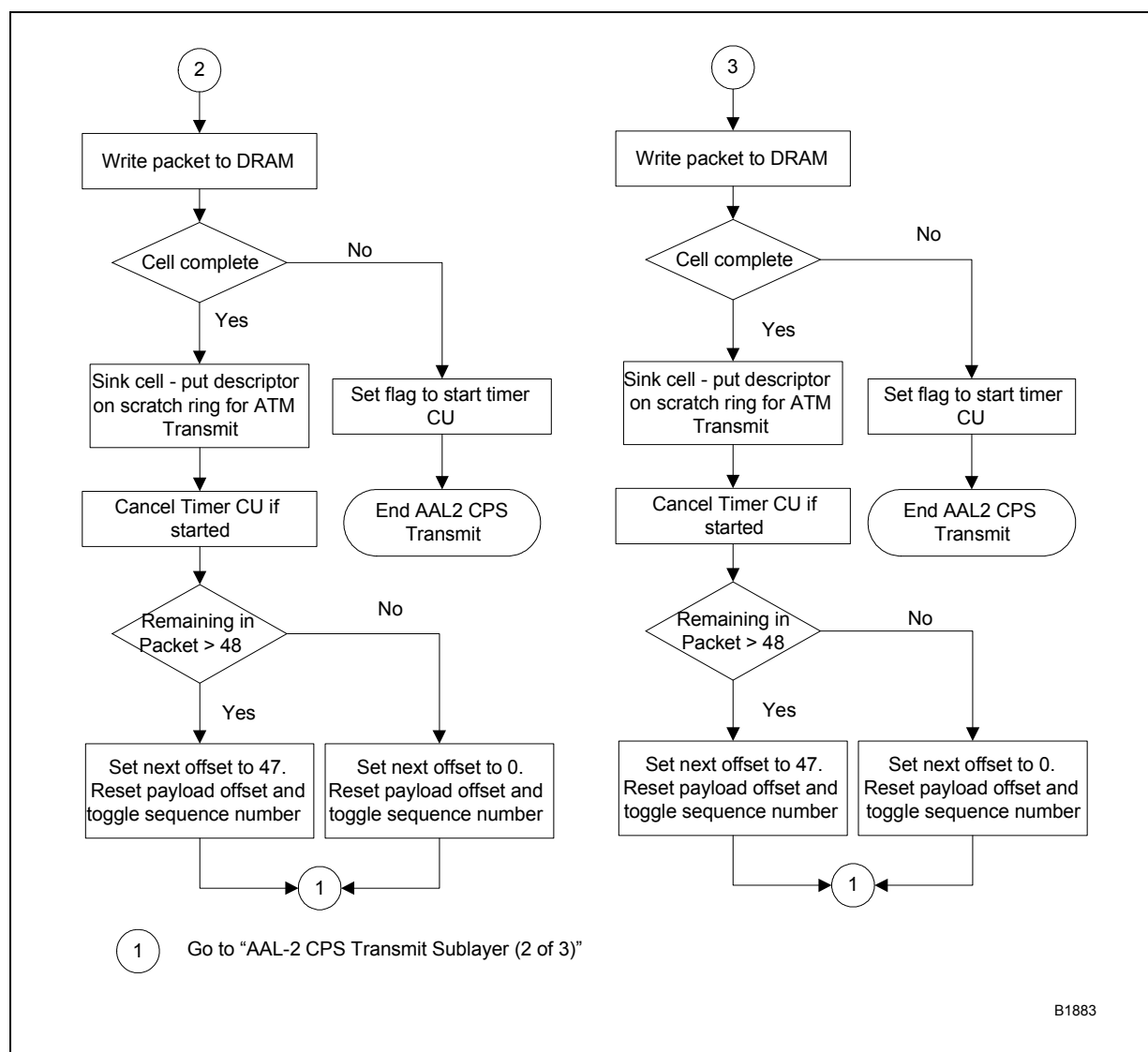


Figure C-20. AAL-2 CPS Transmit Sublayer (3of 3)



C.1.1.5. Timer CU Handling

The timer processing thread implements a time queue to provide timer services to the SSSAR/CPS processing threads. The timers are maintained in buckets where each bucket is associated with a time interval. The CPS sublayer functions register timers by updating the appropriate bucket with the VC queue number of the VC for which `Timer_CU` is to be started. The bucket to register the timer is determined using the `Timer_CU` duration indicated in the AAL parameter.

On starting the timer, the index of the bucket in which the timer was registered is stored in the transmit context of the VC. This allows canceling the timer if the ATM cell is filled before `Timer_CU` expiry. Timers are cancelled by invalidating the VC queue number written into the bucket before starting the timer.

The dedicated timer thread executes in an infinite loop. At the beginning of the loop, the thread reads the cycle count from the `TIMESTAMP_HIGH` and `TIMESTAMP_LOW` CSRs. It then registers to receive a signal when the timestamp reaches a programmable value corresponding to one time interval from current time by updating the `ACTIVE_CTX_FUTURE_COUNT` CSR. The thread then relinquishes control of the CPU and waits for the signal to be set.

On receiving the registered signal at the end of the time interval, the timer thread wakes up and generates a `Timer_CU` expired message for each VC that had `Timer_CU` set to go off in the previous time interval. The interface between the `Timer_CU` thread and the SSSAR/CPS processing threads is a next neighbor ring (configured such that the ME writes to its own NN registers).

Figure C-21. Interface Between Timer Thread and SSSAR/CPS Threads

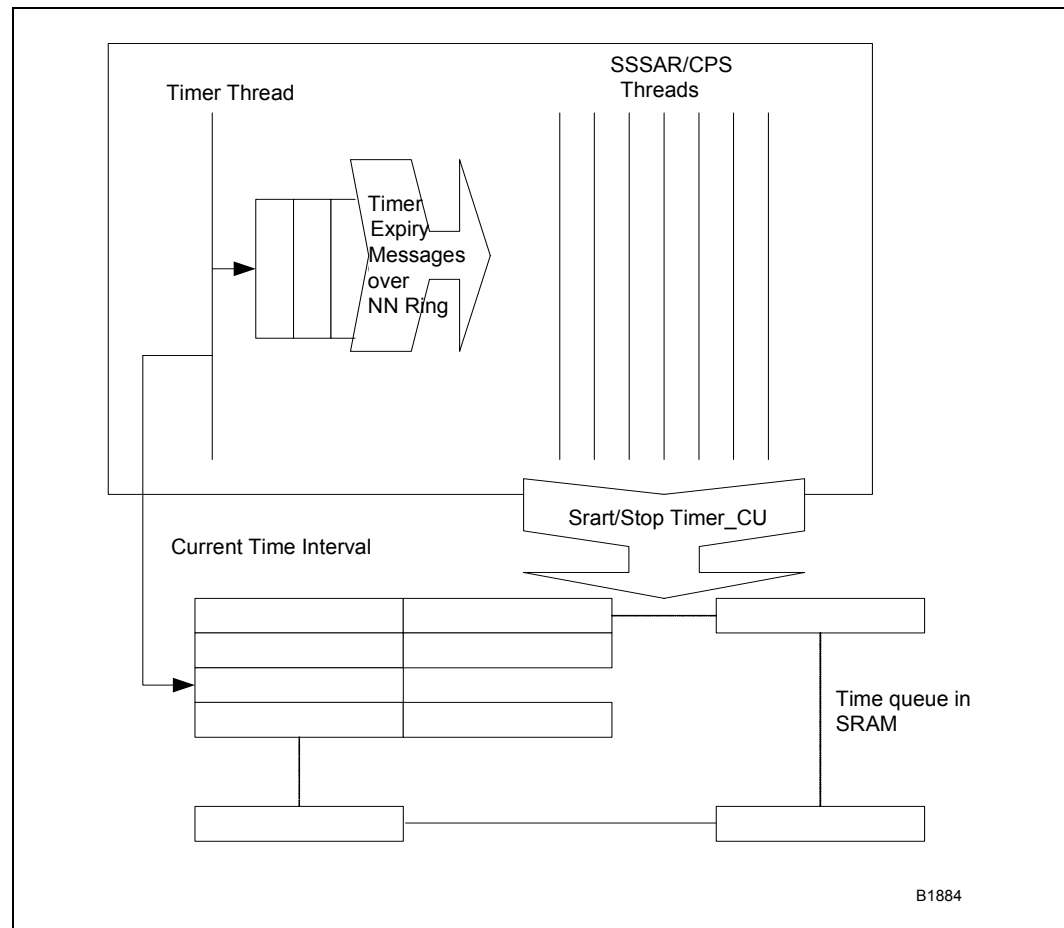
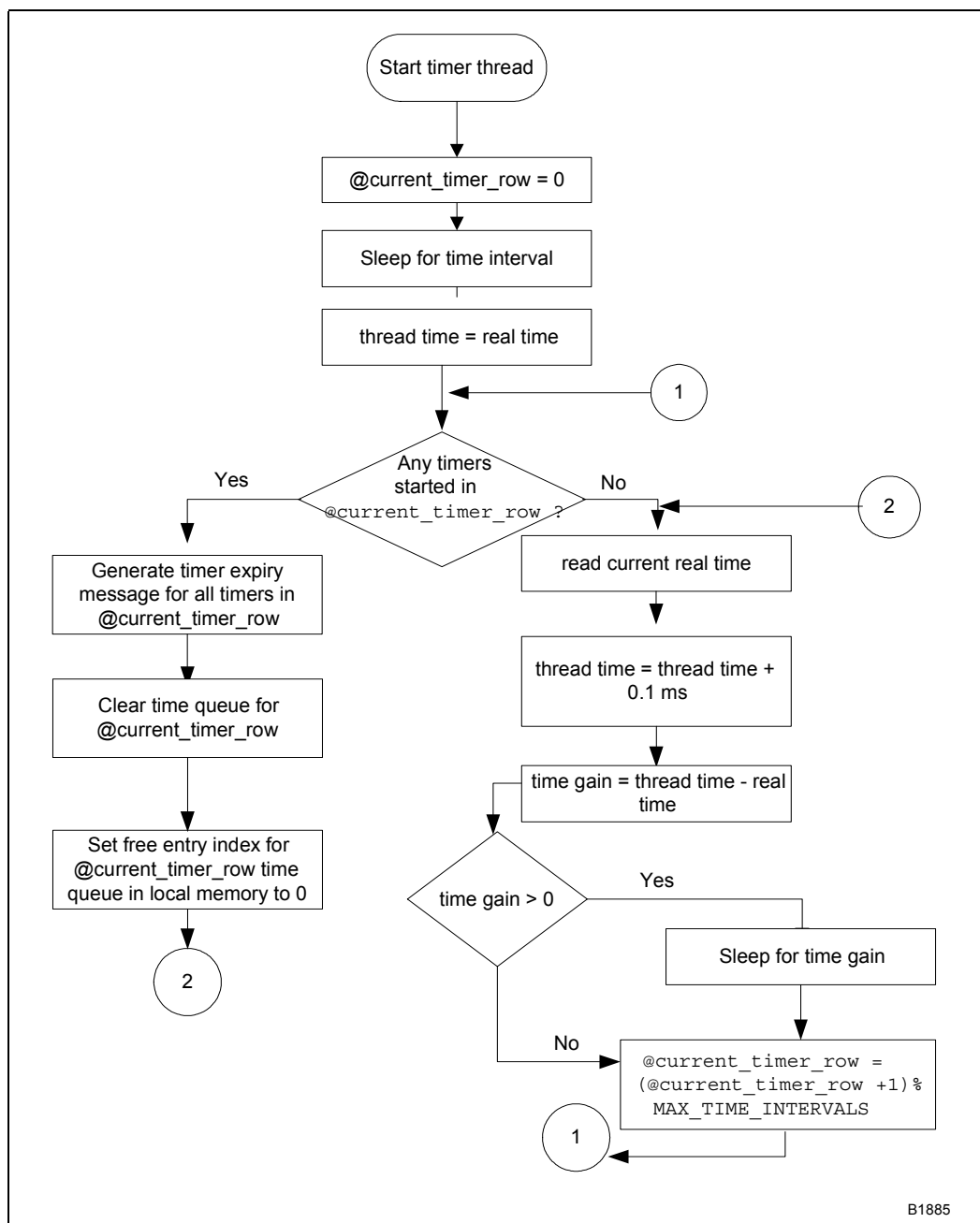


Figure C-22 shows the flowchart for the timer processing thread.

Figure C-22. Timer Processing Thread Flow



C.2 Configuration/Switches

Table C-11 shows the pre-processor switches supported by the AAL-2 Transmit microblock.:

Table C-11. AAL2 TX Supported Pre-Processor Switches

Switch	Description
AAL2_SINGLE_CPS_PER_CELL	Set this switch to disable interleaving of CPS packets in ATM cells by the CPS sublayer functions. This causes ATM cells to be generated with only one CPS packet.
AAL2_COUNTERS	When defined, statistics counters are enabled.

C.2.1 Assumptions, Dependencies and Risks

The following are the assumptions:

- The DRAM accesses are always done at 8-byte or quad-word boundaries. To overcome this restriction, the last two longwords of the CPS packet are also stored in the VC transmit context thereby allowing merge of old data in DRAM to be done with the new CPS packet data, without having read back the DRAM data from last write.
- Each thread in IXP2400 has up to 16 DRAM transfer registers and can read up to 64 bytes of data from DRAM in one access. Due to quad-word alignment restriction in DRAM access, more than one read access may be required to read the CPS SDU when CPS SDU sizes are larger than 60-bytes and the offset from where the SDU is to be read is not aligned on a 8-byte boundary.
- The AAL-2 transmit microblock uses a pool of processing threads design, which has its drawback when there are less than seven AAL-2 VCs in the system. At any time only one thread would be processing packets for a VC and all other threads enqueue transmission requests for that VC into the local memory queue for that thread. This is achieved by storing the VC queue number in the CAM along with the ID of the thread which is locking this VC for processing. When there are less than seven AAL-2 VCs in the system, only a few number of AAL-2 transmit microblocks perform actual SSSAR/CPS processing. All other threads enqueue the transmission request to the thread processing the VC. This results in inefficient usage of the parallel processing capabilities of the ME.
- The AAL-2 TX microblock supports up to 64K queues. If there are more queues, additional space are allocated in SRAM to store contexts for each queue.
- A 52 byte cell is sent out via a TBUF element. The ATM framer (hardware) adds a HEC byte in the cell header. No HEC byte is placed in the ATM cell created by AAL-2 TX microblock.

C.3 Data Structures

This section describes the data structures used for the packet transmit microblock.

C.3.1 AAL2 Transmit VC Context

The AAL2 transmit VC context is used by the CPS sublayer functions for multiplexing and packing CPS packets into ATM cells for a VC. All fields in the AAL2 Transmit VC context except LW3 are used dynamically by the AAL-2 transmit microblock and are initialized to zero at startup by the Xscale core.

Table C-12. AAL-2 Transmit VC Context Entry

LW	Bits	Size	Field	Description
0	31:0	32	Second to last LW	Second to last long word of the CPS packet data merged into the ATM cell being filled
1	31:0	32	Last LW	Last long word of the CPS packet data merged into the ATM cell being filled
2	31:17	15	Reserved	Reserved
	16:16	1	Sequence number	Used to sequence AAL2 cells. Written to STF.
	15:8	8	STF Offset	The offset which points to the first complete CPS packet in current cell. Written to STF.
	7:0	8	Payload offset	Used to track how full the current cell is. Indicates the offset in the current cell from where more CPS packet bytes can be stored.
3	31:0	32	ATM Header	ATM cell header for all cells to be transmitted on this VC. This field is configured by the Xscale core.
4	31:0	32	Cell Descriptor	Buffer descriptor of current cell being built.
5	31:31	1	Timer CU Running	Timer running bit. Indicates Timer CU started for this VC.
	30:30	1	Timer CU start requested	This bit is set by the CPS processing threads to request starting of Timer CU if there is no other CPS packet pending for processing and the current ATM cell buffer is partially filled
	23:8	16	Timer tag	Timer tag identifying the location where timer CU is started in the timer thread calendar queue. Used for cancelling the timer if required.
	7:0	8	Timer CU duration	The timer duration in 0.1 ms units as indicated in the AAL parameter for the current CPS SDU.
6	31:0	32	Reserved	Reserved
7	31:0	32	Reserved	Reserved

C.3.2 CRC-5 Tables

The AAL-2 transmit microblock uses two lookup tables to compute CRC-5 on the 19 bits of the CPS packet header. The lookup tables are stored in SRAM by the Xscale core and are read into the local memory at startup by the AAL-2 Transmit microblock as part of initialization. [Table C-13](#) shows the tables in SRAM.

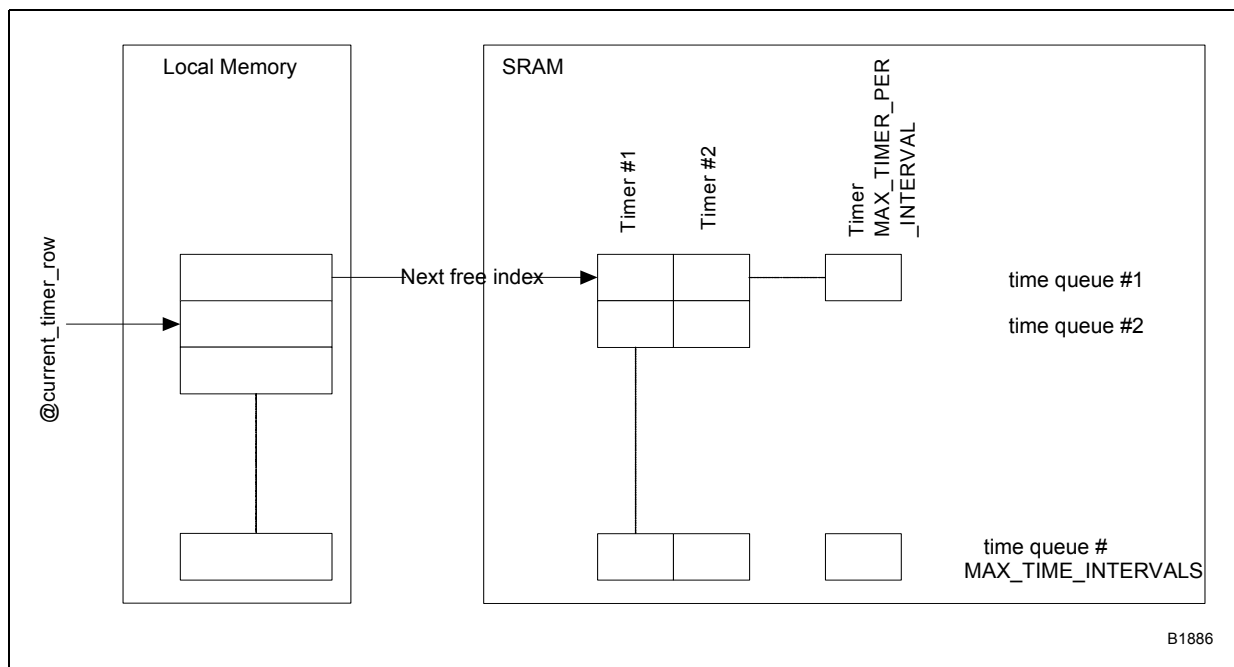
Table C-13. AAL-2 TX Microblocks CRC-5 SRAM tables

able Name	Size	Description
CRC-5 8 bit lookup table	256 bytes	CRC-5 table used for CRC result using 8 bits of CRC data.
CRC-5 3 bit lookup table	32 bytes	CRC-5 table used for CRC result using 3 bits of CRC data.

C.3.3 Timer CU Data Structure

Timer CU data structure is implemented as a collection of time queues. Each time queue represents an interval of time equal to the granularity of timer CU. Each time queue holds a maximum of `MAX_TIMERS_PER_INTERVAL` number of timers. As real time ticks, the timer processing thread reads the next time queue from SRAM and generates timer expiry messages for all VCs registered to have their timer CU expire in that time interval. A pointer `@current_timer_row` is used to maintain the current time queue being processed by the timer thread. The number of time queues is configured as `MAX_TIME_INTERVALS` to allow timer CU values ranging from 0.1 ms to 10 ms. [Figure C-23](#) shows the time queue data structure in SRAM and local memory.

Figure C-23. Timer CU Data Structures in SRAM and Local Memory



To register a timer in a given time interval, the CPS processing thread writes the VC queue number of the VC for which timer CU is to be started in the next free location in the time queue for that time interval. An array of `MAX_TIMER_INTERVALS` long words is maintained in local memory to

store the index of the next free location in the time queue. The time queue for to start a timer is obtained as $((@current_timer_row + \text{timer CU value}) \% \text{MAX_TIME_INTERVALS}) 1$; the timer CU value is expressed in units of 0.1 ms and can range from 1 to 100. The next free location in the time queue is obtained from the index maintained in local memory for that time queue. If there are no free locations available in the time queue, the next time queue is checked to see if there is space to register the timer. This process is repeated `MAX_ALLOWED_TIMER_ROW_SKIP` number of timers until a time queue is found with space to register the timer. If no time queue could be found to register the timer, a timer expiry message is generated immediately to ensure transmission of the partially filled ATM cell for that VC. The default value for `MAX_TIMER_INTERVAL` is configured as 128 and `MAX_TIMERS_PER_INTERVAL` is configured as 128. The default value for `MAX_ALLOWED_TIMER_ROW_SKIP` is configured as 10. All these values can be tuned as per specific applications requirement by suitably defining their values in `dl_system.h`.

C.3.4 Counters

The counters available via the `AAL2_COUNTERS` build switch are shown in [Table C-14](#). Each VC queue is associated with a block of four 32-bit counters.

Table C-14. AAL2 Transmit Counters

Switch	Description
<code>AAL2_NUM_TX_CELLS</code>	Number of ATM cells transmitted
<code>AAL2_NUM_TX_PKTS</code>	Number of AAL-2 packets transmitted
<code>AAL2_NUM_TX_DISCARD</code>	Number of AAL-2 packets discarded by AAL-2 TX
<code>AAL2_NUM_TX_ERR</code>	Number of erroneous AAL-2 transmit request

C.4 Performance Analysis

[Table C-15](#) shows the cycle counts and I/O latencies for processing 44 byte CPS packets.:

Table C-15. Cycle Counts and I/O Latencies

Scenario	Cycle count /thread	I/O Latency
CPS packet processing (VC not locked, VC queue not empty)	326	815
CPS packet processing (VC locked, VC queue not empty)	288	679
CPS packet processing (VC not locked, VC queue empty)	344	876
SSSAR processing on first/middle CPS packet	308	795
SSSAR processing on last CPS packet	332	558
CPS packet processing (Timer CU started)	286	560
Timer CU expiry handling (pad and transmit ATM cell)	152	502

Note: The control space usage is 2446.

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 3.1
API	Acronym for <i>application programming interface</i> .
application programming interface	(API) A set of routines, classes, methods, structures, and/or functions used to write applications.
ATM	Asynchronous Transfer Mode - A transfer mode in which information is organized into cells. It is asynchronous in the sense that the recurrence of cells containing information from an individual user is not necessarily periodic.

B

big endian	A compiler term specifying that, for multibyte values, the most significant byte is first. See also <i>little endian</i> , <i>network byte order</i> .
byte order	The way a system stores numeric data, with the most or least significant byte first. Most significant byte first, or <i>big endian</i> byte order, is also known as <i>network byte order</i> . See also <i>endianness</i> .

C

CBR	Constant Bit Rate—an ATM service class.
CBS	Committed Burst Size—an IP QoS traffic contract parameter/metric.
CIR	Committed Information Rate—an IP QoS traffic contract parameter/metric.
CLP	Cell Loss Priority—an ATM QoS traffic contract parameter/metric.

content addressable memory	(CAM) This is a hardware feature where a content match is performed to get an index to associated information.
context pipeline	A software pipeline where in different functions are performed on different microengines as time progresses and the packet context is passed between the functions or microengines. Each microengine constitutes a context pipe-stage and cascading two or more context pipe-stages constitutes a context pipeline. The context pipeline get it's name from the fact that it is the context that moves through the pipeline.
control plane	The abstraction for a functional area of an application that controls and configures the data plane and handles exception packets, distinguished from the <i>data processing plane</i> . Control plane activities are typically performed by <i>code modules</i> within the <i>IXA application</i> . Compare <i>management plane</i> , whose activities are usually outside the IXA application, in a <i>host</i> application.
core component	A packet-processing entity that configures its microblock, initializes and maintains common data structures that may be updated by other applications, and provides exception as well as control message handlers to process packets/messages sent by the microblock.
core component infrastructure	The core component infrastructure includes a number of APIs to support the creation and setup of core components.
CRC	Cyclic Redundancy Check - A mathematically computed numerical value transmitted with packet data to ensure the integrity of packet data transmitted between endpoints.
critical section	A critical section is section of code in which only one microengine thread has exclusive modification privileges for a global resource (such as a memory location) at any one time. The IXP2400 uses inter-thread signaling to implement critical sections across microEngines.
CSR	Acronym for <i>control status register</i> .

D

Decap	Decapsulation - Removing one or more protocol headers from a packet.
DiffServ	Differentiated Service. A means of classifying IP packets into “classes”, based on the DiffServ codepoint (DSCP) in the packet's IP header.
Dispatch Loop	A Dispatch Loop combines microblocks running on a microengine thread and implements the data flow between them.
DRR	Deficit Round Robin. A QoS queue-scheduling algorithm.
DSCP	DiffServ Code Point. A 6-bit field in the IPv4 header.

E

EE	Acronym for execution engine.
Encap	Encapsulation - Adding one ore more protocol headers to a packet.

endian, endianness	A compiler term for the <i>byte order</i> of multibyte values. See <i>big endian</i> and <i>little endian</i> .
Ethernet	A local area network (LAN) technology designed for interconnecting networking nodes over a shared medium, as specified in standard IEEE 802.3. Also typically used to refer to the Layer 2 networking protocol as specified in standard IEEE 802.2.

F

Fast Path	The data path where in the packet is completely processed on the MEv2 microengines without any intervention from the XScale Core.
Folding	A software technique used by threads running on the same ME, to optimize read/modify/writes in a critical section. The technique uses the CAM and strict thread ordering enforced via inter-thread signaling to fold the read/modify/write into a single read, multiple modifies and one or more writes depending on the cache eviction policy.
Functional Pipeline	A software pipeline where in the context remains with an microengine while different functions are performed on the packet as time progresses. The microengine execution time is divided into “n” pipe-stages and each pipe-stage performs a different functions. The Functional pipeline get it's name from the fact that it is the function that moves through the pipeline.

G

GFR	Guaranteed Frame Rate - an ATM service class.
-----	---

H

Head of Line Blocking	A term used to describe a situation where the transmit operation on a group of ports is blocked by a single port at the head of the transmit queue. This scenario typically occurs when the port at the head of the transmit queue is blocked because of flow control issues and the remaining ports on the queue have data pending but need to wait for this port to finish its transmit operation.
HEC	Header Error Check - An 8-bit field within an ATM header that is generated by a sender, and checked by a receiver, to determine the validity of an ATM header.

I

Intel® Internet Exchange Architecture	(IXA) A new approach to designing networking and telecommunications equipment based on reprogrammable silicon and open interfaces. Manufacturers of networking and communications equipment can use components from the IX-based product portfolio for designing new, more intelligent network systems.
---------------------------------------	---

intrinsic	A C function-like interface that implements a chip-specific hardware feature, not otherwise supported by the C language. Direct use of intrinsics results in non-portable code.
IP	An acronym for <i>internet protocol</i> , a standard network protocol. See also <i>TCP/IP</i> .
IPv4	Internet Protocol Version 4.
IPv6	Internet Protocol Version 6.
IXP	Acronym for <i>Intel</i> [®] <i>Internet Exchange Processor</i> , and a current instance of this processor.
IXP2400, IXP 2800	Internet eXchange network processors. The IXP2400 has 8 microengines targeted at OC-48 POS line rates and the IXP2800 has 16 microengine targeted at OC-192 POS line rates.

J

K

L

L2	Layer 2.
L3	Layer 3.
LLCSNAP	Logical Link Control/Sub Network Access Protocol - Data link layer packet encapsulation headers that identify a protocol, as well as client and control information. Refer to IEEE standards 802.3 with 802.2.
LPM	Longest Prefix Match—algorithm IP routers apply to an IP packet destination address to determine the packet's egress port, and hence forward the packet out the egress port.
little endian	A compiler term specifying that, for multibyte values, the least significant byte is first. See also <i>big endian</i> .
longword	A 32-bit word; 4 bytes long.

M

MAC	Medium Access Control—A protocol layer responsible for providing access to a shared communications medium. Also Medium Access Controller - The device used to interface with the physical layer medium.
ME	An acronym for <i>microengine</i> .
MEv2	A microengine specific to the IXP2xxx network processor family.
Microblock	A discrete unit of IXP2xxx code written in microcode or MicroC that is written to the guidelines specified in the IXA Software Framework. Microblocks conform to one of three different types: source, transform or sink. Typically, a microblock has an XScale component that is used to configure and manage the microblock.

Microblock Group	One or more microblocks that have been combined into a thread executable on a microengine. Typically all threads on the microengine executes the same microblock group, but it is not required. Furthermore, a typical use instantiates the same microblock group on several microengines.
Microengine	One of many (8 for IXP2400, 16 for IXP2800) programmable, specialized processors.
Mixed Pipeline	A software pipeline where some microengines run a single function (context pipe-stage) and others run multiple functions (functional pipeline)
MPKT	M Packet - An IXP2xxx media bus interface data transfer unit that can be configured to be 64, 128, or 256 bytes in length.
MEv2	Microengine version 2, which are the microengines used for the IXP2400 and IXP2800 network processors.
microcode	Hardware-specific machine code. A <i>code module</i> written in microcode can run only on the processor it is written for.
mutual exclusion, mutex	Mutual exclusion is used to guard the critical sections accessed by threads.

N

nrtVBR	Non-Real Time Variable Bit Rate—an ATM service class.
network byte order	The system of storing numeric data with the most significant byte first. See also <i>big endian</i> , <i>endianness</i> .
network services application	General descriptive term for the kind of application you build with the <i>IXA SDK</i> .

O

OAM	Operations Administration and Maintenance—A group of network management functions that provide network fault indication, performance information, and data and diagnosis functions within an ATM network. Also the type of ATM cell payload used to carry such information.
OC-12, OC-48c	Optical Carrier (SONET) - Level (e.g. Level = 3, 12, 48, 192). Often used to specify data rates; the base level rate is 51.84 Mbps (OC-1); each level thereafter operates at a multiple of the base level rate (thus, OC-3 runs at 155.52 Mbps, OC-12 runs at 622.08 Mbps, etc).
OS	Acronym for <i>operating system</i> .
OSSL	Acronym for <i>operating system services library</i> .
Operating System Services Library	(OSSL) An OS abstraction API used within the 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 runs on IXP1200, IXP2400, IXP2800 and future IXP chips.

P

payload

The part of a packet that carries data, as opposed to those parts that carry information about the packet.

Q

quadword

A 64-bit word; 8 bytes long.

QoS

Acronym for *quality of service*.

quality of service

A networking term that specifies a guaranteed throughput level. (*QoS*)

R

Resource Manager

A programming interface between Intel XScale® core applications and the microcode running on the microengines for the IXP2400 and IXP2800 network processors.

RR

Round Robin. A scheduling algorithm in which entities/queues are services/scheduled in a consistent serial manner.

rtVBR

Real Time Variable Bit Rate - an ATM service class.

RX

Receive.

S

SAR

Segmentation And Reassembly—The process of transforming frames-to-cells and cells-to-frames.

SDE

Acronym for *software development environment*.

SDK

Acronym for *software development kit*.

semaphore

Semaphores are the primary means for providing thread synchronization.

sink microblock

A function or macro that disposes of a packet, that is, either enqueues it within the IXP or sends it to an external interface.

slow path

The execution path of the packets that require exceptional handling. This may be error packets or packets that need to be handled differently than the normal case. In this case, it takes longer to process because they are handled by a general-purpose processor (Intel XScale® core in our case). See also *Fast Path*.

software pipeline

The MEv2 employs a software pipeline model in the fast path processing of packets. There are three different types of pipelines—*Context Pipeline*, *Functional Pipeline*, and *Mixed Pipeline*.

source microblock	A function or macro that obtains a packet, that is, either dequeues it within the IXP or gets it from an external interface.
SP	Acronym for <i>scheduling policy</i> .
stdmac	Acronym for <i>standard macros</i> . Assembly macros that are microengine-specific, for instruction simplification.

T

TCP	An acronym for <i>transmission control protocol</i> , a standard network protocol in which transmission status can be confirmed. Establishes a point-to-point connection, in contrast to <i>UDP</i> which is connectionless. See also <i>TCP/IP</i> .
TCP/IP	A standard network protocol, using <i>TCP</i> over <i>IP</i> . See <i>TCP</i> and <i>IP</i> .
TM4.1	Traffic Management version 4.1 - An ATM specification for managing/controlling traffic congestion within an ATM network by the actions of buffering, adjusting transmission rates, and policing VCs.
TOS	Type of Service. Refers to an 8-bit field in the IPv4 header.
Tx	Transmit.
thread	A thread is an independent task, which can be processed in parallel with other tasks.
transform microblock	A function or macro that parses, analyzes, classifies, or modifies a packet.

U

UBR	Unspecified Bit Rate - an ATM service class.
UPC	Usage Parameter Control—VC traffic contract characteristics, that permit ATM network nodes to monitor, control, and police the traffic within the ATM network.

V

VBR	Variable Bit Rate - an ATM service class.
VC	Virtual Connection or Virtual Channel— A communications channel between ATM systems nodes that provides for the sequential transport of ATM cells.
VCI	Virtual Connection Identifier—A 16-bit numerical tag within an ATM cell header that identifies a virtual channel over which the cell is to travel.
VPI	Virtual Path Identifier—An 8-bit numerical tag within an ATM cell header that indicates the virtual path over which the cell should be routed.
VPN	Virtual Private Network.

VPORT Virtual Port - A field accompanying a MPKT that identifies the port (and possibly line card) to/from which the MPKT payload is sent/received.

W

WAN Wide Area Network—A network that spans a large geographical area relative to a LAN (Local Area Network). A WAN typically experiences greater traffic delays (due to distance between nodes and greater network congestion) and packet loss (due to switches dropping packets).

WRR Acronym for *weighted round robin*.

X

XScale core The ARM architecture core processor in the IXP2400 and IXP2800 network processors.

Y

Z



1 Introduction51

- 1.1 About this Manual51
 - 1.1.1 Microblocks51
 - 1.1.2 Core Components52
- 1.2 Organization of the Manual52
- 1.3 Other Sources of Information52

Applications

2 System Data Structures and Design Choices57

- 2.1 Buffer Handle57
- 2.2 Packet Meta Data (Buffer Descriptor)58
- 2.3 Optimizing DRAM Bank Scheduling for Buffers59
- 2.4 Buffer Chaining59
 - 2.4.1 Flat Queueing (Cell Based Dequeue)59
 - 2.4.2 Hierarchical Queueing (Packet Based Dequeue)61
- 2.5 Statistics and Handling of 64-bit Counters63
 - 2.5.1 Using SRAM/Scratch Atomic Operations64
 - 2.5.2 Read-Modify-Write in Critical Sections64
- 2.6 Implementation of Dropping Packets65
- 2.7 Global Build Switches65

3 System Data Structures for Packet Replication67

- 3.1 Packet Replication67
 - 3.1.1 Buffering Design67
 - 3.1.2 Child Buffer68
 - 3.1.3 Cell Mode68
 - 3.1.4 Packet Mode69
- 3.2 Parent and Child Meta Data (Buffer Descriptor)70
 - 3.2.1 Cell Mode70
 - 3.2.1.1 Child Meta Data70
 - 3.2.1.2 Parent Meta Data71
 - 3.2.2 Packet Mode72
 - 3.2.2.1 Child Meta Data72
 - 3.2.2.2 Parent Meta Data73
- 3.3 Child Buffer Freelist73
- 3.4 Packet Copier73
- 3.5 Dropping of Packet Copies74
 - 3.5.1 Dropping Packet Copies in Packet Mode74
 - 3.5.2 Dropping Packet Copies in Cell Mode74
- 3.6 Differentiating Child Buffer from Parent Buffer74
- 3.7 Build Switches75

Microblocks 77

Receive

4 Packet RX Microblock81

- 4.1 Overview81
- 4.2 Assumptions and Dependencies81

4.3	Packet RX State Machine	82
4.4	Data Structures	83
4.4.1	Receive Reassembly Context (RXC)	83
4.4.2	Statistics	84
4.4.3	Jump Table	84
4.5	Build Switches	85
4.6	Algorithm	85
4.6.1	Single Microengine Design	85
4.6.2	Two Microengine Design	86
4.7	Flow Chart For Single Microengine Design	87
4.8	Performance Analysis	91
4.8.1	Characterization Data	92
5	CSIX RX Microblock	95
5.1	Overview	95
5.2	Assumptions and Dependencies	95
5.3	Basic Algorithm	96
5.4	Optimizations to the Basic Algorithm	97
5.4.1	Optimizations for the Single C-Frame Packet	97
5.4.2	Optimizing the DRAM Bank Utilization	98
5.5	Extending the Design to Run on Two Microengines	98
5.6	Data Structures	99
5.6.1	Receive Reassembly Context (RXC) for One Microengine Design	99
5.6.2	Receive Reassembly Context (RXC) for Two Microengine Design	99
5.6.3	Lookup Key	100
5.6.4	Statistics	100
5.7	Build Switches	101
5.8	Performance Analysis	101
5.8.1	Characterization Data	102
6	ATM AAL5 RX Microblock	105
6.1	Overview	105
6.1.1	ATM Terminology	105
6.1.2	ATM Layer Functions	106
6.1.3	AAL5-SAR Functions	106
6.1.4	CPCS Functions	106
6.1.5	Design Overview	106
6.1.5.1	OC-48 (SPHY_1_32)	107
6.1.5.2	Quad OC-12 (SPHY_4_8) or OC-24/OC-12 (SPHY_1_32)	107
6.1.5.3	A Maximum of 2048 Ports Addressing	108
6.2	Configuration/Switches	108
6.3	Assumptions, Dependencies and Risks	110
6.4	Data Structures	111
6.4.1	Reassembly Context (RXC) or VC Info table	111
6.4.2	AAL5 RX Counters	113
6.4.3	Hash Table	113
6.4.3.1	Collisions Resolutions	115
6.5	Algorithm and Pseudocode	117
6.5.1	Issues and Challenges in the Implementation	117
6.5.2	Algorithm	118

- 6.5.2.1 Two Microengine Design118
 - 6.5.2.2 Single Microengine Design120
- 6.6 Flow Chart (Two Microengine Design)122
- 6.7 Performance Analysis130
 - 6.7.1 Two Microengine Design130
 - 6.7.2 Single Microengine Design (4 threads per OC-12 port)131
- 6.8 Resource Usage132
 - 6.8.1 Control store132
 - 6.8.2 Registers and Signals132
 - 6.8.3 I/O Commands133
 - 6.8.4 Characterization Data134

Transmit

7 CSIX TX Microblock141

- 7.1 Overview141
- 7.2 Assumptions and Dependencies141
- 7.3 Data Structures142
 - 7.3.1 Transmit Context (TxC)142
 - 7.3.2 Transmit Control Word (TCW)142
 - 7.3.3 TM Header—per C-Frame)143
 - 7.3.4 Packet Header (added only to first c-frame)143
 - 7.3.5 Reference CSIX Base Header—One Shortword144
 - 7.3.6 Reference CSIX Unicast Extension Header144
 - 7.3.7 Statistics144
- 7.4 Design145
 - 7.4.1 Interleaving Multiple Requests145
 - 7.4.2 Optimizing for the Minimum Packet Case146
 - 7.4.3 Running the Microblock on Two Microengines146
- 7.5 Flow Chart for Single Microengine Design147
- 7.6 Performance Analysis150
 - 7.6.1 Characterization Data151

8 Packet TX for SPHY and MPHY-4155

- 8.1 Overview155
- 8.2 Assumptions and Dependencies156
- 8.3 Data Structures156
 - 8.3.1 Context-Relative Thread Execution Status Flag156
 - 8.3.2 TX Request—One Longword157
 - 8.3.3 Queue Structure for Each Port157
 - 8.3.3.1 Queue Entry Structure157
 - 8.3.3.2 Queue Descriptor Structure158
 - 8.3.4 Output Transmit Control Word—Two Longwords159
 - 8.3.5 Statistics159
- 8.4 Build Switches159
- 8.5 Design160
- 8.6 Flow Chart161
- 8.7 Performance Analysis168
 - 8.7.1 Characterization Data168

9	Packet TX for MPHY-16	173
9.1	Overview	173
9.2	Assumptions and Dependencies	174
9.3	Data Structures	174
9.3.1	Globals Stored in Absolute Registers	174
9.3.2	Context Relative Thread Execution Status Flag— <code>exe_stat_flag</code>	174
9.3.3	TX Request—One Longword	175
9.3.4	Queue Structure for Each Port	175
9.3.4.1	Queue Entry Structure	175
9.3.4.2	Queue Descriptor Structure	176
9.3.5	Output Transmit Control Word —Two Longwords	177
9.3.6	Statistics	177
9.4	Design	178
9.5	Flow Chart	179
9.6	Performance Analysis	184
9.6.1	Characterization Data	184
10	Packet Transmit for OC-192 POS	189
10.1	Overview	189
10.2	Assumptions and Dependencies	189
10.3	Data Structures	190
10.4	Design	191
10.5	Performance Analysis	192
10.5.1	Characterization Data	192
11	ATM AAL5 TX Microblock	197
11.1	Overview	197
11.1.1	CPCS Functions	197
11.1.2	AAL5-SAR Functions	198
11.1.3	ATM Layer Functions	198
11.1.4	Extended Port Addressing	199
11.1.4.1	FPGA Addressing	199
11.1.5	Design Overview	200
11.1.5.1	OC-48200	
11.1.5.2	Quad OC-12201	
11.2	Assumptions, Dependencies and Risks	201
11.3	Data Structures	202
11.3.1	Packet Metadata	202
11.3.2	Transmit Context (TXC)	202
11.3.3	Counters	204
11.3.4	Switches	204
11.4	Algorithm	205
11.4.1	OC-48—One Page Summary	205
11.4.2	Quad OC-12—One page Summary	206
11.4.3	OC-48—Second Level of Detail	207
11.4.4	Quad OC-12—Second Level of Detail	209
11.5	Performance Analysis	210
11.5.1	OC-48211	
11.5.2	Quad OC-12211	

11.5.3	Characterization Data	212
11.6	Possible Optimizations	215
12	Packet TX—Multiports Microblock	217
12.1	Overview	217
12.2	Assumptions and Dependencies	218
12.3	Data Structures	218
12.3.1	Globals Stored in Absolute Registers	219
12.3.2	Context Relative Thread Execution Status Flag	219
12.3.3	TX Request—One Long Word	220
12.3.4	Packet Queue Structure for Each Port	220
12.3.4.1	Packet Queue Entry for Each Port	220
12.3.4.2	Port Status Info for Each Port—16 Long Words	221
12.3.5	Output Transmit Control Word—2 Long Words	223
12.3.6	Statistics	223
12.4	Design	223
12.4.1	Picked_from_tx_request	224
12.4.2	Picked_from_virtual_queue_or_picked_none	226
12.5	Performance Analysis	229
12.5.1	Performance Analysis for IXP24XX	229
12.5.2	Performance Analysis for IXP28XX	230
12.5.3	Characterization Data	230

Queue Manager

13	Queue Manager For OC-48 Microblock	237
13.1	Overview	237
13.2	Assumptions/Dependencies	238
13.3	Data Structures	238
13.3.1	Queue Descriptor	238
13.3.2	Packet Counts in Local Memory	239
13.4	Build Switches	239
13.5	Design	239
13.6	Differences Between Cell and Packet QM	243
13.6.1	Q-Array Hardware Configuration	243
13.6.2	Hierarchical Queuing	243
13.6.3	Dequeue Response	243
13.6.4	Multiple Queues on Transmit	243
13.7	Flow Chart	243
13.8	Performance Analysis	246
14	Queue Manager For OC-192 Microblock	247
14.1	Overview	247
14.2	Assumptions/Dependencies	248
14.3	Data Structures	248
14.3.1	Queue Descriptor	248
14.4	Build Switches	249
14.5	Design	249
14.6	Differences between OC-48 and OC-192 QMs	251

14.7	Performance Analysis	251
14.7.1	Characterization Data	252
15	Packet Queue Manager Microblock	255
15.1	Overview	255
15.2	Q-Array Hardware Configuration	255
15.3	Hierarchical Queuing	255
15.4	Dequeue Response	255
15.5	Multiple Queues on Transmit	256
15.6	Performance Analysis	256
15.6.1	Characterization Data	256
16	ATM Queue Manager Microblock	259
16.1	Overview	259
16.2	Assumptions/Dependencies	259
16.3	Data Structures	259
16.3.1	Queue Descriptor	260
16.3.2	Packet Counts in Local Memory	260
16.4	External Interfaces	261
16.4.1	LLCSNAP Encap and Cell QM	261
16.4.2	ATM QM and TM 4.1 Shaper	261
16.4.3	TM 4.1 Scheduler to ATM QM	262
16.4.4	ATM QM and AAL-5 TX	262
16.5	Build Switches	263
16.6	Design	263
16.7	Flow Chart	266
16.8	Performance Analysis	270
16.8.1	Characterization Data	271
Scheduler Microblocks		
17	Fabric Scheduler For OC-48	277
17.1	Overview	277
17.2	Assumptions, Dependencies and Risks	277
17.3	Data Structures	278
17.3.1	Queue and Queue Groups	278
17.3.2	Globals	279
17.3.3	Queue Group Data Structure	279
17.3.4	Queue Data Structure	279
17.4	Design Decomposition	280
17.4.1	Scheduler Thread	280
17.4.2	QM Message Handler Thread	280
17.4.3	Flow Control Handler Thread	280
17.4.4	Packets In Flight Handler Thread	281
17.5	Flow Chart for Scheduler Thread	282
17.6	Flow Chart for Flow Control Thread	285
17.7	Flow Chart for QM Message Handler Thread	285
17.8	Performance Analysis	287
18	Fabric Scheduler For OC-192	289
18.1	Overview	289

18.2	Assumptions, Dependencies and Risks	289
18.3	Data Structures	289
18.3.1	VoQ Data Structure	290
18.3.2	Globals	290
18.4	Design	291
18.4.1	Flow Control Handling	292
18.4.2	Packets In Flight Handling	293
18.5	Implementing a Hierarchical Scheduler	294
18.6	Performance Analysis	297
18.6.1	Characterization Data	297
19	OC-48 WRR/DRR Packet Scheduler	301
19.1	Overview	301
19.2	Assumptions, Dependencies and Risks	301
19.3	Design Decomposition	302
19.3.1	Scheduler thread	302
19.3.2	QM Message Handler thread	303
19.4	Flow Control	303
19.5	Intel XScale® core Interface	303
19.6	Data Structures	303
19.6.1	Port Specific Data Structure	304
19.6.2	Queue Specific Data Structure	304
19.6.3	Data Structures in Registers	305
19.7	Algorithm and Pseudo Code	305
19.7.1	Issues/Challenges in implementation	305
19.7.2	Scheduler Thread	306
19.7.3	QM Message Handling thread	309
19.7.4	Flow Chart for Scheduler Thread	311
19.7.5	Flow Chart for QM Message Handler Thread	313
19.8	Performance Analysis	315
20	OC-192 DRR Egress Scheduler	317
20.1	Overview	317
20.2	DRR Algorithm	317
20.2.1	Traditional DRR	317
20.2.2	The Pre-Sorted DRR algorithm	318
20.3	Assumptions and Dependencies	319
20.4	Design Decomposition	320
20.4.1	Enqueueing	320
20.4.2	Dequeueing	321
20.4.3	Class Schedule Block	321
20.4.3.1	Interface	321
20.4.3.2	Pseudo Code	322
20.4.4	Count Block	323
20.4.4.1	Interface	323
20.4.4.2	Pseudo Code	323
20.4.5	Port Schedule Block	324
20.4.5.1	Interface	324
20.4.5.2	Pseudo Code	324
20.5	Other Features	326
20.5.1	Flow Control	326

	20.5.2	WRED Support	326
20.6		Data Structures	326
	20.6.1	Queue Specific Data Structure	326
	20.6.2	Port Specific Data Structure	327
	20.6.3	Packets Counter for Each Round	328
	20.6.4	Current Round for Each Port	328
	20.6.5	Data Structures in Registers	328
20.7		Performance Analysis	329
	20.7.1	Characterization Data	329
21		Egress Queue Manager (DiffServ) Microblock	333
	21.1	Overview	333
	21.2	Assumptions and Dependencies	333
	21.3	Microblock Interfaces	333
	21.4	Data Structures	333
	21.5	Flow Chart	334
		21.5.1 Synchronization	334
		21.5.2 Algorithm	334
	21.6	Micro-code Budget	335
		21.6.1 Performance Analysis	335
		21.6.2 Memory Footprint Analysis	336
22		Egress Scheduler (DiffServ) Microblock	337
	22.1	Overview	337
	22.2	Assumptions and Dependencies	339
	22.3	Microblock Interfaces	340
		22.3.1 Input Microblock Variables	341
		22.3.2 Output Microblock Variables	341
	22.4	Design Decomposition	342
	22.5	Data Structures	342
	22.6	Flow Chart	344
		22.6.1 Inter-thread synchronization	344
		22.6.2 QM Message Handler Thread Algorithm	345
		22.6.3 Scheduler Thread Algorithm	348
	22.7	Micro-code Budget	351
		22.7.1 Performance Analysis	351
		22.7.2 Memory Footprint Analysis	351
23		TM4.1 Shaper and Scheduler	
		Microblock	353
	23.1	ATM TM4.1 overview	353
		23.1.1 What is TM4.1	353
			23.1.1.1 GCRA(T, τ)
		23.1.2 Service Classes in TM4.1	354
			23.1.2.1 CBR
			23.1.2.2 rtVBR
			23.1.2.3 nrtVBR
			23.1.2.4 UBR (Plain UBR)
			23.1.2.5 UBR with PCR (UBR+)
			23.1.2.6 Differentiated UBR
			23.1.2.7 GFR

23.1.2.8	ABR	355
23.2	Implications of TM4.	1355
23.3	Design Overview	355
23.3.1	Software Blocks Overview	356
23.3.2	Time Queue Data Structure (TQ)	357
23.3.2.1	SRAM Time Queues For Low Bit Rate Traffic	358
23.3.3	Local Memory Time Queue for High Bit Rate Traffic	360
23.3.3.1	What exactly are high and low bit rate VCs?	361
23.3.4	TM4.1 Conformance for Low Bit Rate VCs	361
23.3.4.1	GCRA Conformance	361
23.3.4.2	Delay conformance	361
23.4	External Interfaces and Communication Data Structures	362
23.4.1	Interface Between the QM and the GCRA	362
23.4.2	Interface Between the GCRA and the Write-out/Scheduler	363
23.4.3	Interface Between the Write-out/Scheduler and the QM	363
23.5	Design Details	364
23.5.1	GCRA Shaper	364
23.5.1.1	Data Structures	364
23.5.1.2	Overview of Operation	365
23.5.1.3	Shaping Decision for Low Bit Rate VCs	365
23.5.1.4	Shaping Decision for High Bit Rate VCs	365
23.5.1.5	Shaping Macro	365
23.5.1.6	Pseudocode	366
23.5.1.7	Divide by 53	367
23.5.2	Write-out Block	368
23.5.3	Data Structures	368
23.5.3.1	Queues for Low Bit Rate VCQ traffic	368
23.5.3.2	Queues for High Bit Rate VCQ traffic	368
23.5.4	Overview of Operation	369
23.5.4.1	Processing for Low bit rate VCs	369
23.5.4.2	Processing for high bit rate VCs	369
23.5.5	Write-out block Pseudocode	370
23.5.6	Scheduler	371
23.5.6.1	Data Structures	371
23.5.7	Overview of Operation	373
23.5.7.1	Operations Not on a per Cell Transmission Slot Basis	373
23.5.8	Pseudocode	374
23.6	Flow Charts for Blocks	377
23.6.1	GCRA Flow Chart	378
23.6.2	F-GCRA Shaping Macro Flow Chart	379
23.6.3	Shaper (Macro only) Flow Chart	380
23.6.4	Write-out Block Flow Chart	381
23.6.5	Scheduler Flow Chart	382
23.6.6	Dequeue Flow Chart	383
23.7	Performance Analysis	384
23.8	2048 Ports Hierarchical Port-rate Shaping	384
23.8.1	Data structures	384
23.8.1.1	Port State	384
23.8.1.2	Local Memory Map	385
23.8.1.3	Static Port Schedule	386
23.8.1.4	Local Memory Time queues	387

23.8.2	Write-out Operation	389
23.8.3	Scheduler Operation	389
23.9	Up to 8 Ports Hierarchical Port-rate Shaping	389
23.9.1	Overview	389
23.9.2	Scheduler Data Structures	390
23.9.2.1	Port Information	390
23.9.2.2	Local Memory Map	391
23.9.3	Scheduler Operation	392
23.10	Performance Analysis	393

Forwarder

24	IPv4 Forwarder Microblock	397
24.1	Overview	397
24.2	Assumptions and Dependencies	398
24.3	Dependencies	399
24.4	Configuration Options	400
24.4.1	Build Switches	400
24.4.2	Default Configuration	401
24.5	RFC Compliance	402
24.6	Data Structures	403
24.6.1	Control Block	403
24.6.2	Directed Broadcast Table	404
24.6.3	Next Hop Information	405
24.6.3.1	Flags	406
24.6.3.2	Nexthop ID type	406
24.6.3.3	Nexthop ID	407
24.6.3.4	Fabric Port ID	407
24.6.3.5	Output Port ID	407
24.6.3.6	MTU	407
24.6.4	Nexthop Database	407
24.6.5	IPv4 Counters	408
24.6.6	Route Table	409
24.7	Longest Prefix Match Lookup	409
24.7.1	Introduction	409
24.7.2	Data Structures	409
24.7.3	Lookup	412
24.8	Intel® XScale™ Core Interface	414
24.8.1	Symbols	414
24.8.2	Exception Codes	414
24.9	High Level Flow Chart for Microblock	416
24.10	Performance Analysis	417
24.10.1	Characterization Data	418
25	IPv6 Forwarder Microblock	419
25.1	Overview	419
25.2	Assumptions	419
25.3	Dependencies	421
25.4	Configuration Options	421
25.4.1	Build Switches	421
25.4.2	Default Configuration	422

25.5	RFC Compliance	423
25.6	Data Structures	423
25.6.1	L3 Next Hop Information	423
25.6.1.1	Valid	424
25.6.1.2	Generic Flags	425
25.6.1.3	Blade ID	425
25.6.1.4	Next-Hop ID Type	425
25.6.1.5	Next-Hop ID	425
25.6.1.6	Output Port	426
25.6.1.7	MTU	426
25.6.2	Intermediate Next Hop Information	426
25.6.2.1	Status	427
25.6.2.2	Generic Flags	427
25.6.2.3	Next-Hop Index1-4	427
25.6.3	Next-Hop Database	427
25.6.4	IPv6 Counters	428
25.6.5	Route Table	429
25.7	Longest Prefix Match Lookup	429
25.7.1	Introduction	429
25.7.2	Data Structures	430
25.7.2.1	Creating the Route Table	431
25.7.2.2	Route Add Example	432
25.7.3	Route Lookup using Longest Prefix Match (LPM)	432
25.7.4	Lookup Pseudo Code	432
25.7.4.1	Optimized Route Lookup Algorithm	433
25.7.4.2	Trie-Cache Example	436
25.7.4.3	Controlled Prefix Expansion	437
25.8	Intel XScale® core Interface	438
25.8.1	Symbols	438
25.8.2	Exception Codes	439
25.9	High Level Microblock Flow Charts	439
25.10	Performance Analysis	442
25.10.1	Characterization Data	444
26	IPv6 To IPv4 Tunneling Microblock	447
26.1	Overview	447
26.2	Assumptions	447
26.3	Dependencies	449
26.4	Configuration Options	449
26.4.1	Build Switches	449
26.4.2	Default Configuration	450
26.5	RFC Compliance	451
26.5.1	Decapsulation	451
26.5.2	Encapsulation	452
26.6	Statistics Counters	453
26.7	V6V4-Tunnel-Decap Microblock	453
26.7.1	Introduction	453
26.7.1.1	Header Cache and Metadata Requirements	454
26.7.2	Data Structures	455
26.7.2.1	Tunnel Next-Hop information	455
26.7.2.2	Ingress Source List	456

26.7.2.3	Trie Table Option	458
26.7.3	Process Flow	458
26.7.3.1	Tunneling Header Checks	460
26.7.3.2	Automatic Tunneling Verification	461
26.7.3.3	6to4 Tunneling Verification	462
26.7.3.4	Source Address Validation Using the Ingress Source List	463
26.8	V6V4-Tunnel-Encap Microblock	464
26.8.1	Introduction	464
26.8.1.1	Header Cache and Metadata Requirements	464
26.8.2	Data Structures	465
26.8.2.1	Tunnel Next-Hop Information	465
26.8.3	Process Flow	466
26.8.3.1	Obtaining the Tunnel Endpoint Addresses	468
26.8.3.2	Creating the IPv4 Header	469
26.9	XScale Interface	470
26.9.1	Symbols	470
26.9.2	Exception Codes	471
26.10	Performance Analysis	471
27	IPv6 To IPv4 Translation Microblock	473
27.1	Overview	473
27.1.1	Traditional (Outbound) NAT-PT	473
27.1.1.1	Basic NAT-PT	473
27.1.1.2	NAPT-PT	474
27.1.2	Two-way (Bi-directional) NAT-PT	474
27.1.3	Application Level Gateway (ALG)	474
27.2	Assumptions	474
27.2.1	Dependencies	475
27.2.2	Configuration Options	477
27.2.2.1	Build Switches	477
27.2.3	Default Configuration	477
27.2.4	RFC Compliance	477
27.2.5	Statistics Counters	478
27.3	Overview of NAT-PT Microblock	478
27.4	Data Structures	479
27.4.1	Header Cache and Meta-Data Requirements	479
27.4.2	Translation Data Structures	480
27.4.2.1	NAT Table	480
27.4.2.2	NAPT Table	482
27.4.3	Process Flow	486
27.5	XScale Core Interface	492
27.5.1	Symbols	492
27.5.2	Exception Codes	493
27.5.3	Performance Analysis	494
27.5.4	Characterization Data	494

Layer 2

28	Layer-2 Decapsulation and Classify	503
28.1	PPP Decapsulation and Classify	503

28.2	Ethernet Decapsulation, Classify and Filter	503
28.3	LLCSNAP Decapsulation/Classify	505
28.4	Performance Analysis	506
28.4.1	Characterization Data	507
29	Layer-2 Encapsulation	513
29.1	PPP Encapsulation	513
29.2	Ethernet Encapsulation	513
29.3	LLCSNAP Encapsulation	513
29.4	Performance Analysis	514
29.4.1	Ethernet Encap	514
29.4.2	Ethernet Encap Characterization Data	515
29.4.3	PPP Encap	517
29.4.4	LLCSNAP Encap	520
29.4.4.1	LLCSNAP Encap Characterization Data	520

DiffServ

30	6-tuple Exact Match Classifier Microblock	525
30.1	Overview	525
30.2	Assumptions and Dependencies	526
30.2.1	Configuration Options	527
30.2.1.1	Build Switches	527
30.2.1.2	Default Configuration	527
30.3	Microblock Design	527
30.3.1	Functionality	527
30.3.2	Microblock Interfaces	528
30.3.2.1	Input Microblock Variables	528
30.3.2.2	Output Microblock Variables	528
30.3.2.3	Imported Variables	529
30.3.3	Data Structures	530
30.3.4	Flow Chart	532
30.3.4.1	Synchronization	532
30.3.4.2	Classification algorithm	532
30.3.4.3	Statistics Gathering	536
30.4	Microcode Budget	537
30.4.1	Performance Analysis	537
30.4.2	Memory Footprint Analysis	538
31	Three Color Meter Microblock	539
31.1	Overview	539
31.2	Functionality	541
31.3	Assumptions and Dependencies	541
31.3.1	Assumptions	541
31.3.2	Configuration Options	542
31.3.2.1	Build Switches	542
31.3.2.2	Default Configuration	542
31.4	Microblock Interfaces	543
31.4.1	Input Microblock Variables	543
31.4.2	Output Microblock Variables	543
31.4.3	Imported Variables	543

31.5	Data Structures	544
31.6	Flow Chart	547
31.6.1	Synchronization	547
31.6.2	SRTCM Algorithm	549
31.6.2.1	Per-packet Token Updates	550
31.6.2.2	Color-aware Metering	551
31.6.2.3	Statistics Gathering	552
31.6.3	TRTCM Algorithm	553
31.6.3.1	Per-packet Token Updates	553
31.6.3.2	Color-aware Metering	555
31.6.3.3	Statistics Gathering	556
31.7	Micro-code Budget	557
31.7.1	Performance Analysis	557
31.7.2	Memory Footprint Analysis	558
32	DSCP Marker Microblock	561
32.1	Overview	561
32.2	Microblock Interfaces	561
32.2.1	Input Microblock Variables	561
32.2.2	Output Microblock Variables	561
32.3	Flow Chart	562
32.3.1	Synchronization	562
32.3.2	Marking Algorithm	562
32.4	Micro-code Budget	563
32.4.1	Performance Analysis	563
32.4.2	Memory Footprint Analysis	564
33	DSCP Classifier Microblock	565
33.1	Overview	565
33.2	Microblock Design	565
33.2.1	Functionality	565
33.2.2	Assumptions and Dependencies	566
33.2.3	Configuration Options	566
33.2.3.1	Build Switches	566
33.2.3.2	Default Configuration	566
33.2.4	Microblock Interfaces	567
33.2.4.1	Input Microblock Variables	567
33.2.4.2	Output Microblock Variables	567
33.2.4.3	Imported Variables	568
33.2.5	Data Structures	568
33.2.6	Flow Chart	570
33.2.6.1	Synchronization	570
33.2.6.2	Classification Algorithm	570
33.2.6.3	Statistics Gathering	573
33.2.7	Micro-code Budget	574
33.2.7.1	Performance Analysis	574
33.2.7.2	Memory Footprint Analysis	575
34	Weighted Random Early Detection (WRED) Microblock	577
34.1	Overview	577
34.1.1	RED 935	577

34.1.1.1	Average queue length	578
34.1.1.2	Packet Drop Probability	578
34.1.2	RED	579
34.2	Functionality	579
34.3	Assumptions and Dependencies	580
34.3.1	Configuration Options	580
34.3.1.1	Build Switches	580
34.3.1.2	Default Configuration	581
34.4	Microblock Interfaces	581
34.4.1	Input Microblock Variables	581
34.4.2	Output Microblock Variables	581
34.4.3	Imported Variables	582
34.5	Data Structures	582
34.6	Flow Chart	587
34.6.1	Synchronization	587
34.6.2	WRED Algorithm	589
34.6.2.1	WRED for low-speed queues	589
34.6.2.2	WRED for high-speed queues	595
34.7	Micro-code Budget	596
34.7.1	Micro-code Budget for WRED Low-speed Queues	596
34.7.1.1	Performance Analysis	596
34.7.1.2	Memory Footprint Analysis	597
34.7.2	Micro-code Budget for WRED High-speed Queues	597
34.7.2.1	Performance Analysis	597
34.7.2.2	Memory Footprint Analysis	598

MPLS Microblocks

35	FTN Forwarder Microblock	601
35.1	Overview	601
35.2	Functionality	601
35.3	Assumptions and Dependencies	602
35.3.1	Assumptions	602
35.3.2	Component Interfacing	602
35.4	Microblock Interfaces	603
35.4.1	Input Microblock Variables	603
35.4.2	Output Microblock Variables	604
35.5	Data Structures and Forwarding Algorithm	605
35.5.1	Overview	605
35.5.2	NHLFE Table	606
35.5.3	NHLFE Set Table	607
35.5.4	NHLFE Counters Table	608
35.5.5	Forwarding Algorithm	608
35.5.6	Initial Label Stack Building	611
35.5.7	Allocation to Microengines	611
35.5.8	Thread Ordering and Synchronization	612
35.5.9	FTN Forwarder Micro-code Budget	614
35.5.9.1	Performance Analysis	614
35.5.9.2	Memory Footprint Analysis	615
35.5.10	FTN Forwarder Characterization Data	615

36	ILM Forwarder Microblock	619
36.1	Overview	619
36.2	Functionality	619
36.3	Assumptions and Dependencies	620
36.3.1	Assumptions	620
36.3.2	Component Dependencies	621
36.4	Microblock Interfaces	621
36.4.1	Input Microblock Variables	621
36.4.2	Output Microblock Variables	622
36.5	Data Structures and Forwarding Algorithm	622
36.5.1	Overview	622
36.5.2	ILM Table	624
36.5.3	ILM_NHLFE Set	625
36.5.4	Input Segment Counters	626
36.5.5	InPort Counters	627
36.5.6	Forwarding Algorithm	627
36.5.7	Allocation to Microengines	633
36.5.8	Thread Ordering and Synchronization	633
36.5.9	ILM Forwarder Micro-code Budget	634
36.5.9.1	Performance Analysis	634
36.5.9.2	Memory Footprint Analysis	635
36.5.10	ILM Forwarder Microblock Characterization Data	636

Services Microblocks

37	Packet Copier Microblock	641
37.1	Overview	641
37.2	Requirements	642
37.3	Data Flow	643
37.4	External Interfaces	643
37.4.1	Packet Copier Request Message	644
37.4.2	Packet Copier Response Message	644
37.5	Internal Data Structures	644
37.6	Packet Replication Algorithm	645
37.7	Startup/Shutdown	646
37.8	Performance Analysis	646
38	Freelist Manager	651
38.1	Overview	651
38.2	Assumptions and Dependencies	651
38.3	Algorithm	651
38.3.1	Overview	651
38.3.2	Three-Level Hierarchical Implementation	652
38.3.3	Summary	652
38.4	Data Structures	652
38.5	Performance Analysis	652

Core Components 657

39	Core Components Overview	659
39.1	Overview	659
39.1.1	Functional and Data Flow	659
39.1.2	Functional APIs Design Concept	661
39.2	APIs for Dynamic Property Updates	661
39.2.1	Dynamic Properties and Clients	662
39.2.2	Property Updates API and Data Structures	662
39.2.2.1	Properties Data Structure	662
39.2.3	Property ID	663
39.2.4	Property API Generic Prototype	663
39.2.5	Current Behavior of Property API	663
39.3	Handler Registration	664
39.3.1	Support for the IXA Portability Framework and Core Components Infrastructure	664
39.3.1.1	Usage of init() and fini() Functions	664
39.3.2	Operating System Independence of Core Components	665
39.4	High-Level Overview of the Core Components	666
39.4.1	Applications	666
39.4.1.1	IPv4 Application	667
39.4.1.2	DiffServ Application	668
39.4.1.3	MPLS Application	669
39.4.1.4	IPv6 Application	670
39.4.2	Building Block Core Components	672
39.4.2.1	POS RX	672
39.4.2.2	IPv4 Forwarder	672
39.4.2.3	Queue Manager (QM)	672
39.4.2.4	Scheduler	673
39.4.2.5	CSIX TX	673
39.4.2.6	CSIX RX	674
39.4.2.7	ATM/POS TX	674
39.4.2.8	Ethernet RX	674
39.4.2.9	Ethernet TX	675
39.4.2.10	L2 Table Manager	675
39.4.2.11	Route Table Manager	675
39.4.2.12	Six-Tuple Classifier	676
39.4.2.13	Single Rate Three Color Meter	676
39.4.2.14	Weighted Random Early Detection (WRED)	677
39.4.2.15	Queue Manager for DiffServ	677
39.4.2.16	Scheduler for DiffServ	677
39.4.2.17	IPv6 Forwarder	677
39.4.2.18	IPv6 to IPv4 Tunneling	678
39.4.2.19	Route Table Manager for IPv6	678
39.4.2.20	Stack Driver	679
39.4.2.21	System Application	679
39.4.2.22	Message Helper and Support Library	679
39.4.2.23	SoftSAR Core Components	680
39.4.3	MPLS Forwarder Core Component	681
40	System Application	683
40.1	Overview	683
40.2	Assumptions	683
40.3	Design	684

- 40.3.1 Initialization Sequence684
- 40.3.2 Shutdown and Re-initialization685
 - 40.3.2.1 Re-initialization686
 - 40.3.2.2 Start and Shut Down API686
- 40.3.3 Load Microcode and Start Microengines686
 - 40.3.3.1 Loading Microcode and Starting Microengines API687
- 40.3.4 User Initialization/Shutdown Hooks687
 - 40.3.4.1 Initialization/Shutdown Hooks687
- 40.3.5 Core Component Infrastructure Initialization688
- 40.3.6 Execution Engines688
- 40.3.7 Core Components689
- 40.3.8 Scratch Rings690
- 40.3.9 Property Master691
 - 40.3.9.1 Mastered Properties691
- 40.3.10 Support Facilities691

Receive Components

41 POS RX Core Component695

- 41.1 Overview695
- 41.2 Assumptions and Dependencies695
 - 41.2.1 Assumptions695
 - 41.2.2 Dependencies695
- 41.3 Data flow696
- 41.4 Configuration and Initialization696
 - 41.4.1 Static Configuration Data696
 - 41.4.2 Patching Symbols and Memory Allocation697
- 41.5 External API697
 - 41.5.1 Core Component Infrastructure API697
 - 41.5.2 Messaging API697
 - 41.5.3 Library API698

42 CSIX RX Core Component699

- 42.1 Data flow699
- 42.2 Assumptions and Dependencies700
 - 42.2.1 Assumptions700
 - 42.2.2 Dependencies700
- 42.3 Configuration and Initialization701
- 42.4 External API701
 - 42.4.1 Core Component Infrastructure API701
 - 42.4.2 Messaging API701
 - 42.4.3 Library API702

43 Ethernet RX Core Component703

- 43.1 Overview703
- 43.2 Assumptions and Dependencies703
 - 43.2.1 Assumptions703
 - 43.2.2 Dependencies703
- 43.3 Data flow704
 - 43.3.1 Data Input and Output704
 - 43.3.2 Packet Data Flow705

- 43.4 Configuration and Initialization706
 - 43.4.1 Dynamic Configuration Data706
 - 43.4.2 Static Configuration Data706
 - 43.4.3 Patching Symbols707
- 43.5 Modularity707
- 43.6 External API708
 - 43.6.1 Core Component Infrastructure API708
 - 43.6.2 Messaging API708
 - 43.6.3 Library API709

Transmit Components

- 44 CSIX TX Core Component713**
 - 44.1 Overview713
 - 44.2 Data flow713
 - 44.3 Assumptions and Dependencies714
 - 44.3.1 Assumptions714
 - 44.3.2 Dependencies714
 - 44.4 Configuration and Initialization714
 - 44.5 External API715
 - 44.5.1 Core Component Infrastructure API715
 - 44.5.2 Messaging API715
 - 44.5.3 Library API716
- 45 ATM/POS TX Core Component717**
 - 45.1 Overview717
 - 45.2 Data flow717
 - 45.3 Assumptions and Dependencies718
 - 45.3.1 Assumptions718
 - 45.3.2 Dependencies718
 - 45.4 Configuration and Initialization719
 - 45.4.1 Dynamic Configuration Data719
 - 45.4.2 Static Configuration Data720
 - 45.4.3 ATM/POS Media Card Operation Mode720
 - 45.4.4 Patching Symbols722
 - 45.5 External API724
 - 45.5.1 Core Component Infrastructure API724
 - 45.5.2 Messaging API724
 - 45.5.2.1 Library API724
- 46 Ethernet TX Core Component725**
 - 46.1 Overview725
 - 46.2 Assumptions and Dependencies725
 - 46.2.1 Assumptions725
 - 46.2.2 Component Dependencies726
 - 46.3 Data flow726
 - 46.3.1 Packet Input and Output726
 - 46.3.2 Packet Data Flow727
 - 46.4 Configuration and Initialization728
 - 46.4.1 Dynamic Configuration Data728
 - 46.4.2 Static Configuration Data729

- 46.4.3 Table Creations729
 - 46.4.3.1 Local Interface Table729
 - 46.4.3.2 L2 Table729
 - 46.4.3.3 ARP Cache729
- 46.4.4 Patching Symbols730
- 46.4.5 Modularity731
- 46.5 External API732
 - 46.5.1 Data Structures732
 - 46.5.2 Core Component Infrastructure API732
 - 46.5.3 Messaging API733
 - 46.5.4 Library API733

47 Ethernet ARP Module735

- 47.1 Assumptions and Dependencies736
 - 47.1.1 Assumptions736
 - 47.1.2 Dependencies736
- 47.2 Data flow737
 - 47.2.1 ARP Generation Data Flow737
 - 47.2.2 ARP Reception Data Flow738
- 47.3 External API739
 - 47.3.1 Error Codes739
- 47.4 Modularity740
 - 47.4.1 External API741

Queue Manager Components

48 Queue Manager Core Component745

- 48.1 Overview745
- 48.2 Configuration and Initialization745
 - 48.2.1 Configuration745
 - 48.2.2 Configuration Items746
 - 48.2.3 Initialization746
- 48.3 Data Flow747
- 48.4 External API748
 - 48.4.1 Core Component Infrastructure API748
 - 48.4.2 Functional API748
 - 48.4.2.1 Messaging API748
 - 48.4.2.2 Library API748

49 Queue Manager (DiffServ) Core Component749

- 49.1 Overview749

Scheduler Components

50 Scheduler Core Component753

- 50.1 Overview753
- 50.2 Configuration and Initialization754
 - 50.2.1 Configuration754
 - 50.2.2 CSIX Scheduler755
 - 50.2.3 Packet Scheduler755
 - 50.2.4 Configuration Items755

50.2.5	Initialization	756
50.3	Core Component Infrastructure API	757
51	Scheduler (DiffServ) Core Component	759
51.1	Overview	759
Forwarder Components		
52	IPv4 Forwarder Core Component	763
52.1	Overview	763
52.2	Assumptions and Dependencies	763
52.2.1	Assumptions	763
52.2.2	Dependencies	764
52.3	Configuration and Initialization	764
52.4	IPv4 Forwarder Core Component Modules	766
52.4.1	ICMP	767
52.4.1.1	Supported ICMP Error Messages	767
52.4.1.2	Rate Limiting	768
52.4.2	Forwarding and IP Header Validation Module	769
52.4.2.1	Forwarding	769
52.4.2.2	IP Header Validation	769
52.4.2.3	Identifying Packets for Local Delivery	771
52.4.3	IP Options Handling	771
52.4.4	Fragmentation support	772
52.4.5	Packet Handling Module	773
52.4.6	Message Handling Module	773
52.5	External API	773
52.5.1	Data Structures, Types and Macros	773
52.5.2	Core Component Infrastructure API	774
52.5.3	Messaging API	774
52.5.4	Library API	775
53	IPv6 Forwarder Core Component	777
53.1	Data Flow	777
53.2	Assumptions and Dependencies	777
53.2.1	Assumptions	777
53.2.2	Component Dependencies	778
53.3	Configuration and Initialization	778
53.4	Modularity	780
53.4.1	Forwarding and IP Header validation Module	781
53.4.1.1	Forwarding	781
53.4.1.2	IP Header Validation	782
53.4.1.3	Identifying packets for local delivery	783
53.4.2	Packet Handling Module	784
53.4.3	Message Handling Module	784
53.4.4	ICMPv6	784
53.4.4.1	RFC2463 MUST Features for ICMPv6 Error Message Processing	785
53.4.5	Neighbor Discovery	785
53.4.5.1	Supported Neighbor Discovery Messages	786
53.4.5.2	Address Resolution	786

	53.4.6	Address Autoconfiguration	786
53.5		External API	787
	53.5.1	Data Structures	787
	53.5.2	Core Component Infrastructure API	787
	53.5.3	Message Helper API	788
	53.5.4	Library API	789
54		IPv6 To IPv4 Tunneling	
		Core Component	791
	54.1	Overview	791
	54.2	Data Flow	791
	54.3	Assumptions and Dependencies	792
	54.3.1	Dependencies	792
	54.4	Configuration and Initialization	792
	54.4.1	Configuration Parameters	792
	54.4.1.1	Size of End Tunnel Next Hop Table	792
	54.4.1.2	Size of Start Tunnel Next Hop Table	792
	54.4.1.3	Format of Ingress Source Validation List	792
	54.4.1.4	Size Hint for Ingress Source Validation List	793
	54.4.2	Initialization	793
54.5		Modularity	793
	54.5.1	Initialization Module	794
	54.5.2	Message Handling Module	794
	54.5.3	Packet Handling Module	794
	54.5.3.1	Microblock Packet Handler	795
	54.5.3.2	IPv4 Packet Handler	795
	54.5.3.3	IPv6 Packet Handler	795
	54.5.4	Decapsulation Module	796
	54.5.5	Encapsulation Module	796
	54.5.6	Reassembly Support Module	796
	54.5.7	ICMP Error Message Support Module	796
	54.5.7.1	Packet Too Big Messages	796
	54.5.7.2	Other Error Messages	796
	54.5.8	Tunneling Header Validation Support Module	796
	54.5.8.1	RFC 2893 IPv4 Source Address Checks	797
	54.5.8.2	RFC 2893 IPv6 Source Address Checks	797
	54.5.8.3	RFC 2893 Automatic Tunnel Embedded Address Checks	797
	54.5.8.4	RFC 3056 6to4 Embedded Address Checks	797
	54.5.9	Configuration Support Module	797
54.6		Bindings	797
54.7		External API	798
	54.7.1	Data Structures, Types and Macros	798
	54.7.2	Core Component Infrastructure API	799
	54.7.3	Message Helper API	799
54.8		Library API	801
55		NAT-PT Translation Core Components	803
	55.1	Dependencies	803
	55.2	High-Level Architecture	804
	55.2.1	Main Module	804

55.2.2	DNS Application Level Gateway	805
55.2.2.1	DNS Name to Address Query from IPv4 Realm	805
55.2.2.2	DNS Name to Address Query from IPv6 Realm	805
55.2.3	FTP Application Level Gateway	805
55.2.3.1	IPv4 FTP Client communicates with IPv6 FTP Server	805
55.2.3.2	IPv6 FTP Client communicates with IPv4 FTP Server	805
55.2.3.3	Packet Header update	806
55.2.4	Specific Design Details	806
55.2.5	NAT-PT Operation	806
55.2.6	NAPT-PT Operation	806
55.2.7	Static Mapping of Addresses	806
55.2.8	Static Port Mapping	807
55.3	Configuration and Initialization	807
55.3.1	Configuration Parameters	807
55.3.1.1	NAT-PT Table Size	807
55.3.1.2	NAT-PT Hash Table Size	807
55.3.1.3	NAT-PT Hash Collision Table Size	807
55.3.1.4	NAT-PT Prefix for IPv6 domain	807
55.3.2	Initialization	808
55.3.2.1	NAT-PT Table Base	808
55.3.2.2	NAT-PT V6V4 Hash Table Base	808
55.3.2.3	NAT-PT V4V6 Hash Table Base	808
55.3.2.4	NAT-PT Hash Collision Table Base	808
55.3.2.5	Blade ID	808
55.3.2.6	48-bit Hash Multiplier	808
55.3.2.7	128-bit Hash Multiplier	808
55.3.2.8	NAT-PT Prefix for IPv6 domain	808
55.4	External API	809
55.4.1	Data Structures, Types and Macros	809
55.4.2	Core Component Infrastructure API	809
55.4.3	Messaging API	810
55.4.4	Library API	811
55.5	Modularity	811
55.5.1	Initialization Module	812
55.5.2	Message Handling Module	812
55.5.3	Packet Handling Module	812
55.5.3.1	Microblock Packet Handler	812
55.5.4	Fragmentation & Reassembly Support Module	813
55.5.5	ICMP Translation Module	813
55.5.6	Translation Module	813
55.5.7	DNS ALG Module	813
55.5.8	FTP ALG Module	813
55.5.9	Packet Sending Support Module	813
55.5.10	Configuration Support Module	814
55.6	Bindings	814

DiffServ Components

56 Six-Tuple Classifier Core Component

56.1	Overview	817
56.2	Data and Control Flow	818

56.2.1	Packet Inputs	818
56.2.2	Packet Outputs	818
56.2.3	Message Inputs	819
56.3	Assumptions, Dependencies and Risks	819
56.3.1	Assumptions	819
56.3.2	Dependencies	819
56.4	Configuration and Initialization	820
56.4.1	Static Configuration Data	820
56.4.2	Dynamic Configuration Data	822
56.4.3	Patching Symbols	822
56.4.4	Initialization and Shutdown Data Flow	822
56.5	External API	823
56.5.1	Data Structures	823
56.5.2	Core Component Infrastructure API	824
56.5.3	Message Helper API	824
56.5.4	Library API	824
56.6	Modularity	825
57	Three Color Meter Core Component	827
57.1	Overview	827
57.2	Assumptions, Dependencies and Risks	827
57.2.1	Assumptions	827
57.2.2	Dependencies	827
57.3	Configuration and Initialization	828
57.3.1	Static Configuration Data	828
57.3.2	Dynamic Configuration Data	829
57.3.3	Patching Symbols	829
57.3.4	Initialization and Shutdown Data Flow	829
57.4	Data and Control Flow	831
57.4.1	Packet Inputs	831
57.4.2	Packet Outputs	831
57.4.3	Message Inputs	831
57.5	External API	832
57.5.1	Data Structures	832
57.5.2	Core Component Infrastructure API	832
57.5.3	Message Helper API	832
57.5.4	Library API	833
57.6	Modularity	833
58	Weighted Random Early Detection (WRED) Core Component	835
58.1	Overview	835
58.2	Assumptions, Dependencies and Risks	835
58.2.1	Assumptions	835
58.2.2	Dependencies	835
58.3	Configuration and Initialization	836
58.3.1	Static Configuration Data	836
58.3.2	Dynamic Configuration Data	837
58.3.3	Patching Symbols	837
58.3.4	Initialization and Shutdown Data Flow	837
58.4	Data and Control Flow	839

- 58.4.1 Packet Inputs839
- 58.4.2 Packet Outputs839
- 58.4.3 Message Inputs839
- 58.5 External API840
 - 58.5.1 Data Structures840
 - 58.5.2 Core Component Infrastructure API840
 - 58.5.3 Message Helper API840
 - 58.5.4 Library API841
- 58.6 Modularity841
- 59 DSCP Classifier Core Component843**
 - 59.1 Overview843
 - 59.1.1 Data and Control Flow843
 - 59.1.1.1 Packet Inputs844
 - 59.1.1.2 Packet Outputs844
 - 59.1.1.3 Message Inputs844
 - 59.2 Assumptions, Dependencies and Risks845
 - 59.2.1 Assumptions845
 - 59.2.2 Dependencies845
 - 59.3 Configuration and Initialization846
 - 59.3.1 Static Configuration Data846
 - 59.3.2 Dynamic Configuration Data847
 - 59.3.2.1 Patching Symbols847
 - 59.3.3 Initialization and Shutdown Data Flow847
 - 59.4 External API848
 - 59.4.1 Data Structures848
 - 59.4.2 Core Component Infrastructure API849
 - 59.4.3 Message Helper API849
 - 59.4.4 Library API849
 - 59.5 Modularity850

Support Libraries

- 60 Route Table Manager855**
 - 60.1 Overview855
 - 60.2 Usage Model855
 - 60.2.1 Using Routes and Next Hops855
 - 60.2.2 Initialization857
 - 60.2.3 Pre-assigned Next Hop Identifiers857
 - 60.2.4 Default Route857
 - 60.3 Design Criteria858
 - 60.3.1 Data Storage and Retrieval858
 - 60.3.2 Guaranteed Table Validity858
 - 60.3.3 Multiple Table Support858
 - 60.3.4 Control Plane Design859
 - 60.3.5 Microengine Timing859
 - 60.3.6 High-level Route Table859
 - 60.3.7 Lookup Library Initialization859
 - 60.3.8 Single Direct Client860
 - 60.3.9 Duplicate Routes860
 - 60.4 Modularity860

60.5	External API	861
60.5.1	Data Structures and Types	862
60.5.2	Macros	862
60.5.3	Core Component Infrastructure API	862
61	Route Table Manager for IPV6	
	Core Component	865
61.1	Overview	865
61.2	Usage Model	865
61.2.1	Using Routes and Next Hops	865
61.2.2	Initialization	867
61.2.3	Pre-assigned Next Hop Identifiers	867
61.2.4	Default Route	867
61.2.5	Support for equal cost next hops	867
61.3	Design Criteria	867
61.3.1	Data Storage and Retrieval	868
61.3.2	Guaranteed Table Validity	868
61.3.3	Multiple Table Support	868
61.3.4	Control Plane Design	868
61.3.5	High-level Route Table	868
61.3.6	Lookup Library Initialization	868
61.3.7	Single Direct Client	869
61.3.8	Duplicate Routes	869
61.4	External API	869
61.4.1	Data Structures, Types, and Macros	869
61.5	Core Component Infrastructure API	870
62	L2 Table Manager	871
62.1	Design Considerations	871
62.2	L2 Table Entries	871
62.3	Data Flow and Behavior	873
62.4	Multi-Client Support	874
62.5	Microblock Synchronization	874
62.6	Initialization and Usage	874
62.7	Modularity	875
62.8	External API	876
62.8.1	Data Structures, Types, and Macros	876
62.8.2	Library Functions	876
63	Message Helper and Support Library	879
63.1	Overview	879
63.2	Assumptions	879
63.3	Overview	879
63.3.1	Message Helper Library	881
63.3.2	Support Library	882
63.4	Usage	882
63.4.1	Client Usage	882
63.4.1.1	Example	883
63.4.2	Core Component Usage	884
63.5	Message Support Library Internal Design	885

- 63.5.1 Core Component Infrastructure885
 - 63.5.1.1 Initialization885
 - 63.5.1.2 Shutdown885
 - 63.5.1.3 Call-Table886
- 63.6 Data Flow888
 - 63.6.1 Asynchronous Call: Data Flow888
 - 63.6.2 Synchronous Call: Data Flow890
- 63.7 Resource Manager Only System891
 - 63.7.1 Initialization891
 - 63.7.2 Internal Operation891
 - 63.7.3 Asynchronous and Fire and Forget892
 - 63.7.4 Synchronous892
- 63.8 Message Support Library API892

Stack Driver Component

64 Stack Driver895

- 64.1 Overview895
- 64.2 Assumptions and Dependencies895
 - 64.2.1 Assumptions895
 - 64.2.2 Dependencies895
- 64.3 Stack Driver896
 - 64.3.1 Stack Driver Design896
 - 64.3.1.1 Stack Driver Design897
 - 64.3.1.2 Packet Flow898
 - 64.3.1.3 Synchronizing Properties898
 - 64.3.2 Design of the Core Component Module899
 - 64.3.2.1 Core Component Module Design899
 - 64.3.2.2 Execution Context900
 - 64.3.3 External API900
 - 64.3.3.1 Core Component Module Data Structures and Types900
 - 64.3.3.2 Core Component Infrastructure API901
 - 64.3.3.3 Core Component Infrastructure Separation901
 - 64.3.3.4 Initialization902
 - 64.3.3.5 Shutdown902
 - 64.3.3.6 Packet and Message Processing API903
 - 64.3.3.7 Properties API903
 - 64.3.4 Packet Classifier Design903
 - 64.3.4.1 Theory of Operation903
 - 64.3.4.2 Packet Classifier Design904
 - 64.3.4.3 Packet Classifier Data Structures904
 - 64.3.4.4 Packet Classifier External API905
 - 64.3.4.5 Packet Classifier Data Flow905
 - 64.3.5 Outgoing Packet Classifier Design906
 - 64.3.5.1 Outgoing Packet Classifier Data Structures906
 - 64.3.5.2 Outgoing Packet Classifier Internal API906
 - 64.3.5.3 Outgoing Packet Classifier Data Flow907
 - 64.3.5.4 Outgoing Packet Classifier Design Scalability907
 - 64.3.6 VIDD for VxWorks908
 - 64.3.6.1 VIDD System Data Structures909
 - 64.3.6.2 VIDD Local Data Structures909
 - 64.3.7 MUX Interface API909

- 64.3.7.1 VIDD System Function Calls910
- 64.3.7.2 MUX APIs used by the VIDD911
- 64.3.8 Packet Processing—VxWorks Example911
 - 64.3.8.1 Core Component Module Side911
 - 64.3.8.2 VIDD Side912
 - 64.3.8.3 Outgoing Packet Processing Pseudocode912
 - 64.3.8.4 Pseudocode for Outgoing Packets—in the Core Component Module913
- 64.3.9 VIDD for Linux*913
 - 64.3.9.1 VIDD System Data Structures for Linux913
 - 64.3.9.2 VIDD System API for Linux913
 - 64.3.9.3 VIDD Linux Driver Support API914
- 64.3.10 Transport Module Design914
 - 64.3.10.1 Transport Module Data Structures914
 - 64.3.10.2 Transport Module External API914
- 64.3.11 Start-up Configuration File Requirements916

SoftSAR Components

65 SoftSAR Core Components919

- 65.1 Architecture Overview919
 - 65.1.1 Hardware Architecture Overview919
 - 65.1.2 General Software Architecture Overview920
 - 65.1.3 ATM Building Blocks921
 - 65.1.3.1 Microblocks Overview921
 - 65.1.3.2 ATM Core Components Overview922
- 65.2 Functionality923
 - 65.2.1 Handle and Handle Manager Concept923
- 65.3 ATM Objects Control923
- 65.4 Plug-in Core Components924
 - 65.4.1 Plug-in Registering924
- 65.5 Support for Single/Dual IXP Hardware Configuration925
- 65.6 Multi-Instances Support926
- 65.7 SoftSAR Core Components927
 - 65.7.1 Dynamic Behavior928
 - 65.7.2 SAR Control Main Core Component930
 - 65.7.2.1 Functionality930
 - 65.7.2.2 Assumptions and Dependencies930
 - 65.7.2.3 Decomposition931
 - 65.7.2.4 Data and Control Flow931
 - 65.7.2.5 Configuration and Initialization933
 - 65.7.3 External API933
 - 65.7.3.1 Data Structures933
 - 65.7.3.2 Core Component Infrastructure API934
 - 65.7.3.3 Messaging API934
 - 65.7.3.4 Library API935
 - 65.7.3.5 Plug-in API935
- 65.8 SAR Control Agent Core Component Design935
 - 65.8.1 Assumptions and Dependencies936
 - 65.8.1.1 Assumptions936
 - 65.8.1.2 Dependencies936
 - 65.8.2 Decomposition936

- 65.8.3 Data and Control Flow937
- 65.8.4 Configuration and Initialization937
- 65.9 ATM RX Core Component938
 - 65.9.1 Assumptions and Dependencies938
 - 65.9.1.1 Assumptions938
 - 65.9.1.2 Dependencies938
 - 65.9.2 Shared Data Structures939
 - 65.9.3 Data flow939
 - 65.9.3.1 Data Input and Output939
 - 65.9.3.2 Synchronization939
 - 65.9.4 Configuration and Initialization940
 - 65.9.4.1 Static Configuration Data940
 - 65.9.4.2 Hash Mechanism for VC Searching941
 - 65.9.5 Startup/Shutdown941
 - 65.9.6 External API941
 - 65.9.6.1 Core Component Infrastructure API942
- 65.10 ATM TX Core Component942
 - 65.10.1 Data flow942
 - 65.10.2 Assumptions and Dependencies942
 - 65.10.3 Assumptions942
 - 65.10.4 Dependencies943
 - 65.10.5 Configuration and Initialization943
 - 65.10.5.1 Dynamic Configuration Data944
 - 65.10.6 Static Configuration Data944
 - 65.10.7 Startup/Shutdown944
 - 65.10.8 External API944
 - 65.10.8.1 Core Component Infrastructure API944
- 65.11 TM4.1 Core Component945
 - 65.11.1 Assumptions and Dependencies945
 - 65.11.2 Shared Data Structures945
 - 65.11.3 Data and Control Flow946
 - 65.11.3.1 Support for Port Shaping Table946
 - 65.11.3.2 Support for HBR VC948
 - 65.11.4 Configuration and Initialization948
 - 65.11.5 Startup/Shutdown Discussion948
 - 65.11.6 External API949
 - 65.11.6.1 Core Component Infrastructure API949

MPLS Components

- 66 MPLS Forwarder Core Component953**
 - 66.1 Overview953
 - 66.2 Data Flow953
 - 66.3 Assumptions and Dependencies954
 - 66.3.1 Assumptions954
 - 66.3.2 Component Dependencies955
 - 66.3.3 IPv4 and MPLS Core Component Cooperation955
 - 66.3.4 Configuration and Initialization956
 - 66.3.4.1 Static Configuration Data956
 - 66.3.4.2 Dynamic Configuration Data956
 - 66.3.4.3 Patching Symbols956

- 66.4 Modularity958
- 66.5 External API959
 - 66.5.1 Data Structures and Types960
 - 66.5.2 Core Component Infrastructure API960
 - 66.5.3 Message Helper API961
 - 66.5.4 Library API961

A Network Simulator963

- A.1 Overview963
- A.2 Architecture963
 - A.2.1 Connecting NetSim to the IXP Workbench965
 - A.2.2 Assigning NetSim to an IXP Simulation Port965
- A.3 Initializing NetSim966
- A.4 NetSim Workbench Script Reference967
 - A.4.1 void netsim_verbose(int errorlevel)967
 - A.4.2 void netsim_load_tcs(char* tcs_filename)967
 - A.4.3 void netsim_start(void)967
- A.5 XML Basics967
 - A.5.1 XML Documents968
 - A.5.2 Node Element968
 - A.5.3 Node Attributes968
 - A.5.4 Empty Node969
 - A.5.5 Document Type Definition (*.DTD)969
 - A.5.5.1. Defining Elements in the DTD969
 - A.5.5.2. Defining Attributes in the DTD971
 - A.5.5.3. Importing DTDs from an External File972
 - A.5.5.4. Importing DTDs from Multiple Files973
 - A.5.5.5. Defining Constant Values Using ENTITY Definitions973
- A.6 NetSim TCS and DTD Configuration974
 - A.6.1 TCS File Structure for NetSim974
 - A.6.1.1. Sections of TCS File974
 - A.6.2 Creating and Assigning Streams to a Device/Port975
 - A.6.2.1. The Device and Port Attributes975
- A.7 Miscellaneous Features978
 - A.7.1 Inducing Frame/Cell Errors978
 - A.7.2 File Logging979
- A.8 Traffic Capture with Validation982
 - A.8.1 TCS Configuration for Capture with Validation982

B AAL-2 Receive Microblock985

- B.1 Overview985
 - B.1.1 AAL-2 CPS986
 - B.1.2 AAL-2 SSSCS986
 - B.1.3 Design Overview986
 - B.1.3.1. Packet Sequencing through Thread Synchronization986
 - B.1.3.2. Buffer Freelist986
 - B.1.3.3. CPS and SSSAR Processing987
- B.2 Assumptions987
- B.3 Configuration Options987
- B.4 Data Structures988
 - B.4.1 AAL-2 CPS Receive Context (CPS RXC) Table988

	B.4.2	CID Receive Context (CID RXC) Table	988
	B.4.3	Hash Tables	988
	B.4.4	CRC-5 Tables	990
	B.4.5	Counters	990
	B.4.6	Local Memory Queue	990
B.5		Algorithm	991
	B.5.1	Cell RX	991
	B.5.2	Thread Locking and Cell Queuing	991
	B.5.3	CPS/SSSAR Processing	993
	B.5.4	Performance Analysis	997
C		AAL-2 Transmit Microblock	999
	C.1	Overview	999
		C.1.1	Design Overview1001
			C.1.1.1. Packet Sequencing through Thread Synchronization1001
			C.1.1.2. SSSAR/CPS SDU Processing1004
			C.1.1.3. SSSAR Sublayer Functions1005
			C.1.1.4. CPS Sublayer Functions1006
			C.1.1.5. Timer CU Handling1010
	C.2	Configuration/Switches	1013
		C.2.1	Assumptions, Dependencies and Risks1013
	C.3	Data Structures	1013
		C.3.1	AAL2 Transmit VC Context1014
		C.3.2	CRC-5 Tables1015
		C.3.3	Timer CU Data Structure1015
		C.3.4	Counters1016
	C.4	Performance Analysis	1016
D		Glossary	1017

2-1	Format of the Buffer Handle	57
2-2	Buffer Handle Format for Q-Array in Packet Mode	57
2-3	Packet Metadata Format	58
2-4	Build Switches that May be Applied to all Projects	65
3-1	Child Meta Data in Cell Mode	71
3-2	Parent Meta Data in Cell Mode when Packet is Replicated	72
3-3	Child Meta Data in Packet Mode	72
3-4	Parent Meta Data in Packet Mode when Packet is Replicated	73
3-5	Rx Status Valid Values	75
4-1	Receive Reassembly Context	83
4-2	Statistics Counter Offsets from Base for Port	84
4-3	Jump Table Entries	84
4-4	Compile Time Build Switches Relevant to the Packet Receive Microblock	85
4-5	Performance Analysis of the Block	91
4-6	Packet RX Block Performance Analysis: I/O Operations for min Packet	91
4-7	Packet RX Microblock Characterization Data	92
5-1	CSIX Receive Algorithm for a Single Microengine Design	97
5-2	CSIX RX Algorithm Running on Two Microengines	98
5-3	CSIX Receive Reassembly Context for one Microengine	99
5-4	CSIX Receive Reassembly Context for First Microengine	99
5-5	CSIX Receive Reassembly Context for Second Microengine	100
5-6	CSIX Receive Lookup Key for One Microengine Design	100
5-7	CSIX Receive Lookup Key for Two Microengine Design	100
5-8	CSIX Receive: Statistics Counter Offsets from Base for VOQ	100
5-9	Compile Time Options Used in the One Microengine CSIX Receive Microblock....	101
5-10	Compile Time Options Used in the Two Microengine CSIX Receive Microblock....	101
5-11	CSIX RX—Budget and Cycle Count for One Microengine Design	101
5-12	CSIX RX—Budget and Cycle Count for Two Microengine Design	101
5-13	CSIX RX Performance Analysis—I/O Operations for Min Packet Worst Case .	102
5-14	CSIX Rx Microblock Characterization Data	102
6-1	An Example of Split Port Number Bits	108
6-2	AAL5 RX Microblock Pre-Processor Switches	109
6-3	RXC Data Structure	111
6-4	Valid Combinations	112
6-5	Port Based Counters	113
6-6	VC Based Counters	113
6-7	CRC32 From Two LW Used to Find a Hash Table	114
6-8	Buckets Structure in the Primary Hash Table	114
6-9	Buckets Structure in the Secondary Hash Table	114
6-10	VC_key#	115
6-11	Instruction Estimates—Two Microengine Design	130
6-12	Break up of the Critical Path Cycles for the Worst Case Scenario	130
6-13	Instruction Estimates and I/O Usage for ATM RX Microblock	131
6-14	SRAM Memory	132
6-15	Local Memory	132
6-16	Registers and Signals—Two Microengine Design	132
6-17	Registers and Signals—Single Microengine Design	133
6-18	I/O Commands—Two Microengine Design	133

6-19	I/O Commands—Single Microengine Design.....	134
6-20	ATM AAL5 RX Microblock Characterization Data	134
7-1	CSIX Transmit Context	142
7-2	CSIX Transmit Control Word.....	143
7-3	CSIX Traffic Manager Header Added for Each c-frame	143
7-4	CSIX Traffic Manager Header Added for First c-frame	143
7-5	Reference CSIX Base Header - 1 Shortword.....	144
7-6	Reference CSIX Unicast Extension Header.....	144
7-7	CSIX Transmit Statistics	144
7-8	Logical Phases of CSIX TX Block	145
7-9	Logical Phases of CSIX TX Block Interleaving for Multiple Requests.....	146
7-10	CSIX Transmit Performance Analysis: Worst Case Cycle Count	150
7-11	Instruction Cycle Counts in the Critical Path	150
7-12	CSIX Transmit Performance Analysis: I/O Operations	151
7-13	CSIX Tx Microblock Characterization Data	151
8-1	Possible Solutions Based on Three Packet Transmit Modes	155
8-2	Context-Relative Thread Execution Status Flag—exe_stat_flag	156
8-3	TX Request - One Longword	157
8-4	Queue Entry Structure	157
8-5	Queue Descriptor Structure	158
8-6	Output Transmit Control Word - Two Longwords.....	159
8-7	Compile Time Options Used in the CSIX Receive Microblock.....	159
8-8	Packet TX Performance Analysis - Worst Case Cycle Count	168
8-9	Packet TX Performance Analysis - Worst Case I/O Operations	168
8-10	Packet TX for SPHY and MPHY-4 Microblock Characterization Data	168
9-1	Packet TX for MPHY-16: Globals Stored in Absolute Registers	174
9-2	Context Relative Thread Execution Status Flag - exe_stat_flag.....	174
9-3	TX Request - One Longword	175
9-4	Queue Entry Structure	175
9-5	Queue Descriptor Structure	176
9-6	Output Transmit Control Word - Two Longwords.....	177
9-7	Packet TX Statistics	177
9-8	Performance Analysis - Instruction Estimate	184
9-9	Performance Analysis - I/O Operations.....	184
9-10	Packet TX for MPHY-16 Microblock Characterization Data	184
10-1	Three-word NN Ring between the first and the Second Microengine	190
10-2	Six-word NN Ring for Non-sop M-packet	190
10-3	Instruction Cycle Counts in the Critical Path for Two Microengine Design	192
10-4	Performance Requirements for OC-192 POS—Two Microengine Design.....	192
10-5	Packet TX for OC-192 POS Microblock Characterization Data	192
11-1	Format of 8-byte CPCS Trailer.....	197
11-2	Format of 5-byte Cell Header.....	198
11-3	Format of TXC1.....	202
11-4	Format of TXC2.....	203
11-5	Counters Available via the Counters Build Switch	204
11-6	AAL5 TX Microblock Switches	204
11-7	OC-48 Algorithm - Pipestage 1	205
11-8	OC-48 Algorithm - Pipestage 2	206
11-9	Quad OC-12 Algorithm Pipestage 1.....	206
11-10	Quad OC-12 Algorithm Pipestage 2.....	206

11-11	OC-48 Performance Analysis.....	211
11-12	OC-48 Critical Path Cycles per Stage.....	211
11-13	Quad OC-12 Performance Analysis.....	211
11-14	Quad OC-12 Critical Path Cycles per Stage	212
11-15	AAL5 TX Microblock Resource Usage.....	212
11-16	AAL5 TX Microblock Characterization Data	212
11-17	Number of Cycles in Worst Case for Tasks Performed in Critical Path	215
12-1	Globals Stored in Absolute Registers.....	219
12-2	Context Relative Thread Execution Status Flag.....	219
12-3	Transmit Request—One Long Word	220
12-4	Packet Queue Entry for Each Port	220
12-5	Structure of Port Status for a Port in the Local Memory.....	221
12-6	Output Transmit Control Word—2 Long Words	223
12-7	32-bit Statistics Counters in SRAM	223
12-8	Instruction Estimate for IXP24XX Packet Transmit Microblock.....	229
12-9	I/O Operations Performed in Different Phases.....	229
12-10	Instruction Estimate for IXP28XX Packet Transmit Microblock.....	230
12-11	I/O Operations Performed in Different Phases of the Microblock.....	230
12-12	Packet TX-Multiports Microblock Characterization Data	230
13-1	Compile Time Options Used in the Queue Manager.....	239
13-2	Ingress Queue Manager Performance Analysis: Worst Case Cycle Count	246
13-3	Ingress Queue Manager Performance Analysis: I/O Operations	246
14-1	Compile Time Options Used in the Queue Manager.....	249
14-2	Differences between the OC-48 and OC-192 Queue Manager	251
14-3	QM OC-192—Worse Cycle Count Estimate for Different Phases.....	251
14-4	IQM OC-192—I/O Operations Performed in the Different Phases.....	251
14-5	OC-192 Queue Manager Microblock Characterization Data.....	252
15-1	Egress Queue Manager Performance Analysis: Cycle Count.....	256
15-2	Packet Queue Manager Microblock Characterization Data	256
16-1	Scratch Ring (LCSNAP Encap and ATM QM)—Three Long Words	261
16-2	NN Ring (ATM QM and TM 4.1 Shaper)—Two Long Words	261
16-3	Scratch Ring (TM 4.1 Scheduler and ATM QM)—One Long Word.....	262
16-4	Scratch Ring (ATM QM and AAL-5 TX)—Two Long Words.....	262
16-5	ATM QM Compile Time Switches	263
16-6	Cycle Count.....	271
16-7	I/O Operations Performed in Different Phases of ATM QM	271
16-8	ATM Queue Manager Microblock Characterization Data	271
17-1	Ingress Scheduler Globals	279
17-2	Queue Group Fields	279
17-3	Queue Fields	279
17-4	Instruction Cycle Estimates for Scheduler Components	287
18-1	Two-word VoQ in Local Memory	290
18-2	Globals Stored in Absolute Registers Shared by all Threads	290
18-3	Format of the Active List Register	290
18-4	Instruction Cycle Count for the Scheduler.....	297
18-5	Fabric Scheduler For OC-192 Microblock Characterization Data	297
19-1	Egress Scheduler: Port-Specific Fields	304
19-2	Egress Scheduler: Queue-Specific Fields.....	305
19-3	Egress Scheduler: Registers for Data Structures.....	305
19-4	Egress Scheduler Performance Analysis: Cycle Counts.....	315

20-1	virtual Queue with 4 longwords Data Structure	327
20-2	Data Structure Stored on Each Port.....	327
20-3	Data Structures Stored in Registers.....	328
20-4	OC-192 DRR—Worse Cycle Count Estimate	329
20-5	OC-192 DRR Microblock Characterization Data	329
21-1	Cycle Count Table (including unfilled defers).....	335
21-2	I/O Latency Analysis Table	335
21-3	Memory Access Summary Table	335
21-4	SRAM Footprint.....	336
21-5	Scratchpad Footprint.....	336
21-6	Code Store Footprint.....	336
22-1	Transition Message Format (NN ring between QM and Scheduler)	341
22-2	Flow Control Message Format (Scheduler xfer registers).....	341
22-3	Dequeue Message Format (scratch ring between Scheduler and QM)	342
22-4	Port Record	343
22-5	Queue Record.....	344
22-6	Global Register	344
22-7	Cycle Count Table (including unfilled defers).....	351
22-8	I/O Latency Analysis Table	351
22-9	Memory Accesses Summary Table	351
22-10	Code Store Footprint.....	351
23-1	GRCA Write-out Block for Each Cell for Low-bit Rate VCs.....	363
23-2	GRCA Write-out Block for Plain UBR w/priority VCs	363
23-3	GRCA Write-out Block for Each Cell for High-bit Rate VCs	363
23-4	Definition of Variables Stored in Local Memory	371
23-5	UBR w/priority Table Entry.....	372
23-6	Instruction Counts for the Blocks	384
23-7	Local Memory Map.....	385
23-8	Performance-oriented Port State Data Structure	390
23-9	Updated Local Memory Map	391
23-10	Worst and Average Cycle Count Estimate for the Shaper Microblock.....	393
23-11	Worst and Average Cycle Count Estimate for the Scheduler Microblock	393
23-12	I/O Operations Performed in the worst Case by the Shaper and Scheduler.....	394
24-1	Listing of Build Switches	400
24-2	Table of Next Microengine Values	401
24-3	RFC1812 <i>MUST</i> Check Items.....	402
24-4	RFC1812 <i>SHOULD</i> Check Items	402
24-5	Optional RFC2644 Check Items	403
24-6	Control Block Layout	404
24-7	Next Hop Information Associated with Each Route—Uncompressed Format ..	405
24-8	Next Hop Information Associated with Each Route—Compressed Format	406
24-9	Packet Action Flags	406
24-10	IPv4 Counters and SRAM Offsets.....	408
24-11	Trie Entry Layout.....	409
24-12	List of IPv4 Microblock Symbols	414
24-13	IPv4 Microblock Exceptions	414
24-14	IPv4 Forwarder Cycle Counts	418
24-15	IPv4 Forwarder I/O Latency Analysis for Minimum Packet	418
24-16	IPv4 Characterization Data	418
25-1	Compile Time Build Switches.....	421

25-2	Table of Next ME Numbers	422
25-3	RFC2460 “Must” Checks Performed in the Microblock	423
25-4	RFC2373 <i>Should</i> Checks Performed in the Microblock	423
25-5	Fields of a <i>Real</i> Next-Hop Entry	424
25-6	Generic Flags	425
25-7	Next-Hop ID Types	425
25-8	Fields of an Intermediate Next-Hop Entry	426
25-9	Counter Offsets in Incoming Statistics Block	428
25-10	Counter Offsets in Outgoing Statistics Block	429
25-11	Longest Prefix Match Algorithm Performance	437
25-12	IPv6 Symbols	438
25-13	IPv6 Exception Codes	439
25-14	IPv6 Cycle Count Analysis	443
25-15	IPv6 I/O Latency Analysis for Min Packet	443
25-16	IPv6 Forwarder Microblock Characterization Data	444
26-1	IPv6 to IPv4 Tunneling Build Switches	449
26-2	Next ME Values	450
26-3	RFC 3056 <i>Must</i> Checks Performed in the Microblock	451
26-4	RFC 2893 <i>Should</i> Items for IPv4 Source Address	451
26-5	RFC 2893 <i>Should</i> Items for IPv6 Source Address	451
26-6	RFC 2893 <i>Must</i> Items for IPv4 Destination Address	452
26-7	RFC 3056 <i>Must</i> Items for IPv6 Source and Destination Address	452
26-8	Tunnel Next-Hop Entry for V6V4-Tunnel-Decap Microblock	455
26-9	Format of Ingress Source List Block	456
26-10	Next Hop Fields	465
26-11	Tunnel Next-Hop Flags for V6V4-Tunnel-Encap Microblock	466
26-12	IPv4 Header Creation for Encapsulated Packets	469
26-13	Symbols	470
26-14	Exception Codes	471
26-15	Cycle Count Analysis	471
26-16	I/O Latency Analysis for Min Packet	471
27-1	Buffer Parameters Defined	476
27-2	Compile Time Build Switches	477
27-3	Counters and Offsets from the Base Address	478
27-4	NAT Table Data Format	480
27-5	V6-V4 NAT Hash Index	482
27-6	Data Format	482
27-7	IPv4 Address Used for NAPT-PT	483
27-8	V6-V4 NAPT Hash Index	485
27-9	Symbols Required for NAT-PT Microblock and the Core component to Work ..	493
27-10	Exception Codes Generated by the NAT-PT Microblock	493
27-11	Translation Microblock Cycle Count Analysis	494
27-12	I/O Latency Analysis (IPv4-TCP)	494
27-13	I/O Latency Analysis (IPv6-TCP)	494
27-14	IPv6 to IPv4 Tunnel Decap Microblock Characterization Data	495
27-15	IPv6 to IPv4 Tunnel Encap Microblock Characterization Data	497
28-1	Layer-2 Decapsulation Worst Case Instruction Count	506
28-2	Ethernet Decap Microblock Characterization Data	507
28-3	PPP Decap Microblock Characterization Data	509
29-1	Format of an Entry in the Layer-2 Table for LLC SNAP	514

29-2	Layer-2 Encapsulation Worst Case Cycle Count	514
29-3	Layer-2 Encapsulation I/O Operations	514
29-4	Ethernet Encap Microblock Characterization Data	515
29-5	PPP Encap Worst Case Cycle Count	517
29-6	PPP Encap I/O Operations	517
29-7	PPP Encap Microblock Characterization Data	518
29-8	LLCSNAP Encap Microblock Characterization Data	520
30-1	Build Switches for IPv4 6-tuple Classifier Microblock	527
30-2	Input Variables Consumed	528
30-3	Output Variables Modified	528
30-4	Variables Imported by this Block	529
30-5	Hash Entry Definition	531
30-6	Cycle Count Table (including unfilled defers)	537
30-7	I/O Latency Analysis Table	537
30-8	Memory Accesses Summary Table	538
30-9	DRAM/SRAM Footprint	538
30-10	Code Store Footprint	538
31-1	Build Switches for TCM Microblock	542
31-2	Input Variables Consumed by SRTCM	543
31-3	Output Variables Modified by SRTCM	543
31-4	Variables Imported by this Block	543
31-5	SRTCM Table Entry Definition	546
31-6	Cycle Count Table for SRTCM Version (including unfilled defers)	557
31-7	Cycle Count Table for TRTCM Version (including unfilled defers)	557
31-8	Cycle Count table for SRTCM and TRTCM version (including unfilled defers)	558
31-9	I/O Latency Analysis Table (for all versions)	558
31-10	Memory Access Summary Table	558
31-11	SRAM Footprint on Ingress IXP 2400	558
31-12	SRAM Footprint on Egress IXP 2400	559
31-13	Code Store Footprint	559
32-1	Variables Consumed by DSCP Marker	561
32-2	Variables Modified by DSCP marker	561
32-3	Cycle count table (including unfilled defers)	563
32-4	Code Store footprint	564
33-1	Build Switches for DSCP Classifier Microblock	566
33-2	Input Variables Consumed	567
33-3	Output Variables Modified	567
33-4	Variables Imported by this Block	568
33-5	Classification Result Entry Definition	569
33-6	Cycle Count Table (including unfilled defers)	574
33-7	I/O Latency Analysis Table	574
33-8	SRAM Footprint	575
33-9	Code Store Footprint	575
34-1	Build Switches for WRED Microblock	580
34-2	Input Variables Consumed by WRED	581
34-3	Output Variables Modified by WRED	581
34-4	Variables Imported by this Block	582
34-5	WRED Table Entry Definition	586
34-6	Cycle Count Table (including unfilled defers)	596
34-7	I/O Latency Analysis Table—WRED Low-speed Queues	596



34-8	Memory Access Summary Table	596
34-9	SRAM Footprint.....	597
34-10	Code Store Footprint.....	597
34-11	Cycle Count Table (including unfilled defers).....	597
34-12	I/O Latency Analysis Table - WRED for High-Speed Queues.....	598
34-13	Memory Access Summary Table	598
34-14	SRAM Footprint.....	598
34-15	Code Store Footprint.....	598
35-1	Input Variables Consumed by FTN Forwarder.....	603
35-2	Output Variables Modified by FTN Forwarder.....	604
35-3	NHLFE Entry	606
35-4	NHLFE Set Entry.....	607
35-5	NHLFE Counters Table Entry.....	608
35-6	Cycle Count Table (including unfilled defers) - UNIFORM Tunnel Mode.....	614
35-7	Cycle Count Table (including unfilled defers) - PIPE Tunnel Mode	614
35-8	I/O Latency Analysis Table.....	614
35-9	Data Structures Footprint	615
35-10	Code Store Footprint.....	615
35-11	FTN Forwarder Microblock Characterization Data	615
36-1	Input Variables Consumed by ILM Forwarder.....	621
36-2	Output Variables Modified by ILM Forwarder.....	622
36-3	ILM Table Entry	624
36-4	ILM_NHLFE Set Entry.....	625
36-5	InSegment Counters Table Entry.....	626
36-6	Input Port Counters Table	627
36-7	Cycle Count Table (including unfilled defers) for Operation SWAP	634
36-8	Cycle Count Table (including unfilled defers) for Operation POP	634
36-9	Cycle Count Table (including unfilled defers) for Operation POP_FORWARD.....	634
36-10	Cycle Count Table (including unfilled defers) for Operation SWAP_PUSH	635
36-11	I/O Latency Analysis Table.....	635
36-12	Data Structures Footprint	635
36-13	Code Store Footprint.....	635
36-14	ILM Forwarder Microblock Characterization Data	636
37-1	Packet Copier Microblock Build Switches	641
37-2	Packet Copier Microblock Symbols.....	642
37-3	Data Flow through Packet Copier	643
37-4	Packet Copier Request Message.....	644
37-5	Packet Copier Response Message.....	644
37-6	Packet Copy Context Queue Entry	644
37-7	Packet Replication Algorithm Running on One Microengine.....	645
37-8	Packet Replication Algorithm Running on Two Microengines.....	646
37-9	Cycle Counts for Packet Replication on Two Microengines.....	646
37-10	Packet Copier Microblock Characterization Data.....	647
38-1	Freelist Manager Microblock Algorithm	652
38-2	Budget and Cycle Count	652
38-3	I/O Operations for Minimum Packet Worst Case	653
38-4	Freelist Manager Microblock Characterization Data	653
39-1	Dynamic Properties and Clients.....	662
39-2	Properties Data Structure.....	662
39-3	Property API	663

39-4	Core Component Functionality Mapped to Different Framework Layers	666
40-1	Re-initialization Procedure Options.....	686
40-2	Loading Microcode and Starting Engines	686
40-3	Loading Microcode and Starting Engines	687
40-4	User Initialization and Shutdown Hooks.....	687
40-5	Core Component Configuration Parameters.....	689
41-1	POS RX Core Component Static Configuration Data	696
41-2	POS RX Core Component Patch Symbols	697
41-3	POS RX Core Component Infrastructure API	697
41-4	POS RX Core Component Messaging API	697
41-5	POS RX Core Component Library API	698
42-1	CSIX RX Core Component	701
42-2	CSIX RX Core Component Infrastructure API	701
42-3	CSIX RX Core Component Messaging API	701
42-4	CSIX RX Core Component Library API.....	702
43-1	Ethernet RX Core Component Dynamic Configuration Data	706
43-2	Ethernet RX Core Component Static Configuration Data	706
43-3	Symbols and Memory Base Addresses to Patch Packet RX Microblock	707
43-4	Symbols and Memory Base Addresses L2 Decap Microblock.....	707
43-5	Ethernet RX Core Component Functions.....	708
43-6	Ethernet Interface RX Messaging Functions.....	708
43-7	Ethernet Interface RX Library Functions	709
44-1	Patch Symbols for CSIX TX Core Component.....	715
44-2	CSIX TX Core Component Infrastructure API	715
44-3	CSIX TX Core Component Messaging API.....	715
44-4	CSIX TX Core Component Library API	716
45-1	ATM/POS TX Core Component Dynamic Configuration Data	719
45-2	ATM/POS Interface TX Configuration Parameters	720
45-3	ATM/POS Media Card Operation Mode.....	721
45-4	Supported Line Side Interface and Media Interface for each OC Mode	722
45-5	Symbols and Memory Base Addresses Patched into ATM TX Microblock	722
45-6	Symbols and Memory Base Addresses Patched into POS TX Microblock	723
45-7	ATM/POS TX Core Component Infrastructure API	724
45-8	ATM/POS TX Core Component Messaging API	724
45-9	ATM/POS TX Core Component Library API	724
46-1	Ethernet TX Core Component Input Sources	727
46-2	Dynamic Configuration Data	728
46-3	Ethernet TX Core Component Static Configuration Data.....	729
46-4	Symbols and Memory Base Addresses to be Patched	730
46-5	Ethernet TX Core Component Data Structures.....	732
46-6	Ethernet TX Core Component Infrastructure API.....	732
46-7	Ethernet TX Messaging Functions	733
46-8	Ethernet TX Library API	733
47-1	Ethernet ARP Library API	741
48-1	Queue Manager Core Component Configuration Items.....	746
48-2	Queue Manager Core Component Infrastructure API.....	748
48-3	Queue Manager Core Component Messaging API.....	748
48-4	Queue Manager Core Component Library API	748
50-1	Scheduler Core Component Configuration Items	755
50-2	Scheduler Core Component Infrastructure API.....	757

51-1	New Patching Symbols in Scheduler (DiffServ) Core Component.....	759
52-1	IPv4 Forwarder Core Components Statistics	765
52-2	RFC 1812 Checks for IP Addresses	770
52-3	IPv4 Forwarder Core Component Data Structures, Types, and Macros	773
52-4	IPv4 Forwarder Core Component Infrastructure API	774
52-5	IPv4 Forwarder Message Helper API.....	774
52-6	IPv4 Forwarder Library API	775
53-1	Counter Offsets in Incoming Statistics Block.....	779
53-2	Counter Offsets in Outgoing Statistics Block.....	780
53-1	IPv6 Forwarder Data Structures, Types and Macros	787
53-2	IPv6 Forwarder Core Component Infrastructure API.....	787
53-3	IPv6 Forwarder Message Helper API.....	788
53-4	IPv6 Forwarder Library API	789
54-1	IPv6-IPv4 Tunneling Data Structures, Types and Macros.....	798
54-2	IPv6 to IPv4 Tunneling Core Component Infrastructure AP	799
54-3	IPv6 to IPv4 Tunneling Core Component Message Helper API	799
54-4	IPv6 to IPv4 Tunneling Library API	801
55-1	Data Structures, Types and Macros in Translation Core Components	809
55-2	Translation Core Component Infrastructure API	809
55-3	Message Helper API in the Translation Core Component	810
55-4	Library API in the Translation Core Component.....	811
56-1	Static Configuration Items in 6-tuple Classifier.....	820
56-2	Data Structures for Configuring Exact-match Rules.....	823
56-3	6-Tuple Classifier Core Component Infrastructure API	824
56-4	6-Tuple Classifier Core Component Message Helper API	824
56-5	6-Tuple Classifier Core Component Library API	824
56-6	6-Tuple Classifier Core Component Modules	825
57-1	Static Configuration Items in TCM.....	828
57-2	Variables Imported by this Block	829
57-3	TCM Core Component Data Structures	832
57-4	TCM Core Component Infrastructure API	832
57-5	TCM Core Component Message Helper API	832
57-6	TCM Core Component Library API	833
57-7	TCM Core Component Modules.....	833
58-1	Static Configuration Items in WRED core component.....	836
58-2	Variables Imported by this Block	837
58-1	WRED Core Component Data Structures for Message Helper and Library APIs ... 840	
58-2	WRED Core Component Data Structures for Message Helper and Library APIs ... 840	
58-3	WRED Message Helper API	840
58-4	WRED Library API.....	841
58-3	WRED Core Component Modules	841
59-1	Mapping between Output Identifiers.....	844
59-2	Static Configuration Items in 6-tuple Classifier.....	846
59-3	SRTCM Core Component Data Structures	848
59-4	SRTCM Core Component Infrastructure API	849
59-5	SRTCM Core Component Message Helper API	849
59-6	SRTCM Core Component Library API	849
59-7	SRTCM Core Component Modules.....	850

60-1	Route Table Manager Data Structures and Types.....	862
60-2	Route Table Manager Macros.....	862
60-3	Route Table Manager API.....	862
61-1	RTMv6 Data Structures, Types and Macros	869
61-2	RTMv6 Core COmponent Infrastructure API	870
62-1	L2 Table Entry.....	872
62-2	Data Structures, Types, Definitions and Enumerations in L2TM	876
62-3	L2 Table Manager Library API	876
63-1	Components of the Message Helper System.....	880
63-2	Types of Calls Supported by the Message Helper	881
63-3	ix_msup_call_table_s Structure Members	887
63-4	ix_msup_call_s Structure Members - Asynchronous and Synchronous	887
63-5	ix_msup_call_s Structure Members - Asynchronous only	887
63-6	ix_msup_call_s Structure Members - Synchronous only	887
63-7	Message Support Library API	892
64-1	Stack Driver Core Component Data Structures and Types	900
64-2	Stack Driver Core Component Infrastructure API	901
64-3	Stack Driver Core Component Infrastructure Separation API.....	901
64-4	Stack Driver Packet and Message Processing API	903
64-5	Stack Driver Properties API	903
64-6	Stack Driver Packet Classifier Data Structures.....	904
64-7	Stack Driver Packet Classifier External API.....	905
64-8	Stack Driver Outgoing Packet Classifier Data Structures	906
64-9	Stack Driver Outgoing Packet Classifier Internal API	907
64-10	Stack Driver VIDD System Data Structures	909
64-11	Stack Driver VIDD Local Data Structures	909
64-12	Required Functions for the Stack Driver MUX Interface	909
64-13	VIDD System API.....	910
64-14	VIDD MUX API.....	911
64-15	VIDD System Data Structures for Linux.....	913
64-16	VIDD System API for Linux.....	913
64-17	VIDD Linux Driver Support API.....	914
64-18	Transport Module Data Structures	914
64-19	Transport Module External API	915
65-1	General Rules for Calling Sequences	933
65-2	SAR Control MAIN Configuration Items	933
65-3	SAR Data Structures and Data Type Definitions	933
65-4	SAR Core Component Infrastructure API	934
65-5	SAR Messaging API.....	934
65-6	SAR Library API	935
65-7	SAR Control Plug-in API	935
65-8	Configuration Parameters	937
65-9	ATM RX Core Component Static Configuration Properties	940
65-10	ATM RX Core Component Patch Symbols	940
65-11	IXP2400 16 ports—Key for Hash Table	941
65-12	2048 ports—Key for Hash Table.....	941
65-13	Core Component Infrastructure API.....	942
65-14	ATM/POS TX Core Component Dynamic Configuration Data	944
65-15	ATM Interface TX Configuration Parameters.....	944
65-16	ATM TX Core Component Infrastructure API.....	944



65-17	List of Shared Tables	945
65-18	TM4.1 Core Component supported Core Component Infrastructure API	949
66-1	Static Configuration Data	956
66-2	Patching Symbols for ILM Forwarder Microblock.....	956
66-3	Patching Symbols for FTN Forwarder Microblock.....	957
66-4	Data Types and Structures.....	960
66-5	MPLS Core Component Infrastructure API	960
66-6	MPLS Message Helper API.....	961
66-7	MPLS Library API.....	961
A-1	DTD Qualifiers and Syntax.....	970
A-2	DTD: Value Description.....	971
B-3	Configuration Switches.....	987
B-4	CPS RXC Data Structure	988
B-5	CID RXC Data Structure	988
B-6	Primary Hash Lookup Table.....	989
B-7	Secondary Hash Lookup Table	989
B-8	AAL-2 Receive Microblocks CRC-5 SRAM Tables	990
B-9	AAL-2 RX Counters.....	990
B-10	Cycle Counts and I/O Latencies.....	997
C-11	AAL2 TX Supported Pre-Processor Switches	1013
C-12	AAL-2 Transmit VC Context Entry.....	1014
C-13	AAL-2 TX Microblocks CRC-5 SRAM tables.....	1015
C-14	AAL2 Transmit Counters.....	1016
C-15	Cycle Counts and I/O Latencies.....	1016



2-1	Flat Queueing on Ingress IXP2400 (single packet).....	60
2-2	Flat Queueing on Ingress IXP2400 (multiple packets)	61
2-3	Hierarchical Queueing on Egress IXP2400 (single packet)	62
2-4	Hierarchical queueing on the Egress IXP2400 (Multiple Packets)	63
3-1	Packet Replication Example.....	68
3-2	Packet Replication in Cell Mode.....	69
3-3	Packet Replication in Packet Mode.....	70
3-4	RX Status Field Definition	75
4-1	Packet RX State Machine	82
4-2	Packet Receive on Two Microengines	86
4-3	Design for the First Receive Microengine	86
4-4	Design for the First Receive Microengine	87
4-5	POS Receive Flowchart: Page 1 of 3.....	88
4-6	POS Receive Flowchart: Page 2 of 3.....	89
4-7	POS Receive Flowchart: Page 3 of 3.....	90
6-1	Structure of Sub-layers in AAL5.....	105
6-2	Three Stage Functional Pipeline for Two Microengine AAL5 RX Design.....	107
6-3	Four Stages Running on Two Microengines Independently.....	107
6-4	A Maximum of 2048 Ports Addressing Scheme	108
6-5	Collisions Resolutions	116
6-6	IXP2000 ATM RX Flow Chart.....	122
6-7	Pipe Stage 1 Start: Phase 2 Flow Chart.....	123
6-8	SOP and EOP Flow Chart.....	124
6-9	EOP Only Flow Chart (Page 1 of 2)	125
6-10	EOP Only Flow Chart (Page 2 of 2)	126
6-11	MOP Flow Chart.....	127
6-12	SOP Only Flow Chart.....	128
6-13	Pipe Stage 3: Phase 6 Flow Chart	129
7-1	CSIX TX Microblock Running on Two Microengines.....	146
7-2	CSIX Transmit: Flowchart for Phase 1 of Two-Phase Scheme.....	148
7-3	CSIX Transmit: Flowchart for Phase 2 of Two-Phase Scheme.....	149
8-1	SPHY Packet TX Flow Chart: Part 1 of 6	162
8-2	SPHY Packet TX Flow Chart: Part 2 of 6	163
8-3	SPHY Packet TX Flow Chart: Part 3 of 6	164
8-4	SPHY Packet TX Flow Chart: Part 4 of 6	165
8-5	SPHY Packet TX Flow Chart: Part 5 of 6	166
8-6	SPHY Packet TX Flow Chart: Part 6 of 6	167
9-1	Packet Transmit—MPHY-16 Configuration—Flowchart Page 1 of 5	179
9-2	Packet Transmit—MPHY-16 Configuration—Flowchart Page 2 of 5	180
9-3	Packet Transmit—MPHY-16 Configuration—Flowchart Page 3 of 5	181
9-4	Packet Transmit—MPHY-16 Configuration—Flowchart Page 4 of 5	182
9-5	Packet Transmit—MPHY-16 Configuration—Flowchart Page 5 of 5	183
10-1	High-level Design for the First Microengine	191
10-2	High-level Design for the Second Microengine	191
11-1	Format of CPCS-PDU	197
11-2	Format of the 53-byte ATM Cell	198
11-3	Extended Port Addressing.....	199
11-4	Format of the FPGA Header	199
11-5	Two Stage Functional Pipeline for OC-48.....	200
11-6	Two MEs Running Independently	201

12-1	picked_from_tx_request.....	225
12-2	picked_from_virtual_queue_or_picked_none Path	227
13-1	Format of Queue Descriptor.....	238
13-2	Ingress Queue Manager: Phase Initialize	240
13-3	Ingress Queue Manager: Phase 1	241
13-4	Ingress Queue Manager: Phase 2	242
13-5	Ingress Queue Manager: Phase 3	242
13-6	Ingress Queue Manager: Flow Chart Page 1 of 2.....	244
13-7	Ingress Queue Manager: Flow Chart Page 2 of 2.....	245
14-1	QM OC-192 Format of Queue Descriptor	248
14-2	Basic Functionality of the Queue Manager	250
16-1	Format of Queue Descriptor.....	260
16-2	Initialize	264
16-3	Phase 1	264
16-4	Phase 2	265
16-5	Phase 3	265
16-6	Queue Manager Operations Algorithm (Page 1 of 5).....	266
16-7	Queue Manager Operations Algorithm (Page 2 of 5).....	267
16-8	Queue Manager Operations Algorithm (Page 3 of 5).....	268
16-9	Queue Manager Operations Algorithm (Page 4 of 5).....	269
16-10	Queue Manager Operations Algorithm (Page 5 of 5).....	270
17-1	Hierarchical Bit Vector for Ingress Scheduler	278
17-2	Threads in the CSIX scheduler	280
17-3	Format of a CSIX Flow Control Frame.....	281
17-4	Scheduler Thread: Flowchart Page 1 of 2.....	283
17-5	Scheduler Thread: Flowchart Page 2 of 2.....	284
17-6	Flow Control Thread: Flowchart.....	285
17-7	QM Message Handler Thread: Flowchart	286
18-1	High Level Algorithm for the Scheduler.....	291
18-2	Processing of Enqueue Requests.....	291
18-3	Processing of Dequeue Requests.....	292
18-4	Format of a CSIX Flow Control Frame.....	293
18-5	Implementing a Hierarchal Scheduler (1 of 2).....	295
18-6	Implementing a Hierarchal Scheduler (2 of 2).....	296
19-1	High Level Components in the Egress Scheduler.....	302
19-2	Scheduler Thread: Flowchart Page 1 of 2.....	311
19-3	Scheduler Thread: Flowchart Page 2 of 2.....	312
19-4	Queue Manager Message Handler Thread: Flowchart Page 1 of 2.....	313
19-5	Queue Manager Message Handler Thread: Flowchart Page 2 of 2.....	314
20-1	High Level Components in the Egress Scheduler.....	320
21-1	Queue Data Structures in SRAM (maintained by Queue Manager)	334
22-1	Existing implementation of WRR/DRR egress scheduler	337
22-2	Enhanced Egress Scheduler with Hierarchic WRR/SP/DRR.....	338
22-3	Interfaces between Scheduler and Collaborating Microengines	340
22-4	Scheduler Microblock Decomposition	342
22-5	Common Data Structures, Shared between Two Scheduler Threads	343
22-6	Synchronization between scheduler threads	345
22-7	QM Message Handler Thread.....	347
22-8	Scheduler Main Loop	348
22-9	Queue Group Scheduling (Strict Priority).....	349

22-10	Queue Scheduling (Deficit Round Robin)	350
23-1	GCRA(T,t) on the Arrival of a New Cell.....	353
23-2	TM4.1 Architecture overview.....	356
23-3	Time Queues concept	357
23-4	Time Queue Data structure in SRAM for Low Bit Rate VCs	358
23-5	Time Queue in Local Memory for Very High Bit Rate VCs.....	360
23-6	External Interfaces to TM 4.1 Blocks.....	362
23-7	GCRA Flow Chart.....	378
23-8	F-GCRA Shaping Macro Flow Chart	379
23-9	Shaper (Macro only) Flow Chart	380
23-10	Write-out Block Flow Chart.....	381
23-11	Scheduler Flow Chart.....	382
23-12	Dequeue Flow Chart	383
23-13	Sample Port Shaping Table	387
23-14	Correlation between the PortShaping table and the HBR TQ table	388
24-1	IPv4 Microblock Dependencies	400
24-2	Directed Broadcast Table Layout.....	404
24-3	Nexthop Database Layout.....	408
24-4	Route Table Lookup	410
24-5	Trie Table with One Route	411
24-6	Longest Prefix Match Dual Lookup	412
24-7	Trie Table with Two Routes.....	413
24-8	High Level Flow Chart for IPv4 Microblock	416
24-9	IPV4 Microblock Processing Nexthop Information	417
25-1	IPv6 Microblock Dependencies	421
25-2	Structure of a <i>Real</i> Next-Hop Entry.....	424
25-3	Structure of an Intermediate Next-Hop Entry	426
25-4	Next-Hop Database Layout.....	427
25-5	Structure of an IPv6 Address	430
25-6	Route Table Data Structures—Trie-Blocks and Trie-Entries.....	430
25-7	Route Table Data Structures—Multi-Way Tree which Trie-Blocks.....	431
25-8	Representation of a 40-Bit Route Entry 0x3fff020304 Using Trie-Blocks.....	432
25-9	Pseudocode for the Basic Lookup Algorithm.	433
25-10	Flow Chart for the Lookup Algorithm.....	435
25-11	Route Tables with Prefix Optimization	436
25-12	Trie Table Representation for Routing Prefix FFFE:201::/32	437
25-13	Trie Table Representation for Routing Prefix FFFE:200::/32	438
25-14	IPv6 Forwarder Microblock Flowchart.....	440
25-15	IPv6 Header Validation Flowchart.....	441
25-16	IPV6 Microblock Processing Next-Hop Information Flowchart.....	442
26-1	IPv6-IPv4 Tunneling Microblock Dependencies	449
26-2	Ingress Source List Blocks	457
26-3	Process Flow for V6V4-Tunnel-Decap Microblock.....	459
26-4	Process Flow for V6V4-Tunnel-Decap Header Validation	460
26-5	Process Flow for V6V4-Tunnel-Decap Validation for Automatic Tunneling	461
26-6	Process Flow for V6V4-Tunnel-Decap Validation for 6to4 Tunneling	462
26-7	Process Flow for Tunnel Ingress Source Validation.....	463
26-8	Process Flow for V6V4-Tunnel-Encap Microblock.....	467
26-9	Obtaining the IPv4 Tunnel Endpoint Addresses.....	469
27-1	Dependencies of the IPv6 Translation Microblock and Related Modules	476

27-2	NAT State.....	481
27-3	NAPT Table and Indexes Used for Access.....	484
27-4	Process Flow for NAT-PT Microblock	487
27-5	IPv4 to IPv6 Translation (Page 1 of 2)	489
27-6	IPv4 to IPv6 Translation (Page 2 of 2)	490
27-7	IPv6 to IPv4 Translation (Page 1 of 2)	491
27-8	IPv6 to IPv4 Translation (Page 2 of 2)	492
28-1	Hash Table Entry for MAC Filtering	503
28-2	Header Type Field of Packet Metadata.....	506
30-1	Hash Table and Statistics Table Organization	530
30-2	Hash Entry Layout.....	531
30-3	6-tuple Classifier Algorithm (Page 1 of 3)	534
30-4	6-tuple Classifier Algorithm (Page 1 of 3)	535
30-5	6-tuple Classifier Algorithm (Page 3 of 3)	536
31-1	Color-aware and Color-blind Modes of SRTCM.....	540
31-2	Color-aware and Color-blind Modes of TRTCM.....	541
31-3	TCM Data Structures with Statistics.....	544
31-4	TCM Data Structures without Statistics.....	545
31-5	SRTC Meter Data Structures in Local Memory	545
31-6	TCM Inter-thread Synchronization (Page 1 of 2)	548
31-7	TCM Inter-thread Synchronization (Page 2 of 2)	549
31-8	SRTCM Token Update Macro	550
31-9	SRTC Metering Macro	552
31-10	TRTCM Token Update Macro	554
31-11	TRTC Metering Macro.....	556
32-1	DSCP Marking Algorithm	563
33-1	DSCP Rule Table Organization	568
33-2	Classification Result and Statistics Entry Layout	569
33-3	DSCP Classifier Algorithm (Page 1 of 3)	571
33-4	DSCP Classifier Algorithm (Page 2 of 3)	572
33-5	DSCP Classifier Algorithm (Page 3 of 3)	573
34-1	RED Dropping Function	578
34-2	WRED Data Structures in SRAM for Low-speed Queues.....	583
34-3	WRED Data Structures in SRAM for High-speed Queues	584
34-4	Local Memory Data Structures for Low-speed Queues	585
34-5	Local Memory Data Structures for High-speed Queues	586
34-6	WRED Inter-thread Synchronization.....	588
34-7	WRED Algorithm for Low-speed Queues.....	590
34-8	WRED algorithm - phase 1	591
34-9	WRED algorithm - phase 2	594
34-10	WRED Algorithm for High-speed Queues.....	595
35-1	FTN Forwarder Data Structures.....	605
35-2	FTN Forwarder Forwarding Algorithm - Phase 1	609
35-3	FTN Forwarder Forwarding Algorithm - Phase 2	610
35-4	MPLS Stack Entry Format.....	611
35-5	IP and MPLS Microblocks on the Same Microengines	612
35-6	FTN Forwarder Thread Synchronization.....	613
36-1	ILM Forwarder Data Structures.....	623
36-2	ILM Forwarder Algorithm - Phase 1	628
36-3	ILM Forwarder Algorithm - Phase 2 (Page 1 of 4)	629



36-4	ILM Forwarder Algorithm - Phase 2 (Page2 of 4).....	630
36-5	ILM Forwarder Algorithm - Phase 2 (Page 3 of 4).....	631
36-6	ILM Forwarder Algorithm - Phase 2 (Page 4 of 4).....	632
36-7	ILM Reserved Label Range Processing.....	633
37-1	Data Flow through Packet Copier	643
39-1	POS IPv4 Software Architecture	667
39-2	DiffServ Application Overview	668
39-3	MPLS Application Overview	669
39-4	Software Components for IPv4/IPv6 Forwarding and V6/V4 Tunneling.....	671
40-1	System Application Overview	684
40-2	Execution Engine Repository Structure.....	689
40-3	Core Component Repository Structure	690
42-1	CSIX RX Core Component.....	699
43-1	Ethernet RX Component Dependencies	703
43-2	Ethernet RX Core Component Data Flow	705
43-3	Modularity of Ethernet RX Core Component.....	707
44-1	CSIX TX Core Component	713
45-1	ATM/POS TX Core Components	718
46-1	Ethernet TX Component Dependencies.....	725
46-2	Ethernet TX Core Component Data Flow.....	727
46-3	Modularity of Ethernet TX Core Component	731
47-1	Ethernet ARP Component Dependencies.....	736
47-2	Ethernet ARP Generation Data Flow	737
47-3	Ethernet ARP Reception Data Flow	738
47-4	Modularity of the Ethernet ARP Module	740
48-1	Initialization Flow of the Queue Manager Core Componen.....	747
50-1	Scheduler Core Component Data Flow.....	754
50-2	Scheduler Core Component Initialization Data Flow	757
52-1	IPv4 Forwarder Core Component Modules.....	767
52-2	IIPv4 Forwarder Core Component Bindings.....	771
53-1	IPV6 Forwarder Core Component Dependencies.....	778
53-2	Modularity of IPV6 Core Component	781
53-3	IPV6 Forwarder Core Component Bindings	783
54-1	Tunneling Core Component Dependencies	792
54-2	Modularity of Tunneling Core Component.....	794
54-3	Tunneling Core Component Bindings	798
55-1	Translation Core Component Dependencies	804
55-2	Translation Core Component Architecture	804
55-1	Modularity of Translation Core Component.....	812
55-2	Translation Core Component Bindings	814
56-1	Data Flow of 6-tuple Classifier Core Component	818
56-2	6-tuple Classifier Core Component Dependencies	820
56-3	Initialization of a 6-tuple Classifier Core Component	822
56-4	Shutdown of a 6-tuple Classifier Core Component	823
56-5	6-Tuple Classifier Core Component Achitecture	826
57-1	TCM Core Component Dependencies	828
57-2	Initialization of TCM Core Component	829
57-3	Shutdown of TCM Core Component	830
57-4	Data flow of the TCM Core Component	831
57-5	TCM Core Component Architecture	834

58-1	WRED Core Component Dependencies	836
58-2	Initialization of the WRED Core Component	837
58-3	Shutdown of the WRED Core Component	838
58-4	Data flow of the WRED Core Component	839
58-1	WRED Core Component Architecture	842
59-1	Data flow of DSCP Classifier Core Component	843
59-2	DSCP Classifier Core Component Dependencies	845
59-3	Initialization of a DSCP Classifier Core Component	847
59-4	Shutdown of a DSCP Classifier Core Component	848
59-5	SRTCM Core Component Architecture	851
60-1	Adding Information using the Route Table Manager	856
60-2	Routes and Next Hops	856
60-3	Route Table Manager Initialization	857
60-4	Modularity of the Route Table Manager	860
60-5	Route Table Manager Initialization	861
61-1	IPv6 RTM—Adding Information	866
61-2	Routes and Next Hops—IPV6 RTM	866
61-3	RTMv6 Initialization	867
62-1	L2 Table Manager Entry State Diagram	872
62-2	L2 Table Manager in IXA SDK 3.1	873
62-3	L2 Table Manager Usage	875
63-1	Messaging Overview	880
63-2	Asynchronous Call: Data Flow	888
63-3	Synchronous Call: Data Flow	890
64-1	Stack Driver Packet Flow	896
64-2	Stack Driver Modularity	898
64-3	Stack Driver Internal Modularity	899
64-4	Stack Driver Internal Initialization and Registration	902
65-1	Example of Hardware with Two IXP Processors	919
65-2	Example of Hardware with Single IXP Processor	920
65-3	General SoftSAR Software Overview	920
65-4	SoftSAR Software Components	921
65-5	SAR Control Architecture Overview	924
65-6	Place of Software Blocks in Single IXP Hardware Platform	925
65-7	Place of Software Blocks in Dual IXP Hardware Platform	926
65-8	SAR Control Decomposition	927
65-9	Create Operation Data Flow	928
65-10	Remove Operation Data Flow	929
65-11	Read Statistics Operation Data Flow	929
65-12	SoftSAR Main Core Components Dependencies	931
65-13	SAR Control MAIN Core Components Decomposition	931
65-14	Block Ordering Convention	932
65-15	SoftSAR Agent Core Components Dependencies	936
65-16	SAR Control Agent Core Component Decomposition	937
66-1	MPLS Forwarder Core Component Packet Data Flow	954
66-2	MPLS Core Component Logical Sub-modules and Interactions	958
A-1	NetSim Framework Architecture	964
A-2	Devices and Bus Connections—Media/Switch Fabric Connections	965
A-3	Assign Media Bus Port Input	966
B-4	AAL-2 Sub-layers	985



B-5	ATM-AAL-2 Data Indication.....	991
B-6	Local Memory Queues	992
B-7	Algorithm—AAL-2 Queue Cell Microblock	993
B-8	AAL-2 CPS-SSSAR Microblock Flowchart (page 1 of 3)	994
B-9	AAL-2 CPS-SSSAR Microblock Flowchart (page 2 of 3)	995
B-10	AAL-2 CPS-SSSAR Microblock Flowchart (page 3 of 3)	996
C-11	CPS and SSSC Interaction	999
C-12	Format of AAL-2 CPS-Packet	1000
C-13	Format of SSSAR-PDU.....	1001
C-14	Pool of Threads Synchronization With CAM/LM	1002
C-15	AAL-2 TX Transmit Request Sourcing and Thread Synchronization	1003
C-16	AAL-2 Sublayer Selection and Sourcing from Local Memory Thread Queue ..	1005
C-17	AAL-2 Transmit SSSAR Sublayer Functions	1006
C-18	AAL-2 CPS Transmit Sublayer (1 of 3)	1008
C-19	AAL-2 CPS Transmit Sublayer (2 of 3)	1009
C-20	AAL-2 CPS Transmit Sublayer (3 of 3)	1010
C-21	Interface Between Timer Thread and SSSAR/CPS Threads	1011
C-22	Timer Processing Thread Flow	1012
C-23	Timer CU Data Structures in SRAM and Local Memory	1015

