# Intel® IXP400 Software

## Programmer's Guide

**June 2004**

Document Number: 252539-002c

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL® PRODUCTS. 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 Corporation may have patents or pending patent applications, trademarks, copyrights, or other intellectual property rights that relate to the presented subject matter. The furnishing of documents and other materials and information does not provide any license, express or implied, by estoppel or otherwise, to any such patents, trademarks, copyrights, or other intellectual property rights.

Intel products are not intended for use in medical, life saving, life sustaining, critical control or safety systems, or in nuclear facility applications.

The Intel® IXP400 Software v1.2.2 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.

MPEG is an international standard for video compression/decompression promoted by ISO. Implementations of MPEG CODECs, or MPEG enabled platforms may require licenses from various entities, including Intel Corporation.

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.

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 website at http://www.intel.com.

BunnyPeople, Celeron, Chips, Dialogic, EtherExpress, ETOX, FlashFile, i386, i486, i960, iCOMP, InstantIP, Intel, Intel Centrino, Intel Centrino logo, Intel logo, Intel386, Intel486, Intel740, IntelDX2, IntelDX4, IntelSX2, Intel Inside, Intel Inside logo, Intel NetBurst, Intel NetMerge, Intel NetStructure, Intel SingleDriver, Intel SpeedStep, Intel StrataFlash, Intel Xeon, Intel XScale, IPLink, Itanium, MCS, MMX, MMX logo, Optimizer logo, OverDrive, Paragon, PDCharm, Pentium, Pentium II Xeon, Pentium III Xeon, Performance at Your Command, Sound Mark, The Computer Inside., The Journey Inside, VTune, and Xircom are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

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

Copyright © Intel Corporation 2004

# Contents

**1 Introduction** ..... 23
1.1 About the Intel® IXP42X Product Line of Network Processors and IXC1100 Control Plane Processor ..... 23
1.2 Intended Audience ..... 23
1.3 Related Documents ..... 24
1.4 Acronyms ..... 24

**2 Software Architecture Overview** ..... 29
2.1 High-Level Overview ..... 29
2.2 Deliverable Model ..... 30
2.3 Operating System Support ..... 30
2.4 Development Tools ..... 30
2.5 Threading and Locking Policy ..... 31
2.6 Polled and Interrupt Operation ..... 31
2.7 Statistics and MIBs ..... 31
2.8 Network Data Buffer Management ..... 31

**3 Software Development Tools** ..... 33
3.1 Intel Hardware ..... 33
3.1.1 Intel® IXDP425 Network Processor Development Platform ..... 33
3.2 Third-Party Software and Hardware ..... 34
3.2.1 Software Stacks for the Intel® IXP42X Product Line Processors ..... 34
3.2.2 Hardware and Software Tools for the Intel® IXP42X Product Line Processors ..... 35
3.2.3 Software for the Intel® IXDP425 Network Processor Development Platform ..... 36
3.2.4 Open Source Software and Tools ..... 38

**4 Access-Layer Components** ..... 39
4.1 ixNpeDl ..... 39
4.1.1 Functional Interface Description ..... 39
4.1.2 Microcode Images ..... 39
4.1.3 Usage Example ..... 40
4.2 ixFpathAcc (Fast Path) ..... 40
4.2.1 Scope ..... 40
4.2.2 Functional Description ..... 41
4.2.3 Enabling/Modifying/Disabling Fast Path for a VC ..... 42
4.2.4 Automatic Packet Operations ..... 42
4.2.5 Application Support and MAC Address Service ..... 42
4.2.6 Fast-Path Buffers ..... 43
4.2.7 Fast-Path Dependency Diagram ..... 43
4.2.8 Error Handling ..... 43
4.2.9 NPE FPath Classifier/Modifier Configuration ..... 44
4.2.9.1 Fast-Path Classifier Template Definition ..... 44
4.2.9.2 Fast-Path Modifier Template Definition ..... 45
4.2.9.3 Sample Classifier Template ..... 46
4.3 ixEthAcc ..... 47

4.3.1 ixEthAcc Overview ..... 47
4.3.2 Role of the Ethernet NPE Firmware ..... 48
4.3.3 ixEthAcc Component Features ..... 48
4.3.4 Data Plane ..... 50
4.3.4.1 Port Initialization ..... 50
4.3.4.2 Ethernet Frame Transmission ..... 50
4.3.4.3 Ethernet Frame Reception ..... 52
4.3.4.4 Data-Plane Endianness ..... 55
4.3.4.5 External Memory Requirements ..... 55
4.3.5 Control Plane ..... 57
4.3.5.1 MAC Address Learning Database Features ..... 57
4.3.5.2 Ethernet MAC Control ..... 62
4.3.6 Management Information ..... 63
4.3.7 Memory Requirements ..... 64
4.3.7.1 Code Size ..... 64
4.3.7.2 Data Memory ..... 64
4.4 IxAtmdAcc ..... 65
4.4.1 IxAtmdAcc Component Features ..... 66
4.4.2 Configuration Services ..... 67
4.4.2.1 UTOPIA Port Configuration Service ..... 67
4.4.2.2 ATM Traffic Shaping Services ..... 67
4.4.2.3 VC Configuration Services ..... 68
4.4.3 Transmission Services ..... 69
4.4.3.1 Scheduled Transmission ..... 70
4.4.3.2 Transmission Triggers (Tx Low Notification) ..... 72
4.4.3.3 Receive Services ..... 76
4.4.3.4 Buffer Management ..... 79
4.4.3.5 Error Handling ..... 81
4.4.3.6 Memory Requirements ..... 82
4.4.3.7 Performance ..... 82
4.5 IxAtmSch ..... 83
4.5.1 IxAtmSch Overview ..... 83
4.5.2 IxAtmSch Component Features ..... 84
4.5.3 Connection Admission Control (CAC) Function ..... 85
4.5.4 Scheduling and Traffic Shaping ..... 86
4.5.4.1 Schedule Table ..... 86
4.5.4.2 Schedule Service Model ..... 87
4.5.4.3 Timing and Idle Cells ..... 88
4.5.5 Dependencies ..... 88
4.5.6 Error Handling ..... 88
4.5.7 Memory Requirements ..... 89
4.5.7.1 Code Size ..... 89
4.5.7.2 Data Memory ..... 89
4.5.8 Performance ..... 89
4.5.8.1 Latency ..... 89
4.6 ixAtmm ..... 89
4.6.1 IxAtmm Overview ..... 90
4.6.2 IxAtmm Component Features ..... 90
4.6.3 UTOPIA Level-2 Port Initialization ..... 91
4.6.4 ATM-Port Management Service Model ..... 91
4.6.5 Tx/Rx Control Configuration ..... 93

4.6.6 Dependencies ..... 95
4.6.7 Error Handling ..... 95
4.6.8 Management Interfaces ..... 95
4.6.9 Memory Requirements ..... 95
4.6.10 Performance ..... 95
4.7 ixHSSAcc ..... 96
4.7.1 Overview ..... 96
4.7.2 Features ..... 97
4.7.3 Functional Interface Description ..... 98
4.7.4 Key Assumptions ..... 98
4.7.5 Data-Flow Overview ..... 98
4.7.6 HSS Port Configuration ..... 103
4.7.6.1 Description ..... 103
4.7.6.2 Pre-Conditions ..... 103
4.7.7 HSS Channelised Operation ..... 104
4.7.7.1 Channelised Connect ..... 104
4.7.7.2 Channelised Tx/Rx ..... 105
4.7.7.3 Channelised Disconnect ..... 106
4.7.8 HSS Packetised Operation ..... 107
4.7.8.1 Packetised Connect ..... 107
4.7.8.2 Packetised Tx ..... 108
4.7.8.3 Packetised Rx ..... 109
4.7.8.4 Packetised Disconnect ..... 110
4.8 ixNpeMh ..... 111
4.8.1 Sending Messages from an Intel XScale® Core Software Client to an NPE ..... 112
4.8.1.1 Sending an NPE Message ..... 112
4.8.1.2 Sending an NPE Message with Response ..... 112
4.8.2 Receiving Unsolicited Messages from an NPE to Client Software ..... 114
4.8.3 Dependencies ..... 115
4.8.4 Error-Handling ..... 115
4.9 ixQmgr ..... 116
4.9.1 Queue Configuration ..... 117
4.9.2 Queue Identifiers ..... 118
4.9.3 Configuration Values ..... 118
4.9.4 Dispatcher ..... 118
4.9.5 Threading ..... 121
4.9.6 Dependencies ..... 121
4.9.7 Error Handling ..... 121
4.10 ixUsb ..... 122
4.10.1 Overview ..... 122
4.10.2 API Interfaces ..... 122
4.10.3 USB Operation ..... 123
4.10.4 Packet Formats ..... 124
4.10.5 Transaction Formats ..... 125
4.10.6 UDC Device Requests ..... 127
4.10.7 Configuration ..... 129
4.10.8 Endpoint Stall Feature ..... 129
4.10.9 Frame Synchronization ..... 130
4.10.10 SNMP ..... 130
4.10.11 IxUSBManagerInfo ..... 131
4.10.11.1 IxUSBManagerTraps ..... 132

4.10.11.2 SNMP Interface Functions ..... 132
4.10.12 Error Handling ..... 133
4.10.12.1 USB Data Flow ..... 134
4.10.12.2 Dependencies and Data Flows ..... 134
4.11 ixUart ..... 134
4.11.1 Interface Description ..... 135
4.11.2 UART Data Flow ..... 135
4.11.2.1 FIFO Versus Polled Mode ..... 135
4.11.3 Dependencies ..... 136
4.12 ixCryptoAcc ..... 137
4.12.1 Overview ..... 137
4.12.1.1 Authentication Mode ..... 138
4.12.1.2 Security Association Database Management ..... 140
4.12.1.3 Buffer Management ..... 140
4.12.1.4 Error Handling ..... 140
4.12.1.5 Endianness ..... 140
4.12.1.6 Import and Export Regulations of Cryptographic Technology ..... 140
4.12.1.7 NPE-based Security Hardware Accelerator and IxCryptoAcc Scope ..... 140
4.12.1.8 IxCryptoAcc Component Architecture ..... 142
4.12.2 Functional Interface ..... 144
4.12.2.1 Overview ..... 144
4.12.2.2 API ..... 145
4.12.2.3 Basic API Flow ..... 146
4.12.2.4 CCD Update Flow ..... 147
4.12.2.5 Assumptions, Restrictions, and Dependencies ..... 148
4.12.2.6 Dependencies ..... 149
4.12.2.7 Issues ..... 149
4.12.3 Use Models ..... 150
4.12.3.1 IxCryptoAcc Data Flow Overview ..... 150
4.12.3.2 Reference ESP Dataflow ..... 152
4.12.3.3 Reference AH Dataflow ..... 153
4.12.4 Memory Requirements ..... 154
4.12.4.1 Estimated Code Memory ..... 154
4.12.4.2 Data Memory ..... 154
4.12.5 Performance Estimation ..... 155
4.12.6 Overview of NPE Support for Security ..... 156
4.12.6.1 NPE Firmware Versioning ..... 156
4.12.6.2 Supported Encryption and Authentication Algorithms ..... 156
4.12.6.3 Cipher Modes ..... 157
4.12.6.4 Electronic Code Book (ECB) ..... 157
4.12.6.5 Cipher Block Chaining (CBC) ..... 157
4.12.6.6 Counter Mode (CTR) ..... 158
4.12.6.7 Detailed Cryptographic NPE Data Flow ..... 158
4.12.7 Security Hardware Accelerator Demonstration ..... 160
4.13 ixDMAAcc ..... 160
4.13.1 Overview ..... 160
4.13.2 Features ..... 160
4.13.3 Assumptions ..... 160
4.13.4 Dependencies ..... 161
4.13.5 DMA Access Layer API ..... 162
4.13.5.1 IxDmaAccDescriptorManager ..... 163
4.13.6 Parameters Description ..... 163

4.13.6.1 Source Address ..... 163
4.13.6.2 Destination Address ..... 164
4.13.6.3 Transfer Mode ..... 164
4.13.6.4 Transfer Width ..... 164
4.13.6.5 Addressing Modes ..... 164
4.13.6.6 Transfer Length ..... 165
4.13.6.7 Supported Modes ..... 165
4.13.7 Data Flow ..... 167
4.13.8 Control Flow ..... 168
4.13.8.1 DMA Initialization ..... 169
4.13.8.2 DMA Configuration and Data Transfer ..... 170
4.13.9 Restrictions of the DMA Transfer ..... 171
4.13.10 Error Handling ..... 172

**5 Operating System Abstraction Layers ..... 173**

5.1 Overview ..... 173
5.2 ixOsServices ..... 174
5.2.1 Mutual Exclusion Service ..... 174
5.2.2 Trace Service ..... 174
5.2.3 Timer Service ..... 175
5.2.4 ixOsServices Functions ..... 175
5.3 OSSL ..... 176
5.3.1 Thread Management ..... 176
5.3.2 Semaphores ..... 176
5.3.3 Mutual Exclusion ..... 176
5.3.4 Semaphores versus Mutexes ..... 177
5.3.5 Timers ..... 177
5.3.6 Memory Management ..... 177
5.3.7 Message Logging ..... 177
5.3.8 OSSL Functions ..... 178
5.4 Mapping Software Components to Operating System Services ..... 179

**6 System Configuration ..... 181**

6.1 PCI Configuration ..... 181
6.1.1 Initialization APIs ..... 184
6.1.1.1 sysPciInit ..... 184
6.1.2 Address-Mapping APIs ..... 185
6.1.2.1 Overview ..... 185
6.1.3 Mapping APIs ..... 190
6.1.3.1 SysPciMappingAdd ..... 190
6.1.3.2 sysPciToPhys ..... 191
6.1.3.3 sysPhysToPci ..... 191
6.1.4 Miscellaneous PCI-Support APIs ..... 192
6.1.4.1 sysPciAssignAddress ..... 192
6.1.4.2 pciDevConfig ..... 192
6.1.5 PCI Resources-Identification APIs ..... 193
6.1.5.1 pciInfoGet ..... 193
6.1.5.2 pciInfoSearch ..... 194
6.1.5.3 pciDeviceGet ..... 194
6.1.5.4 pciFindDevice ..... 195
6.1.5.5 pciFindClass ..... 195
6.1.5.6 pciAddrShow ..... 196

6.1.5.7 pciDeviceShow ..... 196
6.1.5.8 pciHeaderShow ..... 197
6.1.5.9 pciFindDeviceShow ..... 197
6.1.5.10 pciFindClassShow ..... 198
6.1.6 PCI Configuration-Access APIs ..... 198
6.1.6.1 pciConfigInByte ..... 198
6.1.6.2 pciConfigInWord ..... 199
6.1.6.3 pciConfigInLong ..... 199
6.1.6.4 pciConfigOutByte ..... 200
6.1.6.5 pciConfigOutWord ..... 200
6.1.6.6 pciConfigOutLong ..... 201
6.1.7 PCI Special-Cycle-Generation API ..... 202
6.1.7.1 pciSpecialCycle ..... 202
6.1.8 PCI DMA-Access APIs ..... 202
6.1.8.1 pciDmaTransfer ..... 202
6.1.8.2 pciDmaMbufTransfer ..... 203
6.1.8.3 pciDmaShow ..... 204
6.1.9 PCI Interrupts APIs ..... 204
6.1.9.1 pciIntConnect ..... 204
6.1.9.2 pciIntDisconnect ..... 205
6.1.9.3 pciInt ..... 205
6.1.10 PCI Data-Access APIs ..... 206
6.1.10.1 pciIOOutByte ..... 206
6.1.10.2 pciIOOutWord ..... 206
6.1.10.3 pciIOOutLong ..... 207
6.1.10.4 pciIOInByte ..... 207
6.1.10.5 pciIOInWord ..... 208
6.1.10.6 pciIOInLong ..... 208
6.1.10.7 pciMemOutByte ..... 208
6.1.10.8 pciMemOutWord ..... 209
6.1.10.9 pciMemOutLong ..... 209
6.1.10.10 pciMemInByte ..... 210
6.1.10.11 pciMemInWord ..... 210
6.1.10.12 pciMemInLong ..... 210
6.1.11 PCI Error-Support APIs ..... 211
6.1.11.1 pciErrorCallbackRegister ..... 211
6.2 ADSL on the Intel® IXDP425 / IXCDP1100 Development Platform ..... 212
6.2.1 Functionality Overview ..... 212
6.2.1.1 Controlling Alcatel* ADSL Modem Chipset Through Parallel CTRL-E ..... 212
6.2.2 Interface ..... 213
6.2.3 Management Interface Functions ..... 214
6.2.3.1 Supported Management Information Attributes ..... 214
6.2.3.2 Interfaces Table Support ..... 214
6.2.3.3 IfTable Specific Traps ..... 216
6.2.3.4 ADSL Line Table ..... 217
6.2.3.5 ADSL ATU-C Physical Table ..... 218
6.2.3.6 ADSL ATU-R Physical Table ..... 219
6.2.3.7 ADSL ATU-R Channel Table ..... 220
6.2.3.8 ADSL ATU-R Performance Data Table ..... 220
6.2.4 Modem Self Test ..... 221
6.2.5 Debug ..... 221
6.2.6 ADSL MIB Compliance ..... 221
6.2.6.1 Mandatory Groups for ATU-C Agent Compliance ..... 221

6.2.6.2 Mandatory Groups for ATU-R Agent Compliance ..... 224
6.2.6.3 ADSL MIB Trap Notification Support ..... 227

**A Application Programming Interfaces** ..... 229

A.1 Intel® IXP42X product line Access-Layer API and Codelet Module Index ..... 229
A.2 Intel® IXP42X product line Access-Layer API and Codelet Data Structure Index ..... 231
A.3 Intel® IXP42X product line Access-Layer APIs ..... 233
A.3.1 ADSL Driver (IxAdsl) API ..... 233
A.3.1.1 Defines ..... 233
A.3.1.2 Typedefs ..... 234
A.3.1.3 Functions ..... 234
A.3.1.4 Detailed Description ..... 235
A.3.1.5 Typedef Documentation ..... 235
A.3.1.6 Function Documentation ..... 236
A.3.2 Assertion Macros (IxAssert) API ..... 245
A.3.2.1 Defines ..... 245
A.3.2.2 Detailed Description ..... 245
A.3.3 ATM Driver Access (IxAtmdAcc) API ..... 245
A.3.3.1 Defines ..... 245
A.3.3.2 Typedefs ..... 246
A.3.3.3 Enumerations ..... 246
A.3.3.4 Functions ..... 247
A.3.3.5 Detailed Description ..... 248
A.3.3.6 Define Documentation ..... 248
A.3.3.7 Typedef Documentation ..... 249
A.3.3.8 Enumeration Type Documentation ..... 251
A.3.3.9 Function Documentation ..... 252
A.3.4 ATM Driver Access (IxAtmdAcc) Control API ..... 264
A.3.4.1 Modules ..... 264
A.3.4.2 Defines ..... 264
A.3.4.3 Typedefs ..... 264
A.3.4.4 Functions ..... 265
A.3.4.5 Detailed Description ..... 266
A.3.4.6 Define Documentation ..... 266
A.3.4.7 Typedef Documentation ..... 266
A.3.4.8 Function Documentation ..... 270
A.3.5 ATM Driver Access (IxAtmdAcc) UTOPIA Control API ..... 282
A.3.5.1 Data Structures ..... 282
A.3.5.2 Detailed Description ..... 283
A.3.6 ATM Manager (IxAtmm) API ..... 284
A.3.6.1 Data Structures ..... 284
A.3.6.2 Defines ..... 284
A.3.6.3 Typedefs ..... 284
A.3.6.4 Enumerations ..... 285
A.3.6.5 Functions ..... 285
A.3.6.6 Detailed Description ..... 286
A.3.6.7 Typedef Documentation ..... 286
A.3.6.8 Function Documentation ..... 287
A.3.7 ATM Transmit Scheduler (IxAtmSch) API ..... 295
A.3.7.1 Defines ..... 295
A.3.7.2 Functions ..... 295
A.3.7.3 Detailed Description ..... 296
A.3.7.4 Function Documentation ..... 296

A.3.8 ATM Types (IxAtmTypes) ..... 302
A.3.8.1 Data Structures ..... 302
A.3.8.2 Defines ..... 302
A.3.8.3 Typedefs ..... 303
A.3.8.4 Enumerations ..... 304
A.3.8.5 Detailed Description ..... 304
A.3.8.6 Define Documentation ..... 304
A.3.8.7 Typedef Documentation ..... 304
A.3.8.8 Enumeration Type Documentation ..... 305
A.3.9 Security (IxCryptoAcc) API ..... 306
A.3.9.1 Data Structures ..... 306
A.3.9.2 Defines ..... 306
A.3.9.3 Typedefs ..... 307
A.3.9.4 Enumerations ..... 307
A.3.9.5 Functions ..... 308
A.3.9.6 Detailed Description ..... 309
A.3.9.7 Define Documentation ..... 309
A.3.9.8 Typedef Documentation ..... 309
A.3.9.9 Enumeration Type Documentation ..... 311
A.3.9.10 Function Documentation ..... 311
A.3.10 DMA Types (IxDmaTypes) ..... 316
A.3.10.1 Enumerations ..... 316
A.3.10.2 Detailed Description ..... 316
A.3.10.3 Enumeration Type Documentation ..... 317
A.3.11 DMA Access Driver (IxDmaAcc) API ..... 319
A.3.11.1 Defines ..... 319
A.3.11.2 Typedefs ..... 319
A.3.11.3 Functions ..... 319
A.3.11.4 Detailed Description ..... 319
A.3.11.5 Define Documentation ..... 319
A.3.11.6 Typedef Documentation ..... 320
A.3.11.7 Function Documentation ..... 320
A.3.12 Ethernet Access (IxEthAcc) API ..... 322
A.3.12.1 Data Structures ..... 322
A.3.12.2 Defines ..... 322
A.3.12.3 Typedefs ..... 322
A.3.12.4 Enumerations ..... 323
A.3.12.5 Functions ..... 323
A.3.12.6 Detailed Description ..... 325
A.3.12.7 Define Documentation ..... 325
A.3.12.8 Typedef Documentation ..... 326
A.3.12.9 Enumeration Type Documentation ..... 327
A.3.12.10 Function Documentation ..... 328
A.3.13 Ethernet Access Fast Path (IxEthAccFpathDep) API ..... 346
A.3.13.1 Defines ..... 346
A.3.13.2 Detailed Description ..... 346
A.3.13.3 Define Documentation ..... 346
A.3.14 Ethernet Database (IxEthDB) API ..... 347
A.3.14.1 Data Structures ..... 347
A.3.14.2 Defines ..... 347
A.3.14.3 Typedefs ..... 347
A.3.14.4 Enumerations ..... 347
A.3.14.5 Functions ..... 348

A.3.14.6 Detailed Description ..... 349
A.3.14.7 Define Documentation ..... 349
A.3.14.8 Enumeration Type Documentation ..... 349
A.3.14.9 Function Documentation ..... 349
A.3.15 Ethernet Database Port Definitions (IxEthDBPortDefs) ..... 356
A.3.15.1 Data Structures ..... 356
A.3.15.2 Defines ..... 356
A.3.15.3 Enumerations ..... 356
A.3.15.4 Detailed Description ..... 356
A.3.15.5 Define Documentation ..... 357
A.3.15.6 Enumeration Type Documentation ..... 357
A.3.16 Ethernet NPE (IxEthNpe) API ..... 357
A.3.16.1 Defines ..... 357
A.3.16.2 Detailed Description ..... 361
A.3.16.3 Define Documentation ..... 362
A.3.17 Fast Path Access (IxFpathAcc) API ..... 367
A.3.17.1 Defines ..... 367
A.3.17.2 Typedefs ..... 368
A.3.17.3 Enumerations ..... 368
A.3.17.4 Functions ..... 368
A.3.17.5 Detailed Description ..... 369
A.3.17.6 Define Documentation ..... 369
A.3.17.7 Typedef Documentation ..... 370
A.3.17.8 Enumeration Type Documentation ..... 371
A.3.17.9 Function Documentation ..... 371
A.3.18 HSS Access (IxHssAcc) API ..... 377
A.3.18.1 Data Structures ..... 377
A.3.18.2 Defines ..... 377
A.3.18.3 Typedefs ..... 378
A.3.18.4 Enumerations ..... 378
A.3.18.5 Functions ..... 380
A.3.18.6 Detailed Description ..... 382
A.3.18.7 Typedef Documentation ..... 382
A.3.18.8 Enumeration Type Documentation ..... 384
A.3.18.9 Function Documentation ..... 389
A.3.19 NPE-A (IxNpeA) API ..... 398
A.3.19.1 Data Structures ..... 398
A.3.19.2 Defines ..... 398
A.3.19.3 Enumerations ..... 405
A.3.19.4 Detailed Description ..... 405
A.3.19.5 Define Documentation ..... 405
A.3.20 NPE-Downloader (IxNpeDl) API ..... 408
A.3.20.1 Data Structures ..... 408
A.3.20.2 Defines ..... 408
A.3.20.3 Typedefs ..... 409
A.3.20.4 Enumerations ..... 409
A.3.20.5 Functions ..... 409
A.3.20.6 Detailed Description ..... 410
A.3.20.7 Enumeration Type Documentation ..... 410
A.3.20.8 Function Documentation ..... 410
A.3.21 NPE Message Handler (IxNpeMh) API ..... 415
A.3.21.1 Data Structures ..... 415
A.3.21.2 Defines ..... 415

A.3.21.3 Typedefs ..... 415
A.3.21.4 Enumerations ..... 416
A.3.21.5 Functions ..... 416
A.3.21.6 Detailed Description ..... 417
A.3.21.7 Typedef Documentation ..... 417
A.3.21.8 Enumeration Type Documentation ..... 417
A.3.21.9 Function Documentation ..... 418
A.3.22 OS Memory Buffer Management (IxOsBuffMgt) API ..... 423
A.3.22.1 Defines ..... 423
A.3.22.2 Detailed Description ..... 423
A.3.23 OS Memory Buffer Pool Management (IxOsBuffPoolMgt) API ..... 423
A.3.23.1 Data Structures ..... 423
A.3.23.2 Defines ..... 423
A.3.23.3 Typedefs ..... 425
A.3.23.4 Functions ..... 425
A.3.23.5 Detailed Description ..... 425
A.3.23.6 Define Documentation ..... 425
A.3.23.7 Function Documentation ..... 428
A.3.24 OS Services (IxOsServices) API ..... 432
A.3.24.1 Typedefs ..... 432
A.3.24.2 Enumerations ..... 433
A.3.24.3 Functions ..... 433
A.3.24.4 Detailed Description ..... 434
A.3.24.5 Function Documentation ..... 434
A.3.25 Queue Manager (IxQMgr) API ..... 440
A.3.25.1 Defines ..... 440
A.3.25.2 Typedefs ..... 441
A.3.25.3 Enumerations ..... 441
A.3.25.4 Functions ..... 442
A.3.25.5 Detailed Description ..... 444
A.3.25.6 Define Documentation ..... 444
A.3.25.7 Typedef Documentation ..... 446
A.3.25.8 Enumeration Type Documentation ..... 447
A.3.25.9 Function Documentation ..... 449
A.3.26 Timer Control (IxTimerCtrl) API ..... 460
A.3.26.1 Defines ..... 460
A.3.26.2 Typedefs ..... 460
A.3.26.3 Enumerations ..... 461
A.3.26.4 Functions ..... 461
A.3.26.5 Detailed Description ..... 461
A.3.26.6 Define Documentation ..... 461
A.3.26.7 Typedef Documentation ..... 461
A.3.26.8 Enumeration Type Documentation ..... 462
A.3.26.9 Function Documentation ..... 462
A.3.27 Types (IxTypes) ..... 464
A.3.27.1 Defines ..... 464
A.3.27.2 Typedefs ..... 465
A.3.27.3 Detailed Description ..... 465
A.3.28 UART Access (IxUARTAcc) API ..... 465
A.3.28.1 Data Structures ..... 465
A.3.28.2 Defines ..... 466
A.3.28.3 Typedefs ..... 467
A.3.28.4 Enumerations ..... 467

A.3.28.5 Functions ..... 467
A.3.28.6 Detailed Description ..... 467
A.3.28.7 Enumeration Type Documentation ..... 467
A.3.28.8 Function Documentation ..... 468
A.3.29 Version ID (IxVersionId) ..... 469
A.3.29.1 Defines ..... 469
A.3.29.2 Detailed Description ..... 469
A.3.29.3 Define Documentation ..... 469
A.3.30 USB Access (IxUsb) API ..... 469
A.3.30.1 Functions ..... 469
A.3.30.2 Detailed Description ..... 470
A.3.30.3 Function Documentation ..... 470
A.4 Intel® IXP400 SoftwareCodelets ..... 475
A.4.1 Codelet Overview ..... 476
A.4.1.1 Modules ..... 476
A.4.1.2 Detailed Description ..... 476
A.4.2 AAL5 Main Codelet (IxAal5CodeletMain) API ..... 476
A.4.2.1 Functions ..... 477
A.4.2.2 Detailed Description ..... 477
A.4.3 Aal5 HSS Codelet (IxAal5HssCodelet) API ..... 477
A.4.3.1 Defines ..... 477
A.4.3.2 Functions ..... 477
A.4.3.3 Detailed Description ..... 477
A.4.3.4 Define Documentation ..... 478
A.4.3.5 Function Documentation ..... 478
A.4.4 Atm Codelet (IxAtmCodelet) API ..... 478
A.4.4.1 Enumerations ..... 478
A.4.4.2 Functions ..... 478
A.4.4.3 Detailed Description ..... 479
A.4.4.4 Enumeration Type Documentation ..... 481
A.4.4.5 Function Documentation ..... 482
A.4.5 Crypto Access Codelet (IxCryptoAccCodelet) API ..... 485
A.4.5.1 Defines ..... 485
A.4.5.2 Functions ..... 485
A.4.5.3 Detailed Description ..... 486
A.4.5.4 Define Documentation ..... 487
A.4.5.5 Function Documentation ..... 487
A.4.6 Demo Utils (IxDemoUtils) API ..... 491
A.4.6.1 Functions ..... 491
A.4.6.2 Detailed Description ..... 491
A.4.6.3 Function Documentation ..... 492
A.4.7 DMA Access Codelet (IxAal5Demo) API ..... 496
A.4.7.1 Functions ..... 496
A.4.7.2 Detailed Description ..... 496
A.4.7.3 Function Documentation ..... 496
A.4.8 Ethernet Aal5 (IxEthAal5App) API ..... 498
A.4.8.1 Defines ..... 498
A.4.8.2 Functions ..... 499
A.4.8.3 Detailed Description ..... 499
A.4.8.4 Define Documentation ..... 499
A.4.8.5 Function Documentation ..... 499
A.4.9 Ethernet Access Codelet (IxEthAccCodelet) API ..... 500

A.4.9.1 Data Structures ..... 500
A.4.9.2 Defines ..... 500
A.4.9.3 Functions ..... 501
A.4.9.4 Detailed Description ..... 501
A.4.9.5 Define Documentation ..... 502
A.4.9.6 Function Documentation ..... 503
A.4.10 Fast Path Access Codelet (IxFpathAccCodelet) API ..... 504
A.4.10.1 Functions ..... 504
A.4.10.2 Detailed Description ..... 505
A.4.10.3 Function Documentation ..... 508
A.4.11 HSS Access Codelet (IxHssAccCodelet) API ..... 513
A.4.11.1 Modules ..... 513
A.4.11.2 Functions ..... 513
A.4.11.3 Detailed Description ..... 514
A.4.11.4 Function Documentation ..... 516
A.4.12 HSS Access Codelet Channelised (IxHssAccCodeletChan) ..... 522
A.4.12.1 Defines ..... 523
A.4.12.2 Detailed Description ..... 523
A.4.12.3 Define Documentation ..... 523
A.4.13 HSS Access Codelet Common (IxHssAccCodeletCom) API ..... 523
A.4.13.1 Data Structures ..... 523
A.4.13.2 Functions ..... 524
A.4.13.3 Variables ..... 524
A.4.13.4 Detailed Description ..... 524
A.4.13.5 Function Documentation ..... 524
A.4.13.6 Variable Documentation ..... 525
A.4.14 HSS Access Codelet Config (IxHssAccCodeletConfig) API ..... 525
A.4.14.1 Functions ..... 525
A.4.14.2 Detailed Description ..... 525
A.4.14.3 Function Documentation ..... 525
A.4.15 HSS Access Codelet MBuf Pool (IsHssAccCodeletMbuf) API ..... 526
A.4.15.1 Defines ..... 526
A.4.15.2 Typedefs ..... 526
A.4.15.3 Functions ..... 526
A.4.15.4 Detailed Description ..... 526
A.4.15.5 Function Documentation ..... 526
A.4.16 HSS Access Codelet Packetised (IxHssAccCodeletPkt) API ..... 527
A.4.16.1 Defines ..... 527
A.4.16.2 Detailed Description ..... 528
A.4.16.3 Define Documentation ..... 528
A.4.17 Oam Codelet (IxOamCodelet) API ..... 528
A.4.17.1 Functions ..... 528
A.4.17.2 Detailed Description ..... 529
A.4.17.3 Function Documentation ..... 529
A.4.18 UART Access Codelet (IxUARTAccCodelet) API ..... 533
A.4.18.1 Defines ..... 533
A.4.18.2 Typedefs ..... 533
A.4.18.3 Enumerations ..... 533
A.4.18.4 Functions ..... 533
A.4.18.5 Detailed Description ..... 533
A.4.18.6 Function Documentation ..... 534
A.4.19 USB RNDIS Codelet (IxUSBRNDIS) API ..... 536
A.4.19.1 Modules ..... 536

A.4.19.2 Data Structures ..... 536
A.4.19.3 Defines ..... 536
A.4.19.4 Typedefs ..... 540
A.4.19.5 Enumerations ..... 540
A.4.19.6 Functions ..... 541
A.4.19.7 Detailed Description ..... 541
A.4.20 USB RNDIS End Driver Codelet (IxUSBRNDIS) API ..... 542
A.4.20.1 Defines ..... 542
A.4.20.2 Functions ..... 542
A.4.20.3 Detailed Description ..... 542
A.4.21 USB RNDIS Vendor Codelet (IxUSBRNDIS) ..... 542
A.4.21.1 Defines ..... 542
A.4.21.2 Detailed Description ..... 542
A.5 Intel® IXP42X product line Access-Layer API and Codelet Data Structure Documentation ..... 543
A.5.1 Aal5Statistics Struct Reference ..... 543
A.5.1.1 Data Fields ..... 543
A.5.1.2 Detailed Description ..... 543
A.5.2 ActiveIpFlow Struct Reference ..... 543
A.5.2.1 Data Fields ..... 543
A.5.2.2 Detailed Description ..... 543
A.5.3 ChannelisedStats Struct Reference ..... 543
A.5.3.1 Data Fields ..... 544
A.5.3.2 Detailed Description ..... 544
A.5.4 GeneralStats Struct Reference ..... 544
A.5.4.1 Data Fields ..... 544
A.5.4.2 Detailed Description ..... 544
A.5.5 ix_s_ossl_thread_main_info_t Struct Reference ..... 545
A.5.5.1 Data Fields ..... 545
A.5.5.2 Detailed Description ..... 545
A.5.5.3 Description ..... 545
A.5.6 ix_s_ossl_time_t Struct Reference ..... 545
A.5.6.1 Data Fields ..... 545
A.5.6.2 Detailed Description ..... 545
A.5.7 IxAtmdAccUtopiaConfig Struct Reference ..... 545
A.5.7.1 Data Fields ..... 546
A.5.7.2 Detailed Description ..... 547
A.5.8 IxAtmdAccUtopiaConfig::UtRxConfig_ Struct Reference ..... 547
A.5.8.1 Data Fields ..... 547
A.5.8.2 Detailed Description ..... 548
A.5.8.3 Field Documentation ..... 548
A.5.9 IxAtmdAccUtopiaConfig::UtRxDefineIdle_ Struct Reference ..... 550
A.5.9.1 Data Fields ..... 550
A.5.9.2 Detailed Description ..... 550
A.5.10 IxAtmdAccUtopiaConfig::UtRxEnableFields_ Struct Reference ..... 551
A.5.10.1 Data Fields ..... 551
A.5.10.2 Detailed Description ..... 552
A.5.10.3 Field Documentation ..... 552
A.5.11 IxAtmdAccUtopiaConfig::UtRxStatsConfig_ Struct Reference ..... 555
A.5.11.1 Data Fields ..... 555
A.5.11.2 Detailed Description ..... 556
A.5.11.3 Field Documentation ..... 556

A.5.12 IxAtmdAccUtopiaConfig::UtRxTransTable0_ Struct Reference ..... 556
A.5.12.1 Data Fields ..... 556
A.5.12.2 Detailed Description ..... 557
A.5.13 IxAtmdAccUtopiaConfig::UtRxTransTable1_ Struct Reference ..... 557
A.5.13.1 Data Fields ..... 557
A.5.13.2 Detailed Description ..... 558
A.5.14 IxAtmdAccUtopiaConfig::UtRxTransTable2_ Struct Reference ..... 558
A.5.14.1 Data Fields ..... 558
A.5.14.2 Detailed Description ..... 558
A.5.15 IxAtmdAccUtopiaConfig::UtRxTransTable3_ Struct Reference ..... 558
A.5.15.1 Data Fields ..... 558
A.5.15.2 Detailed Description ..... 559
A.5.16 IxAtmdAccUtopiaConfig::UtRxTransTable4_ Struct Reference ..... 559
A.5.16.1 Data Fields ..... 559
A.5.16.2 Detailed Description ..... 560
A.5.17 IxAtmdAccUtopiaConfig::UtRxTransTable5_ Struct Reference ..... 560
A.5.17.1 Data Fields ..... 560
A.5.17.2 Detailed Description ..... 560
A.5.18 IxAtmdAccUtopiaConfig::UtSysConfig_ Struct Reference ..... 560
A.5.18.1 Data Fields ..... 560
A.5.18.2 Detailed Description ..... 561
A.5.18.3 Field Documentation ..... 561
A.5.19 IxAtmdAccUtopiaConfig::UtTxConfig_ Struct Reference ..... 562
A.5.19.1 Data Fields ..... 562
A.5.19.2 Detailed Description ..... 563
A.5.19.3 Field Documentation ..... 563
A.5.20 IxAtmdAccUtopiaConfig::UtTxDefineIdle_ Struct Reference ..... 564
A.5.20.1 Data Fields ..... 564
A.5.20.2 Detailed Description ..... 565
A.5.21 IxAtmdAccUtopiaConfig::UtTxEnableFields_ Struct Reference ..... 565
A.5.21.1 Data Fields ..... 565
A.5.21.2 Detailed Description ..... 566
A.5.21.3 Field Documentation ..... 566
A.5.21.4 IxAtmdAccUtopiaConfig::UtTxStatsConfig_ Struct Reference ..... 568
A.5.21.5 Data Fields ..... 568
A.5.21.6 Detailed Description ..... 568
A.5.21.7 Field Documentation ..... 568
A.5.22 IxAtmdAccUtopiaConfig::UtTxTransTable0_ Struct Reference ..... 569
A.5.22.1 Data Fields ..... 569
A.5.22.2 Detailed Description ..... 569
A.5.23 IxAtmdAccUtopiaConfig::UtTxTransTable1_ Struct Reference ..... 569
A.5.23.1 Data Fields ..... 569
A.5.23.2 Detailed Description ..... 570
A.5.24 IxAtmdAccUtopiaConfig::UtTxTransTable2_ Struct Reference ..... 570
A.5.24.1 Data Fields ..... 570
A.5.24.2 Detailed Description ..... 571
A.5.25 IxAtmdAccUtopiaConfig::UtTxTransTable3_ Struct Reference ..... 571
A.5.25.1 Data Fields ..... 571
A.5.25.2 Detailed Description ..... 571
A.5.26 IxAtmdAccUtopiaConfig::UtTxTransTable4_ Struct Reference ..... 571
A.5.26.1 Data Fields ..... 571
A.5.26.2 Detailed Description ..... 572

A.5.27 IxAtmdAccUtopiaConfig::UtTxTransTable5_ Struct Reference ..... 572
A.5.27.1 Data Fields ..... 572
A.5.27.2 Detailed Description ..... 572
A.5.28 IxAtmdAccUtopiaStatus Struct Reference ..... 572
A.5.28.1 Data Fields ..... 573
A.5.28.2 Detailed Description ..... 573
A.5.29 IxAtmdAccUtopiaStatus::UtRxCellConditionStatus_ Struct Reference ..... 573
A.5.29.1 Data Fields ..... 573
A.5.29.2 Detailed Description ..... 574
A.5.30 IxAtmdAccUtopiaStatus::UtTxCellConditionStatus_ Struct Reference ..... 574
A.5.30.1 Data Fields ..... 574
A.5.30.2 Detailed Description ..... 575
A.5.31 IxAtmmPortCfg Struct Reference ..... 575
A.5.31.1 Data Fields ..... 575
A.5.31.2 Detailed Description ..... 575
A.5.32 IxAtmmVc Struct Reference ..... 575
A.5.32.1 Data Fields ..... 575
A.5.32.2 Detailed Description ..... 576
A.5.32.3 Field Documentation ..... 576
A.5.33 IxAtmScheduleTable Struct Reference ..... 576
A.5.33.1 Data Fields ..... 576
A.5.33.2 Detailed Description ..... 576
A.5.33.3 Field Documentation ..... 577
A.5.34 IxAtmScheduleTableEntry Struct Reference ..... 577
A.5.34.1 Data Fields ..... 577
A.5.34.2 Detailed Description ..... 577
A.5.34.3 Field Documentation ..... 578
A.5.35 IxAtmTrafficDescriptor Struct Reference ..... 578
A.5.35.1 Data Fields ..... 578
A.5.35.2 Detailed Description ..... 578
A.5.36 IxCryptoAccAuthCtx Struct Reference ..... 579
A.5.36.1 Data Fields ..... 579
A.5.36.2 Detailed Description ..... 579
A.5.37 IxCryptoAccCipherCtx Struct Reference ..... 579
A.5.37.1 Data Fields ..... 579
A.5.37.2 Detailed Description ..... 580
A.5.38 IxCryptoAccCtx Struct Reference ..... 580
A.5.38.1 Data Fields ..... 580
A.5.38.2 Detailed Description ..... 581
A.5.38.3 Field Documentation ..... 581
A.5.39 IxEthAccMacAddr Struct Reference ..... 581
A.5.39.1 Data Fields ..... 581
A.5.39.2 Detailed Description ..... 581
A.5.40 IxEthDBMacAddr Struct Reference ..... 581
A.5.40.1 Data Fields ..... 581
A.5.40.2 Detailed Description ..... 581
A.5.41 IxEthDBPortDefinition Struct Reference ..... 582
A.5.41.1 Data Fields ..... 582
A.5.41.2 Detailed Description ..... 582
A.5.42 IxEthEthObjStats Struct Reference ..... 582
A.5.42.1 Data Fields ..... 582
A.5.42.2 Detailed Description ..... 582

A.5.43 IxFpCodeletFastPathConnection Struct Reference ..... 583
A.5.43.1 Data Fields ..... 583
A.5.43.2 Detailed Description ..... 583
A.5.44 IxFpCodeletTemplate Struct Reference ..... 583
A.5.44.1 Data Fields ..... 583
A.5.44.2 Detailed Description ..... 584
A.5.45 IxHssAccCodeletStats Struct Reference ..... 584
A.5.45.1 Data Fields ..... 584
A.5.45.2 Detailed Description ..... 584
A.5.46 IxHssAccConfigParams Struct Reference ..... 584
A.5.46.1 Data Fields ..... 584
A.5.46.2 Detailed Description ..... 585
A.5.47 IxHssAccPktHdlcFraming Struct Reference ..... 585
A.5.47.1 Data Fields ..... 585
A.5.47.2 Detailed Description ..... 585
A.5.48 IxHssAccPortConfig Struct Reference ..... 585
A.5.48.1 Data Fields ..... 585
A.5.48.2 Detailed Description ..... 586
A.5.49 IxMbufPool Struct Reference ..... 586
A.5.49.1 Data Fields ..... 586
A.5.49.2 Detailed Description ..... 587
A.5.50 IxNpeA_AtmCellHeader Struct Reference ..... 587
A.5.50.1 Data Fields ..... 587
A.5.50.2 Detailed Description ..... 587
A.5.51 IxNpeA_AtmVcFp Struct Reference ..... 587
A.5.51.1 Data Fields ..... 588
A.5.51.2 Detailed Description ..... 588
A.5.52 IxNpeA_NpePacketDescriptor Struct Reference ..... 588
A.5.52.1 Data Fields ..... 588
A.5.52.2 Detailed Description ..... 588
A.5.53 IxNpeA_RxAtmVc Struct Reference ..... 588
A.5.53.1 Data Fields ..... 588
A.5.53.2 Detailed Description ..... 589
A.5.54 IxNpeA_TxAtmVc Struct Reference ..... 589
A.5.54.1 Data Fields ..... 589
A.5.54.2 Detailed Description ..... 589
A.5.55 IxNpeDlVersionId Struct Reference ..... 590
A.5.55.1 Data Fields ..... 590
A.5.55.2 Detailed Description ..... 590
A.5.56 IxNpeMhMessage Struct Reference ..... 590
A.5.56.1 Data Fields ..... 590
A.5.56.2 Detailed Description ..... 590
A.5.57 ixUARTDev Struct Reference ..... 590
A.5.57.1 Detailed Description ..... 590
A.5.58 ixUARTStats Struct Reference ..... 591
A.5.58.1 Detailed Description ..... 591
A.5.59 LearningIpFlow Struct Reference ..... 591
A.5.59.1 Data Fields ..... 591
A.5.59.2 Detailed Description ..... 591
A.5.60 PacketisedStats Struct Reference ..... 591
A.5.60.1 Data Fields ..... 591

A.5.60.2 Detailed Description ..... 592

# Figures

1 Intel® IXP400 Software Architecture Block Diagram ..... 29
2 ixNpeDI Microcode-Download Example ..... 40
3 Data and Control Flow for the Fast Path ..... 41
4 Fast-Path Dependency Diagram ..... 43
5 ixEthAcc Block Diagram ..... 49
6 Ethernet Transmit Frame API Overview ..... 50
7 Ethernet Transmit Frame Data Flow ..... 51
8 Ethernet Receive Frame API Overview ..... 53
9 Ethernet Receive Plane Data Flow ..... 54
10 Mbuf Fields Written by Intel® XScale™ Core (left) and NPE (right) on Ethernet Rx Path ..... 56
11 Mbuf Fields Written by Intel XScale® Core (left) and NPE (right) on Ethernet Tx Path ..... 56
12 Ethernet Receive Frame API Overview ..... 59
13 Examples of Node Insertion Not Requiring Re-Balancing ..... 60
14 Examples of Node Insertion Requiring Re-Balancing ..... 61
15 Hashing ..... 62
16 Buffer Transmission for a Scheduled Port ..... 70
17 IxAtmdAccScheduleTable Structure and Order Of ATM Cell ..... 72
18 TX Done Recycling Using a Threshold Level ..... 73
19 TX Done Recycling Using a Polling Mechanism ..... 74
20 TX Disconnect ..... 75
21 RX Using a Threshold Level ..... 77
22 RX Using a Polling Mechanism ..... 78
23 Rx Disconnect ..... 79
24 Multiple VCs for Each Port, Multiplexed onto Single Line by the ATM Scheduler ..... 86
25 Translation of IxAtmScheduleTable Structure to ATM Tx Cell Ordering ..... 87
26 Services Provided by Ixatmm ..... 92
27 Configuration of Traffic Control Mechanism ..... 94
28 Component Dependencies of IxAtmm ..... 95
29 HSS/HDLC Access Overview ..... 97
30 HssAccess ..... 103
31 hssChannelisedAccess — Connect ..... 104
32 hssChannelisedAccess — Tx/Rx ..... 105
33 hssChannelisedAccess — Disconnect ..... 106
34 hssPacketisedAccess — Connect ..... 107
35 hssPacketisedAccess — Tx ..... 108
36 hssPacketisedAccess — HDLC Rx (Call-Back Implementation) ..... 109
37 hssPacketisedAccess — Disconnect ..... 110
38 Message from Intel XScale® Core Software Client to an NPE ..... 112
39 Message with Response from Intel XScale® Core Software Client to an NPE ..... 113
40 Receiving Unsolicited Messages from NPE to Software Client ..... 114
41 ixNpeMh Component Dependencies ..... 115
42 AHB Queue Manager Hardware Block ..... 116
43 AHB Queue Manager Dispatcher Operation ..... 119

44 Example Code for Polled or Interrupt Driven Dispatcher Operation ..... 120
45 STALL on IN Transactions ..... 129
46 STALL on OUT Transactions ..... 130
47 USB Dependencies ..... 134
48 UART Component Dependencies ..... 136
49 Generalized IPSec Stack Architecture ..... 138
50 ESP Packet Structure ..... 139
51 Authentication Header ..... 139
52 Security Coprocessor and IPSec Stack Scope ..... 141
53 IxCrypto Component Interface Layering ..... 142
54 Detailed IxCryptoAcc Component Architecture ..... 143
55 Basic IxCryptoAcc API Call Process Flow ..... 146
56 IxCryptoAcc API Call Process Flow for CCD (SA) Updates ..... 147
57 IxCryptoAcc Component Dependencies ..... 149
58 IxCrytoAcc Execution Model ..... 150
59 ESP Data Flow ..... 152
60 AH Data Flow ..... 153
61 NPE IxCryptoAcc Component Interface ..... 158
62 Detailed NPE-to-Intel XScale® Core Cryptographic Data Flow ..... 159
63 ixDmaAcc Dependencies ..... 161
64 IxDmaAcc Component Overview ..... 162
65 IxDmaAcc Control Flow ..... 168
66 IxDMAcc Initialization ..... 169
67 DMA Transfer Operation ..... 170
68 Intel® IXP400 Software Operating System Services Layers ..... 173
69 PCI Controller Block Diagram ..... 181
70 PCI Memory Address Space to AHB Address Space ..... 185
71 PCI Memory Address Space to AHB Memory Address Space ..... 186
72 PCI I/O Address Space to AHB Address Space ..... 187
73 PCI I/O Address Space to AHB Memory Address Space ..... 188
74 AHB Memory Address Space to PCI Memory Address Space ..... 189
75 AHB Memory Address Space to PCI Memory Address Space ..... 190
76 Intel® - Alcatel* ADSL Joint Reference Design Board ..... 213

# Tables

1 Intel® IXDP425 Network Processor Development Platform ..... 33
2 Jungo* Software Technologies Application Stack for the Intel® IXP42X Product Line Processors ..... 34
3 Intoto* Application Stack for the Intel® IXP42X Product Line Processors ..... 34
4 MAJIC-MX* Probe for the Intel® IXP42X Product Line Processors ..... 35
5 Raven* and mpDemon* for the Intel® IXP42X Product Line Processors ..... 35
6 visionPROBE* and visionICE* for the Intel® IXP42X Product Line Processors ..... 36
7 MontaVista* Linux* Distribution and Support for the Intel® IXDP425 Network Processor Development Platform ..... 36
8 Wind River* Tornado* and VxWorks* Distribution and Support for the Intel® IXDP425 Network Processor Development Platform ..... 37
9 RedBoot* Boot-Loader Distribution and Support for the Intel® IXDP425 Network Processor Development Platform ..... 37
10 GNUPro* Tools for the Intel® IXDP425 Network Processor Development Platform ..... 37
11 Linux* Kernel ..... 38
12 GNU Compiler ..... 38
13 Checking Instruction Format in the Classifier Template ..... 44
14 Checking Operations in TempCls ..... 44
15 Per-VC TempMdf ..... 45
16 Source/Type Template ..... 46
17 Sample TempCls for PPP over RFC1483 with LLC encapsulation ..... 46
18 Managed Objects for Ethernet Receive ..... 64
19 Managed Objects for Ethernet Transmit ..... 64
20 Static Data Storage ..... 65
21 Ix_mbuf Fields Required for Transmission ..... 80
22 Ix_mbuf Fields of Available Buffers for Reception ..... 80
23 Ix_mbuf Fields Modified During Reception ..... 80
24 Real-Time Errors ..... 82
25 ixAtmdAcc Supported Traffic Rates ..... 83
26 Supported Traffic Types ..... 85
27 IxAtmSch Data Memory Usage ..... 89
28 AHB Queue Manager Configuration Attributes ..... 118
29 API interfaces Available for Access Layer ..... 122
30 IN, OUT, and SETUP Token Packet Format ..... 124
31 SOF Token Packet Format ..... 124
32 Data Packet Format ..... 124
33 Handshake Packet Format ..... 125
34 Bulk Transaction Formats ..... 125
35 Isochronous Transaction Formats ..... 126
36 Control Transaction Formats, Set-Up Stage ..... 126
37 Control Transaction Formats ..... 127
38 Interrupt Transaction Formats ..... 127
39 Host-Device Request Summary ..... 128
40 SNMP Entities ..... 131
41 SNMP Definition of usbEndpointInfo ..... 132
42 SNMP Definition of usbEndpointStats ..... 132
43 Detailed Error Codes ..... 133
44 IxCryptoAcc Component API Summary ..... 145
45 Code Size Estimate ..... 154

46 SA Tunnel (Cryptographic Context) Memory Requirements ..... 154
47 Security Hardware Accelerator Throughput Estimates (Mbps) ..... 155
48 Combined NPE C Version Mappings ..... 156
49 Supported Encryption Algorithms ..... 157
50 Supported Authentication Algorithms ..... 157
51 DMA Modes Supported for Addressing Mode of Incremental Source Address and Incremental Destination Address ..... 165
52 DMA Modes supported for Addressing Mode of Incremental Source Address and Fixed Destination Address. ..... 166
53 DMA Modes supported for Addressing Mode of Fixed Source Address and Incremental Destination Address. ..... 167
54 Intel® IXP400 Software v1.2.2 ixOsServices Function Descriptions ..... 175
55 Intel® IXP400 Software v1.2.2 ixOSSL Function Descriptions ..... 178
56 Access Layer Component OS Service Dependencies ..... 179
57 API Summary ..... 182
58 PCI Interface — Important Data Types ..... 183
59 PCI Interface — Data Global to Module ..... 183
60 MIB Objects Supported in the ifTable ..... 215
61 IfTable Specific Traps ..... 216
62 ADSL Line Table ..... 217
63 ADSL ATU-C Physical Table ..... 218
64 ADSL ATU-R Physical Table ..... 219
65 ADSL ATU-R Channel Table ..... 220
66 ADSL ATU-R Performance Data Table ..... 220

# Revision History

| Date | Revision | Description |
|---|---|---|
| June 2004 | 002b | Updated brand names and document titles. |
| April 2004 | 002a | Special release. |
| February 2003 | 002 | Removed “Intel Confidential” classification. |
| February 2003 | 001 | Initial release of document. |

# Introduction 1

## 1.1 About the Intel® IXP42X Product Line of Network Processors and IXC1100 Control Plane Processor

Next-generation networking solutions must meet the growing demands from users for high performance data, voice, and networked multimedia products. Manufacturers of networking equipment must develop new products under stringent time-to-market deadlines and deliver products that can be easily upgraded in software. The IXP42X product line family is designed to meet the needs of broadband and embedded networking products such as high-end residential gateways; small to medium enterprise (SME) routers, switches, security devices; mini-DSLAMs (Digital Subscriber Line Access Multiplexers) for multi-dwelling units (MxU); wireless access points; industrial control systems and networked printers.

The IXP42X product line delivers wirespeed performance and sufficient “processing headroom” for manufacturers to add a variety of rich software services to support their applications. These are highly integrated network processors that support multiple WAN and LAN technologies giving customers a common architecture for multiple applications. With their Development Platform, a choice of operating systems, and a broad range of development tools, the IXP42X product line provides a complete development environment for faster time-to-market. This network processor family offers the choice of multiple clock speeds at 266, 400, and 533 MHz, with both commercial (0° to 70° C) and extended (-40° to 85° C) temperature options.

The IXP42X product line processors have a unique distributed processing architecture that features the performance of the Intel XScale® Core and three Network Processor Engines (NPEs). The combination of the four high-performance processors provides tremendous processing power and enables wire-speed performance at both the LAN and WAN ports. The three NPEs are designed to offload many computationally intensive data plane operations from the Intel XScale core. This provides ample “processing headroom” on the Intel XScale core for developers to add differentiating product features. Software development is made easier by the extensive Intel XScale technology tools environment that includes compilers, debuggers, operating systems, models, emulators, support services from third party vendors, and fully documented evaluation boards and kits. The compiler, assembler, and linker support specific optimizations designed for the Intel XScale microarchitecture, the ARM* instruction set v.5TE and Intel® DSP extensions.

## 1.2 Intended Audience

This document describes the Intel® IXP400 Software v1.2.2 architecture. It defines each component’s functionality, demonstrates the behavioral links between the components, and provides the common design policies of each component. It is intended for software developers and architects who are employing an IXP42X product line processor.

## 1.3 Related Documents

| Document Title | Document # |
|---|---|
| *Intel® IXP425 Network Processor Based on Intel® XScale™ Microarchitecture Component Specification* | 11725 |
| *Intel® IXP42X Product Line of Network Processors and IXC1100 Control Plane Processor Datasheet* | 250620 |
| *Intel® IXP42X Product Line of Network Processors and IXC1100 Control Plane Processor Specification Update* | 251805 |
| *Intel® XScale™ Microarchitecture Programmer's Reference Manual* | 273436 |
| *ARM* Architecture Version 5TE Specification* | ARM DDI 0100E (ISBN 0 201 737191) |
| *PCI Local Bus Specification*, Revision 2.2 | |
| *Universal Serial Bus Specification*, Revision 1.1 | |
| *UTOPIA Level 2 Specification*, Revision 1.0 | |
| IEEE 802.3 Specification | |
| IEEE 1149.1 Specification | |

## 1.4 Acronyms

| | |
|---|---|
| AAL | ATM Adaptation Layer |
| ABR | Available Bit Rate |
| ACK | Acknowledge Packet |
| ADSL | Asymmetric Digital Subscriber Line |
| AES | Advanced Encryption Standard |
| AH | Authentication Header (RFC2402) |
| AHB | Advanced High-performance Bus |
| AL | Adaptation Layer |
| APB | Advanced Peripheral Bus |
| API | Application Programming Interface |
| AQM | AHB Queue Manager |
| ATM | Asynchronous Transfer Mode |
| BSD | Berkeley Software Design |
| BSP | Board Support Package |
| CAC | Connection Admission Control |
| CBC | Cipher Block Chaining |

| | |
|---|---|
| CBR | Constant Bit Rate |
| CCD | Cryptographic Context Database |
| CDVT | Cell Delay Variation Tolerance |
| CFB | Cipher FeedBack |
| CPCS | Common Part Convergence Sublayer |
| CPE | Customer Premise Equipment |
| CRC | Cyclic Redundancy Check |
| CSR | Customer Software Release |
| CTR | Counter Mode |
| DDR | Double Data Rate |
| DES | Data Encryption Standard |
| DOI | Domain of Interpretation |
| DSL | Digital Subscriber Line |
| E | Empty |
| E1 | Euro 1 trunk line (2.048Mbps) |
| ECB | Electronic Code Book |
| ERP | Endpoint Request Packet |
| ESP | Encapsulation Security Payload (RFC2406) |
| Eth0 | Ethernet NPE A |
| Eth1 | Ethernet NPE B |
| F | Full |
| FCS | Frame Check Sequence |
| FIFO | First In First Out |
| FRAD | Frame Relay Access Device |
| FRF | Frame Relay Forum |
| FXO | Foreign Exchange Office |
| FXS | Foreign Exchange Subscriber |
| G.SHDSL | ITU G series specification for symmetric High Bit Rate Digital Subscriber Line |
| GCI | General Circuit Interface |
| GE | Gigabit Ethernet |
| GFR | Guaranteed Frame Rate |
| GPIO | General Purpose Input Output |
| HDLC | High-Level Data Link Control |

| | |
|---|---|
| HDSL2 | High Bit-Rate Digital Subscriber Line version 2 |
| HEC | Header Error Check |
| HLD | High Level Design |
| HMAC | Hashed Message Authentication Code |
| HPI | Host Port Interface |
| HPNA | Home Phone Network Alliance |
| HSS | High Speed Serial |
| HSSI | High Speed Serial Interface |
| HW | Hardware |
| IAD | Integrated Access Device |
| ICV | Integrity Check Value |
| IKE | Internet Key Exchange |
| IMA | Inverse Multiplexing over ATM |
| IP | Internet Protocol |
| IPsec | Internet Protocol Security |
| IRQ | Interrupt Request |
| ISR | Interrupt Service Routine |
| IV | Initialization Vector |
| IX_MBUF | BSD 4.4–like mbuf implementation for the Intel® IXP42X product line |
| IXA | Internet Exchange Architecture |
| IXP | Internet Exchange Processor |
| LAN | Local Area Network |
| LSB | Least Significant Bit |
| MAC | Media Access Control |
| MBS | Maximum Burst Size |
| MCR | Minimum Cell Rate |
| MD5 | Message Digest 5 |
| MFS | Maximum Frame Size |
| MIB | Management Information Base |
| MII | Media-Independent Interface |
| MLPPP | Multi-Link Point-to-Point Protocol |
| MPHY | Multi PHY |
| MSB | Most Significant Bit |
| MVIP | Multi-Vendor Integration Protocol |

| | |
|---|---|
| MxU | Multi-dwelling Unit |
| NAK | Not-Acknowledge Packet |
| NAPT | Network Address Port Translation |
| NAT | Network Address Translation |
| NE | Nearly Empty |
| NF | Nearly Full |
| NOTE | Not Empty |
| NOTF | Not Full |
| NOTNE | Not Nearly Empty |
| NOTNF | Not Nearly Full |
| NPE | Network Processing Engine |
| OC3 | Optical Carrier - 3 |
| OF | Overflow |
| OFB | Output FeedBack |
| OS | Operating System |
| PBX | Private Branch Exchange |
| PCI | Peripheral Component Interface |
| PCR | Peak Cell Rate |
| PDU | Protocol Data Unit |
| PHY | Physical Layer Interface |
| PID | Packet Identifier |
| PRE | Preamble Packet |
| QMgr | Queue Manager |
| RMII | Reduced Media-Independent Interface |
| rt-VBR | Real Time Variable Bit Rate |
| Rx | Receive |
| SA | Security Association |
| SAR | Segmentation and Re-assembly |
| SCR | Sustainable Cell Rate |
| SDRAM | Synchronous Dynamic Random Access Memory |
| SDSL | Symmetric Digital Subscriber Line |
| SDU | Service Data Unit |
| SHA1 | Secure Hash Algorithm 1 |
| SIO | Standard I/O (input/output) |

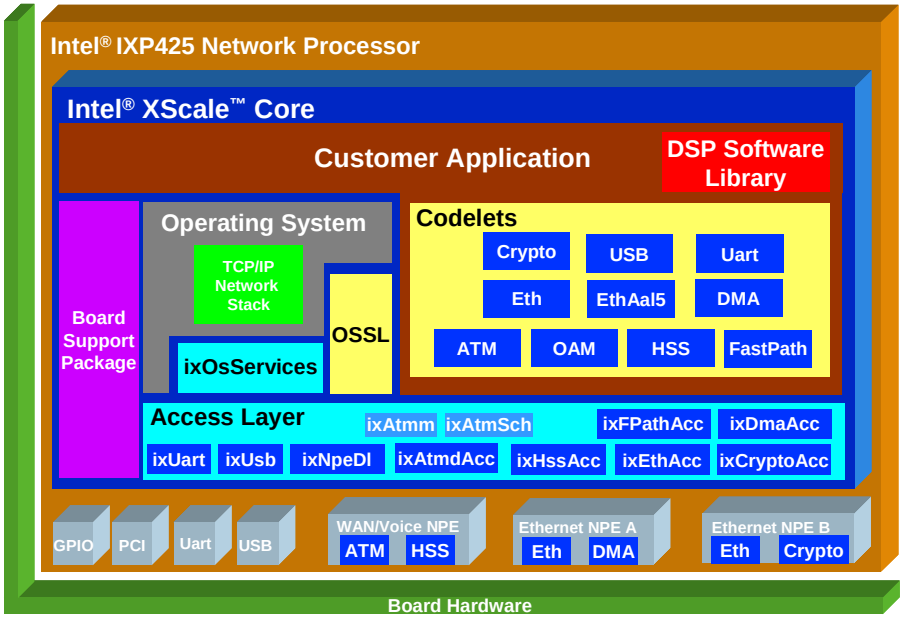
| | |
|---|---|
| SIP | Session Initiation Protocol |
| SNMP | Simple Network Management Protocol |
| SOF | Start of Frame |
| SPHY | Single PHY |
| SVC | Switched Virtual Connection |
| TCD | Target Controller Driver |
| TCI | Transmission Control Interface |
| TCP | Transmission Control Protocol |
| TDM | Time Division Multiplexing |
| ToS | Type of Service |
| Tx | Transmit |
| UBR | Unspecified Bit Rate |
| UDC | Universal Serial Bus Device Controller |
| UDC | USB Device Controller |
| UF | Underflow |
| USB | Universal Serial Bus |
| UTOPIA | Universal Test and Operation PHY Interface for ATM |
| VBR | Variable Bit Rate |
| VC | Virtual Connection |
| VCC | Virtual Circuit Connection |
| VCI | Virtual Circuit Identifier |
| VDSL | Very High Speed Digital Subscriber Line |
| VoDSL | Voice over Digital Subscriber Line |
| VoFR | Voice over Frame Relay |
| VoIP | Voice over Internet Protocol |
| VPC | Virtual Path Connection |
| VPI | Virtual Path Identifier |
| VPN | Virtual Private Network |
| WAN | Wide Area Network |
| xDSL | Any Digital Subscriber Line |
| XOR | Exclusive OR |

# Software Architecture Overview 2

## 2.1 High-Level Overview

The primary design principles of the Intel® IXP400 Software architecture are to enable the Intel® IXP42X product line hardware and allow maximum flexibility. The IXP400 software library consists of a collection of software components specific to the IXP4XX Product Line processor chip and its reference boards.

**Figure 1. Intel® IXP400 Software Architecture Block Diagram**




The **Access Layer** provides a software interface which gives customer code access to the underlying capabilities of the IXP42X product line processors and hardware ports such as AAL0 and AAL5 on UTOPIA, Ethernet, HSS HDLC, UART and USB. This layer is made up of a set of software components (Access Components), which clients can use to configure, control and communicate with the hardware.

The **ixOsServices Layer** provides a very thin set of abstracted OS services. Access Layer components abstract their OS dependency to this module.

The **NPE Firmware** consists of one or more loadable and executable NPE instruction files that implement the NPE functionality behind the IXP400 software library. These files are incorporated into the IXP400 software library at build time, and the library includes a NPE downloader component that provides NPE code version selection and downloading services.

The example **Codelets** are narrowly focused and show how to use each service or function provided by the Intel XScale core library and the underlying hardware. The codelets are organized by hardware port type and typically exercise some Layer-2 functionality on that port, such as: AAL5 PDU Transmit / Receive over UTOPIA, Channelized or HDLC Transmit / Receive over HSS, Ethernet frame Transmit / Receive.

The **Operating System Services Layer (OSSL)** defines a portable interface for operating system services. The codelets abstract their OS dependency to this module.

The **DSP Software Library** comprises a rich and powerful set of general and multimedia signal processing kernels optimized for maximum performance on the Intel XScale microarchitecture. One example of this multimedia signal processing is the ability of the IXP400 software to implement voice-processing algorithms on the Intel XScale core for low voice port counts.

## 2.2 Deliverable Model

Intel® IXP400 Software v1.2.2 consists of these elements:

- IXP400 software v1.2.2 library
- Complete documentation and source code for library components
- NPE firmware images embedded in the IXP400 software v1.2.2 library
- Example codelets

The software releases do not include tools to develop NPE software. The supplied NPE functionality is accessible through the APIs provided by the IXP400 software library.

## 2.3 Operating System Support

The Intel XScale microarchitecture offers a broad range of tools together with support for two widely adopted operating systems. The IXP400 software supports VxWorks* and the standard Linux* 2.4 kernel. MontaVista* Software will provide the support for Linux. Support for other operating systems are planned for future releases.

The IXP400 software's library is OS-independent in that all components are written in ANSI-C with no direct call to any OS library function that is not covered by ANSI-C. This enables the devices to be compatible with multiple operating systems and allows customers the flexibility to port the IXP4XX Product Line processor and to their OS of choice.

## 2.4 Development Tools

The Intel XScale microarchitecture offers a broad range of tools together with support for two widely adopted operating systems. Developers have a wide choice of third-party tools including compilers, linkers, debuggers and BSPs. Tools include Wind River* Tornado* 2.1.1 for VxWorks 5.4 real-time operating system, and the complete GNU* Linux development suite.

## 2.5 Threading and Locking Policy

The IXP400 software Access Layer does not implement processes or threads. The architecture assumes execution within a preemptive multi-tasking environment with the existence of multiple-client threads and uses common, real-time OS functions such as semaphores, task locking, and interrupt control to protect critical data and procedure sequencing. These functions are not provided directly by the OS, but by one or more OS abstraction components.

## 2.6 Polled and Interrupt Operation

It is possible to use access layer components in a polled and in an interrupt driven mode of operation. Access layer components do not autonomously bind themselves to interrupts. It is the responsibility of the software engineer to control when an access layer component is hooked to a particular interrupt source.

All data path interfaces are executable in the context of both IRQ and FIQ interrupts. In addition, all data path interfaces can be executed in a non-interrupt task context.

## 2.7 Statistics and MIBs

The IXP400 software Access Layer components only maintain statistics that access layer clients cannot collect of their own accord. The Access layer components do not provide management interfaces (MIBs). Access Layer clients can use the statistics provided to implement their own MIBs.

## 2.8 Network Data Buffer Management

Buffer management is the general principle of how and where network data buffers are allocated and freed in the entire system. Network data buffers, whose formats are known to all involved components, need to flow between software components. Some components need to allocate and/or free these buffers.

In the IXP400 software, the access-driver software layer follows a simple buffer management principle: all buffers used between access driver software and clients above the access driver software must be allocated and freed by the clients. The client passes a buffer to an access driver component for various purposes (Tx and Rx in general), and the access layer component returns the buffer to the client when the requested job is completed. The access layer component may call a client-registered call-back function to return the buffer, or may put the buffer back on a free queue for the client to poll.

The network data buffers and their formats as well as management must be agreed by all components so that they can efficiently flow in the system. The IXP400 software uses two buffer formats for all network data:

- mbuf
- raw buffer

These two formats are assumed by IXP400 software's access-level components and NPEs.

The mbuf format is originally defined in BSD TCP/IP code distribution. VxWorks from Wind River implements mbuf in a specific way, that is, the buffer is always "external." This simplifies the buffer management without sacrificing functionality and flexibility. Although the mbuf format used is heavily related to VxWorks, it is used in the IXP400 software architecture as one of the two OS-independent buffer formats.The mbuf component is typically used for packet-switched network data. The Access Layer components that use this format are ixAtmdAcc, ixEthAcc, ixFpathAcc, ixHssPacketized, and ixUSB.

Raw buffer format is just a contiguous section of memory represented in one of two ways. One way to pass raw buffers between two components is through an agreement to circularly access the same piece of raw buffer. One component circularly writes to the buffer while the other component circularly reads from the buffer. The buffer length and alignment are parts of the agreement. At run-time, another communication channel is needed to synchronize the read pointer and write pointers between the two components.

The other way to pass raw buffers between two components is through passing a pointer to the buffer between the components. If all buffers are the same size and that size is fixed, the length can be made known during configuration. Otherwise, another communication channel in run-time is needed to tell the length of the buffer. The raw buffer component is typically used for circuit-switched network data (i.e., TDM based). The HSS Channelized service uses raw buffers.

# Software Development Tools 3

This chapter includes information about these software development tools:

- Intel Hardware
- Third-Party Software and Hardware

## 3.1 Intel Hardware

### 3.1.1 Intel® IXDP425 Network Processor Development Platform

**Table 1. Intel® IXDP425 Network Processor Development Platform**

| Product | Intel® IXDP425 Network Processor Development Platform |
|---|---|
| Description | The IXDP425 platform, in conjunction with the operating system and tools, forms an integrated development environment. This development platform can be used to prototype a wide variety of broadband and embedded networking products such as high-end residential gateways, enterprise and small to medium enterprise (SME) routers, switches, and security devices, mini-DSLAMs (Digital Subscriber Line Access Multiplexers), xDSL line cards, wireless access points, industrial control systems, and networked printers. |
| Product Brief URL | http://developer.intel.com |
| Support | Intel |

# 3.2 Third-Party Software and Hardware

## 3.2.1 Software Stacks for the Intel® IXP42X Product Line Processors

**Table 2. Jungo* Software Technologies Application Stack for the Intel® IXP4XX Product Line Processors**

| **Product** | **OpenRG* for the Intel® IXP42X Product Line Processors** |
|---|---|
| **Description** | OpenRG is a scalable suite of software infrastructure and technologies that OEMs require in order to bring residential gateways / IADs to market. OpenRG leverages a wide range of compelling broadband-based applications and services and includes an operating system, drivers, and remote-management capabilities. OpenRG delivers a set of highly integrated solutions required for the home and small office such as:<br>• Home networking (HomePNA, wireless LAN - 802.11b, Bluetooth*)<br>• Network security (Stateful Packet Inspection)<br>• Virtual Private Networking (VPN)<br>• Remote management (Web- and SNMP-based)<br>• Remote-update capabilities<br>• Supports industry-leading standards such as CableHome 1.0<br>Target platforms: DSL modems, Cable modems, CPEs, IADs, Wireless access points, routers. |
| **Product Brief URL** | http://www.jungo.com/openrg/reference_designs_intelixp425.html |
| **Manufacturer** | Jungo (http://www.jungo.com) |
| **Support** | Jungo |

**Table 3. Intoto* Application Stack for the Intel® IXP42X Product Line Processors**

| **Product** | **IGateway* for the** |
|---|---|
| **Description** | Fire wall (with NAT and ALGs), VPN (IPSec, IKE and PKI), and IIDS (Integrated Intrusion Detection System) |
| **Product Brief URL** | http://www.intotoinc.com |
| **Manufacturer** | Intoto Inc. (http://www.intotoinc.com) |
| **Support** | Intoto |

## 3.2.2 Hardware and Software Tools for the Intel® IXP42X Product Line Processors

### Table 4. MAJIC-MX* Probe for the Intel® IXP42X Product Line Processors

| Product | MAJIC-MX* Probe |
|---|---|
| Description | The MAJIC-MX intelligent debug probe provides a high-speed hardware interface between the Intel XScale® Core processors, with the on-chip debug interface, and industry-standard debuggers. It is available with EDB (EPI's source-level debugger), ADS, and any third-party debugger which is either RDI 1.5.1-compliant or MDI-compliant.<br>• 10/100Base-T Ethernet interface<br>• Supports the Intel XScale microarchitecture on-chip trace<br>• Supports on-chip hardware breakpoints<br>• Unlimited software breakpoints<br>• Programmable JTAG Clock (TCK = 2 KHz to 40 MHz) |
| Product Brief URL | |
| Manufacturer | Embedded Performance Inc. (http://www.epitools.com) |
| Support | Embedded Performance Inc. |

### Table 5. Raven* and mpDemon* for the Intel® IXP42X Product Line Processors

| Products | Raven*<br>mpDemon* |
|---|---|
| Description | **Raven**<br>The Raven is a mid-cost interface used in the design, debug, and programming of microprocessor and microcontroller-based embedded systems. One side of the Raven interfaces to the parallel port of a host, IBM-compatible PC and the other side connects to an OCD (On-Chip Debug) port of the target system. This port may be JTAG, E-JTAG, OnCE, COP, BDM, or any of several other types of connections.<br>**mpDemon**<br>The mpDemon* provides the embedded systems developer with an intelligent target interface available for target on-chip debugging. The mpDemon* supports a wide range of target chips as well as the Macraigor Systems API.<br>• 10BaseT Ethernet interface<br>• IEEE-1284 parallel port supporting Nibble, Compatible, EPP, and ECP modes<br>• RS-232 DCE port supporting baud rate up to 115 Kbaud<br>• Support for programming target Lattice, and Xilinx devices<br>• Target flash programming with our host application |
| Product Brief URL | • Raven<br>http://www.ocdemon.net/raven_ds.pdf<br>• mpDemon<br>http://www.ocdemon.net/mp_demon.pdf |
| Manufacturer | Macraigor Systems LLC (http://www.ocdemon.net) |
| Support | Hardware Support Only<br>Contact Macraigor Systems LLC |

**Table 6. visionPROBE* and visionICE* for the Intel® IXP42X Product Line Processors**

| Product | visionPROBE* II<br>visionICE* II |
|---|---|
| Description | **visionPROBE II**<br>The visionPROBE BDM/JTAG debugging cable is at the entry level to the Wind River family of hardware-assisted debugging solutions for embedded microprocessor systems.<br>• High-speed parallel download<br>• RTOS awareness<br>• On-chip debug target control<br>• Flash memory programming<br>• External triggering<br>• Source-level debugging<br>**visionICE II**<br>visionICE II represents a dramatic performance improvement over the original visionICE. It provides a fully-integrated hardware-assisted and software development tool that addresses the entire product development cycle, from board bring-up to debug and production test.<br>• High-speed Ethernet connection<br>• RTOS awareness<br>• On-chip debug target control<br>• Flash memory programming<br>• External triggering<br>• Source-level debugging |
| Product Brief URL | • visionPROBE II<br>http://www.windriver.com/products/visionprobe2/index.html<br>• visionICE II<br>http://www.windriver.com/products/visionice2/index.html |
| Manufacturer | Wind River Systems* (http://www.windriver.com) |
| Support | Wind River Systems |

## 3.2.3 Software for the Intel® IXDP425 Network Processor Development Platform

**Table 7. MontaVista* Linux* Distribution and Support for the Intel® IXDP425 Network Processor Development Platform**

| Product | MontaVista* Support for the Intel® IXDP425 Network Processor Development Platform BE Intel XScale® Core, Linux Kernel 2.4.18, BSP, Tool Chain, and Utilities |
|---|---|
| Description | Linux 2.4.18, GNU Tool Chain, BSP, etc. |
| Product Brief URL | http://www.mvista.com/pro/downloads/ixp425.html |
| Manufacturer | MontaVista (http://www.mvista.com) |
| Support | MontaVista |

### Table 8. Wind River* Tornado* and VxWorks* Distribution and Support for the Intel® IXDP425 Network Processor Development Platform

| | |
|---|---|
| **Product** | **Tornado 2.1.1 Evaluation Kit for the Intel® IXDP425 Network Processor Development Platform** |
| **Description** | Tornado 2.1.1 for Intel XScale® Core, VxWorks 5.4 (60-day evaluation copy) |
| **Product Brief URL** | http://www.windriver.com/products/html/tools_ds.html |
| **Manufacturer** | Wind River Systems (http://www.windriver.com) |
| **Support** | 60-day Evaluation Kit: Register evaluation kit to receive support at: http://www.windriver.com/windsurf |

### Table 9. RedBoot* Boot-Loader Distribution and Support for the Intel® IXDP425 Network Processor Development Platform

| | |
|---|---|
| **Product** | **RedBoot* v 2.0 for BRH** |
| **Description** | RedBoot is a standardized, embedded debug and boot-strap solution that provides firmware for running and debugging eCos* and GNUPro* applications and embedded Linux* systems. |
| **Product Brief URL** | http://www.redhat.com/embedded/technologies/redboot |
| **Manufacturer** | Red Hat (http://www.redhat.com) |
| **Support** | Red Hat |

### Table 10. GNUPro* Tools for the Intel® IXDP425 Network Processor Development Platform

| | |
|---|---|
| **Product** | **GNUPro* Tools** |
| **Description** | With the GNUPro tool suite, developers get the GNU compiler (gcc) for application development in C, C++, and assembly, and the source- and assembly-level GNU debugger (gdb). In addition, GNUPro includes embedded libraries and low-level code for embedded boards and an Instruction Set Simulator (ISS) for software-based co-development.<br>• gcc Highly optimized ANSI-C compiler<br>• g++ ANSI-tracking C++ compiler<br>• gdb Source and assembly-level debugger<br>• gas GNU assembler<br>• ld GNU linker<br>• Insight Graphical user interface for GDB<br>• Source-Navigator Source code comprehension tool<br>• GNUPro also includes libraries, binary utilities, general utilities and documentation. |
| **Product Brief URL** | http://www.redhat.com/software/gnupro/xscale |
| **Manufacturer** | Red Hat (http://www.redhat.com) |
| **Support** | Red Hat |

## 3.2.4 Open Source Software and Tools

### Table 11. Linux* Kernel

| | |
|---|---|
| **Product** | **Intel XScale® Microarchitecture Big-Endian Linux* Kernel** |
| **Description** | Open-source Linux kernel with big-endian Intel XScale core patches. This distribution is unsupported and released “as is.” |
| **How to obtain** | • General Linux Kernel: ftp://ftp.kernel.org/pub/linux/kernel/<br>• Intel XScale core-specific Kernel Patches: ftp://source.mvista.com/pub/ds-patches/2.4/ |

### Table 12. GNU Compiler

| | |
|---|---|
| **Product** | **GNU Compiler Collection (gcc)** |
| **Description** | Open-source C and C++ development tools. |
| **How to obtain** | http://gcc.gnu.org/releases.html |

# Access-Layer Components 4

This chapter describes the IXP400 software v1.2.2 Access Layer components.

## 4.1 ixNpeDl

The NPE downloader (ixNpeDI) component is a stand-alone component providing a facility to download a version of microcode to NPE-A, NPE-B, or NPE-C in the system. The NPE downloader provides the ability to query the microcode versions available within a microcode image and to query microcode versions currently loaded on a particular NPE.

The NPE Downloader component also can reset an NPE and stop/start execution on an NPE.

### 4.1.1 Functional Interface Description

- Contains the default NPE microcode image, which contains up-to-date microcode for each NPE.
- Finds a specified version of microcode from the microcode image, and downloads that version to the specified NPE.
- Checks the microcode image for a particular version specified by the client.
- Reports to a client, on request, the version of microcode currently loaded on a particular NPE.
- Enables a client to start/stop/reset an NPE for performing a download.
- Enables a client to supply a different microcode image to use in place of the default image.

  This “image-override” facility provides increased testability and flexibility, but is not intended for general use.
- Runs on the Intel® IXP42X Product Line of Network Processors and IXC1100 Control Plane Processor Intel XScale core.

### 4.1.2 Microcode Images

All versions of microcode available for download to the NPEs are contained in a microcode image. Each version contains a number of blocks of instruction, data, and state-information microcode that is downloaded into the NPE memory and registers. Each version also contains a download map that specifies how to extract the individual blocks of that version’s microcode.

Given a microcode image in memory, the NPE Downloader can locate versions from that image in memory, extract and interpret the contained download map, and download the code accordingly.

### 4.1.3 Usage Example

Figure 2 shows one way to use the ixNpeDl APIs to download a microcode image to an NPE.

**Figure 2. ixNpeDl Microcode-Download Example**

```
/* Stop the NPE and reset its registers before attempting a download */
    ixNpeDlNpeStopAndReset(...)
/* Get the latest version of NPE code available */
    ixNpeDlAvailableVersionsListGet(....)
/* Download NPE code */
    ixNpeDlVersionDownload(...)
/* Initialize NPE Message handler */
    ixNpeMhInitialize(...)
/* Start NPEs */
    ixNpeDlNpeExecutionStart(...)
```

## 4.2 ixFpathAcc (Fast Path)

This section defines the IxFpathAcc component for IXP400 software. The IxFpathAcc component allows an application level bridging or routing fast path mechanism to configure and access the Fast Path (Classifier /Modifier) functionality available on the NPEs.

### 4.2.1 Scope

IxFpathAcc is an optional IXP400 software component. This section describes these details of the IxFpathAcc:

- Functionality and services
- Interfaces to use the services
- Conditions and constraints for using the services
- Its dependency on other IXP400 software components
- Its performance and resource usage estimates

## 4.2.2 Functional Description

**Figure 3. Data and Control Flow for the Fast Path**

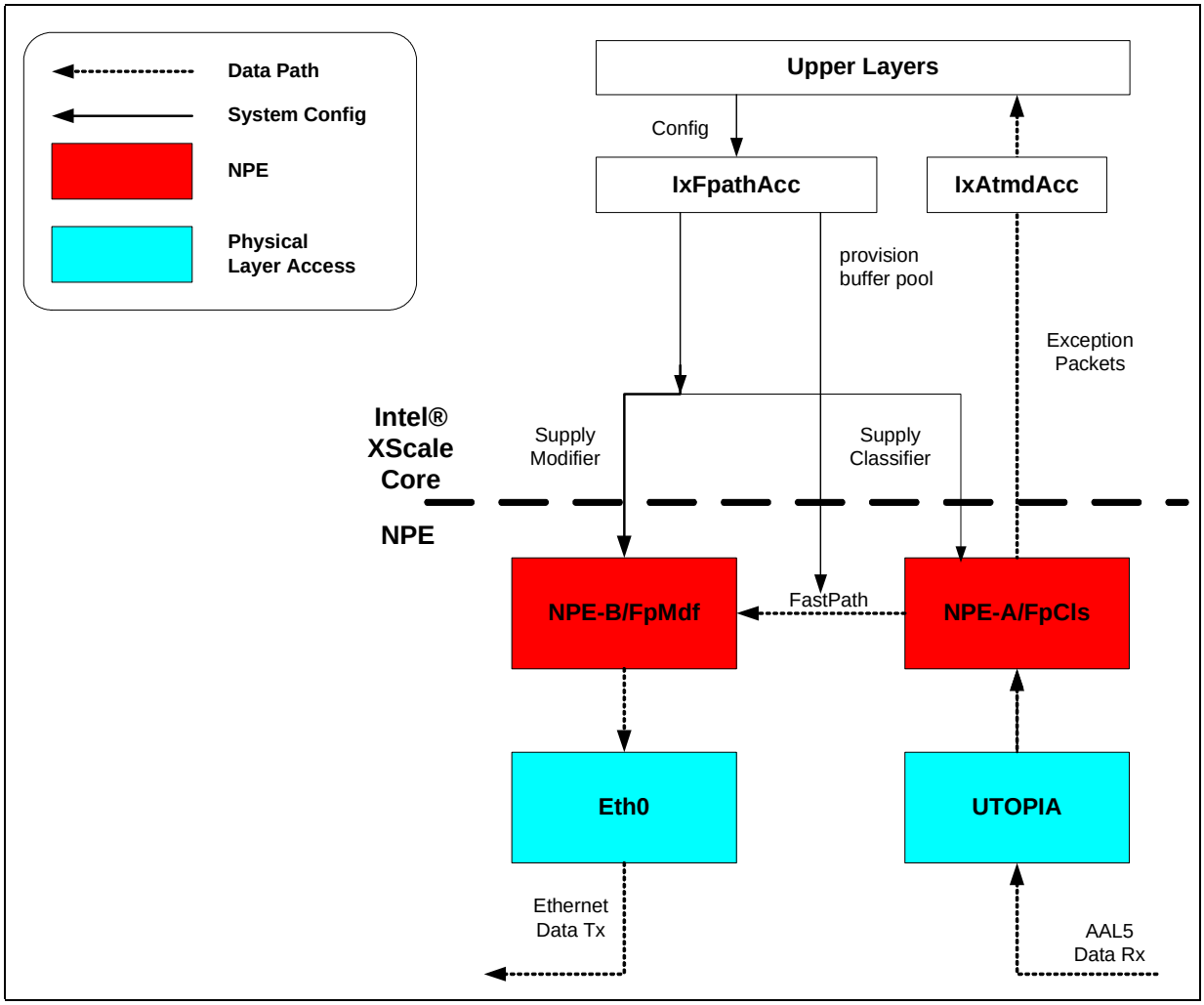



Figure 3 shows the data and control flow for the fast path service. The IxFpathAcc component provides access to the underlying fast path services provided by NPEs. Currently, only NPE-A (ATM) and NPE-B (Ethernet) together support fast-path functionality.

The NPE fast-path service operates by classifying incoming packets according to a set of instructions (classification template) to determine if they qualify for fast path. Qualifying packets are sent directly to NPE-B where they are modified according to a set of modification instructions (modification template) and transmitted on the Ethernet port.

The IxFpathAcc components allow up to eight classification/modification pairs to be configured on the NPEs at a time, one template pair per ATM VC.

### 4.2.3 Enabling/Modifying/Disabling Fast Path for a VC

A fast path controlling entity about IxFpathAcc submits template pairs per VC to IxFpathAcc using ixFpathAccTemplatePairSet(). IxFpathAcc sends the classifier template to NPE-A and sends the modifier template to NPE-B, and automatically enables fast-path processing for the VC. The next packet to arrive is subjected to the fast-path function.

Packets are not transported on the fast path if the VC has not been connected on IxAtmdAcc or if IxEthAcc has not enabled the Ethernet port.

Dynamic modification of a template pair requires fast-path processing for the particular VC to be disabled for a brief period while the new template is installed. Packets continue to flow via the slow path.

If the fast-path service is enabled on a VC, it can be disabled by clearing the template pair.

### 4.2.4 Automatic Packet Operations

For incoming AAL-5 packets, the fast-path service checks the integrity of the AAL-5 CRC, and any packets that fail CRC are automatically sent to the slow path.

AAL-5 packets that qualify for the fast path have the AAL-5 trailer stripped automatically before they are supplied to the packet modifier.

If the Ethernet service (NPE Service) is overloaded — that is, the transmit queue is full — packets, that would otherwise qualify for the fast path are automatically diverted to the slow path.

The modifier automatically inserts the ethernet type for all packets supplied by the classifier. The ethernet type is configurable via the IxFpathAcc interface.

### 4.2.5 Application Support and MAC Address Service

The fast path is specifically designed to support applications such as Ethernet half-bridging and home-gateway routing. In supporting Ethernet bridging to multiple WAN VCs, the fast path provides access to source MAC addresses of incoming ATM packets per VC.

The fast path does not learn any MAC addresses, but makes them available to the Intel XScale core (to be learned as required).

IxFpathAcc provides a call-back mechanism that is invoked each time a packet with a new MAC address is observed on a VC. One MAC-address callback can be registered per VC.

## 4.2.6 Fast-Path Buffers

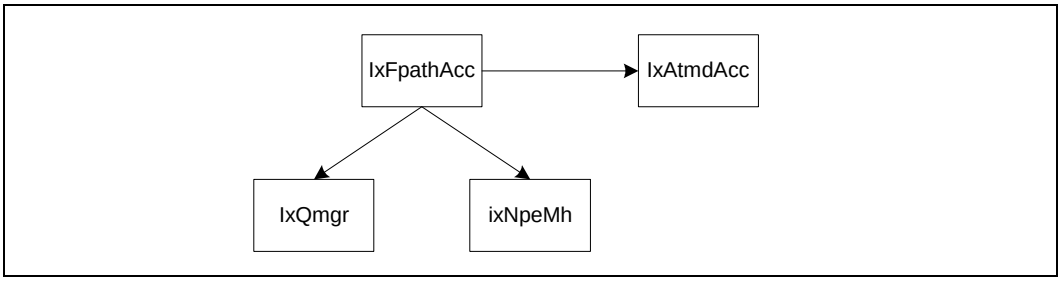
The fast-path service requires buffers that are used to store and convey packets on the fast path. Buffers supplied to IxAtmdAcc (slow path buffers) are never sent on the fast path.

A single fast-path buffer pool is maintained within the fast-path service and must be populated by supplying buffers to IxFpathAcc before any packets can be conveyed on the fast path.

Unlike certain other access components, buffers are not assigned per VC. A buffer can, therefore, contain a packet from any incoming VC that has a classification-template pair assigned. Hence all fast-path buffers must be the same size. The fast-path service does not chain buffers together to accommodate packets that would exceed a single buffers capacity. Any such packets are automatically disqualified from the fast path and sent on the slow path.

## 4.2.7 Fast-Path Dependency Diagram

**Figure 4. Fast-Path Dependency Diagram**




IxQmgr is used by Fpath Acc to supply fast-path buffers to the NPE.

IxNPEMh is used to send and receive fast-path messages to and from the NPEs. It is across this interface that all control information is communicated with the NPEs.

There is a dependency on IxAtmdAcc because under certain circumstances, fast-path buffers can be returned to IxAtmdAcc. These buffers must be recycled to the fast-path pool.

Another dependency is implicit: The fast path cannot be enabled for a VC if the VC has not already been connected on IxAtmdAcc.

There is an implicit dependency on IxEthb: No traffic flows on the fast path if the IxEthb component is not configured for the Eth0 port.

## 4.2.8 Error Handling

IxFpathAcc returns an error type to the user when the client is expected to handle the error. Internal errors are reported using standard IXP400 software error-reporting techniques, such as the OS-services and error-reporting mechanism.

## 4.2.9 NPE FPath Classifier/Modifier Configuration

A proprietary set of instructions (Opcodes) is supported by the Fpath component to facilitate the construction of modifier and classifier templates for the fast-path capability

### 4.2.9.1 Fast-Path Classifier Template Definition

The first four bytes of the classifier template are used to indicate AAL-5 length and UU comparing values. AAL-5 max length is stored at Offset 0 of the template as 16-bit, little-endian value. The UU comparing value is stored at Offset 2. The byte at Offset 3 is reserved and must be left blank.

After that, Nc many of checking instructions with the format shown in Table 13.

**Table 13. Checking Instruction Format in the Classifier Template**

| 1 Byte | 1 Byte | 1 Byte (Optional) | *Np* Byte Depending on *Op, Nd* (Optional) |
|---|---|---|---|
| *Op* | *D0* | *Nd* | *Pt* |
| Operation, unsigned char. | Position (or relative position, relative to the current location) of the first byte to check, signed char, nonnegative means from beginning of the buffer, negative means from the end of the buffer. | Number of bytes of data from the first byte needed to perform the checking, unsigned char. | Parameter for this operation. |

Given a processing operation, Nd may only have limited number of varieties. For example, for the operation “one-out-of-four-match,” below, Nd must be 4.

The defined set of instructions for the FPath classifier definition is shown in Table 14.

**Table 14. Checking Operations in TempCls**

| *Op*name (Value) | *Nd* Varieties | *Np* Value | Definition of the Operation and Parameter[1] |
|---|---|---|---|
| End(0) | N/A[2] | 0 | No more checking is needed. |
| UpperLimit(1) | 2,4 | *Nd* | *Nd* byte as data unsigned integer must be <= *Nd* byte of *Pt* as an unsigned integer. |
| LowerLimit(2) | n/a | 1 | One byte data as an unsigned integer must be >= One byte of *Pt* as an unsigned integer. |
| Match(3) | 1, 2, 3, 8, 10 | *Nd* | *Nd* byte data must be == *Nd* byte of *Pt*. |
| Checksum(4) | 10 (as number of shorts) | 0 | *Nd* many short (i.e. 2-byte) data 16-bit checksum must be == 0xFFFF. |
| 1-out-of-4-match(5) | 2, 4 | 4*4 | *Nd* byte data must be == one of the 4 *Nd*-byte data in *Pt* (although both 2 and 4 bytes matching options will be in 4-byte format in *Pt*). |
| A or B match(6)[3] | N/A | 2 | One byte data must be either one of the two bytes in *Pt*. |
| MacWanLearning(7) | N/A | 0 | Inform learned WAN MAC address. |
| Conditional Branch(8)[3] | SpecialNd | 1 | If one byte match, skip next “SpecialNd” many bytes of TempCls. Else go to the next test. |

**NOTES:**
1. All values in *Pt* other than AalMaxLength are stored MSByte first, i.e. Big-Endian.
2. ‘n/a’ means *Nd* does not exist for this operation.
3. This operation is only used in NAT.

Removal of the AAL-5 trailer is hard-coded in Fpath classifier. The operation is always performed if a packet passes the classifier test, before sending the packet to FpathMdf.

### 4.2.9.2 Fast-Path Modifier Template Definition

The input to the Fpath modifier is an AAL-5 PDU sent by the FPath classifier, together with an identifier to relate the PVC on which the PDU was received and matching indices for 1-out-of-4-match instruction.

The Per-PVC template is as shown in Table 15.

**Table 15. Per-VC TempMdf**

| Bytes | Length | Meaning |
|---|---|---|
| 0,1 | 2 | Protocols enabled<br>• Bit 0 = Bridged<br>• Bit 1 = IP<br>• Bit 2 = NAT<br>• Bit 3 = NAPT<br>• Others = (Unused) |
| 2,3 | 2 | Delta mData/mLen: To be added to mData unconditionally before mbuf is read.<br>This value should be the difference between the base of AAL-5 PDU and one of the following:<br>• Base of Ethernet packet for bridge<br>• Base of IP packet for IP router |
| 4,7 | 4 | Byte 0-3 of destination MAC address of destination 0. |
| 8,11 | 4 | Byte 0-3 of destination MAC address of destination 1. |
| 12,15 | 4 | Byte 0-3 of destination MAC address of destination 2. |
| 16,19 | 4 | Byte 0-3 of destination MAC address of destination 3. |
| 20,21 | 2 | Byte 4-5 of destination MAC address of destination 0. |
| 22,23 | 2 | Byte 4-5 of destination MAC address of destination 1. |
| 24,25 | 2 | Byte 4-5 of destination MAC address of destination 2. |
| 26,27 | 2 | Byte 4-5 of destination MAC address of destination 3. |
| 28,31 | 4 | Local destination IP address of destination 0. (For NAT or NAPT. For NAT no table lookup is needed.) |
| 32,35 | 4 | Local destination IP address of destination 1. (For NAPT) |
| 36,39 | 4 | Local destination IP address of destination 2. (For NAPT) |
| 40,43 | 4 | Local destination IP address of destination 3. (For NAPT) |
| 44.45 | 2 | Local destination port number of destination 0. (For NAPT) |
| 46,47 | 2 | Local destination port number of destination 1. (For NAPT) |
| 48,49 | 2 | Local destination port number of destination 2. (For NAPT) |
| 50,51 | 2 | Local destination port number of destination 3. (For NAPT) |

The template shown in Table 16 is also supplied for applying to all PVCs.

**Table 16. Source/Type Template**

| Byte | Length | Meaning |
|---|---|---|
| 0, 5 | 6 | Local-source MAC address. |
| 6, 7 | 2 | Ethernet type to be put into MAC header. |

### 4.2.9.3 Sample Classifier Template

Table 17 gives a sample classifier template for PPP over RFC 1483 with LLC encapsulation.

**Table 17. Sample TempCls for PPP over RFC1483 with LLC encapsulation**

| Instruction Bytes | Meaning |
|---|---|
| • 0xF4, 0x05<br>• 0x00<br>• 0x00 | • First two bytes indicate the maximum AAL-5 PDU size: 1,524 bytes.<br>• The third byte is set to a default UU indication of 0x00.<br>• The final byte is reserved and set to zero. |
| • 0x03<br>• 0x00<br>• 0x06<br>• 0xfe, 0xfe, 0x03, 0xcf<br>• 0x00, 0x21 | • The first byte (0x03) indicates a byte-matching operation.<br>• The second byte indicates to begin the check at offset zero of the supplied PDU.<br>• The length of the data check is 0x06 bytes, as indicated by the third byte.<br>• The next four bytes are the LLC header.<br>• The last two bytes are the two-byte PPP header. |
| • 0x03<br>• 0x06<br>• 0x00<br>• 0x45 | • The first byte specifies a match operation.<br>• The second byte indicates that the comparison should begin at offset 6 of the supplied PDU.<br>• The third byte indicates that the comparison is just a single byte.<br>• The fourth byte indicates that the byte should represent IPv4 with a five-word header length. |
| 0x01, 0x08, 0x02, 0x05, 0xF4 | This instruction (0x01) verifies that the two-byte field at Offset 8 in the PDU (i.e. the IP length field) is less than the supplied value (0x05f4). |
| 0x02, 0x0b, 0x01 | Checks the value (0x02 instruction) of offset 0x0b (i.e., the IP Time-To-Live field) is greater than or equal to the parameter given (0x01). |

# 4.3 ixEthAcc

## 4.3.1 ixEthAcc Overview

The ixEthAcc component provides data plane, control plane, and management plane information for the Ethernet MAC devices residing on the IXP4XX Product Line processors. The IXP42X product line processors contain one or two 10/100-Mbps Ethernet MAC devices depending on which part is being used.

The data path for each of these devices is accessible via two dedicated NPEs. One Ethernet MAC is provided on each NPE. The NPEs are connected to the North AHB for access to the SDRAM where frames are stored. The control access to the MAC registers is via the APB Bridge, memory mapped to the Intel XScale core.

The ixEthAcc component is strictly limited to supporting the internal Ethernet MACs on the IXP42X Product Line processor.

The services provided by the Ethernet Access component include:

- Ethernet Frame Transmission
- Ethernet Frame Reception
- Ethernet MAC Statistics, Tracking and Reporting
- Ethernet Filtering/Learning Database Management

The Ethernet NPEs also provide AAL-5-to-Ethernet fast-path packet modification. These capabilities are not accessible via the ixEthAcc component.

PHY control is accomplished via the MII interface which is accessible via the MAC control registers. This PHY control is not performed by the ixEthAcc component. A generic MII API is provided by the ethAccMiiControl component. This API supports one reference device, namely the Intel® LXT972 PHY. Nothing precludes this referenced example from being adapted to support alternate Ethernet PHY devices. The features supported by the ethAccMiiControl module include:

- Primitives to exercise the MII
- Advertisement configuration
- Line Speed 10/100-Mbps
- Link-state watch
- Restart negotiation.
- Peer-advertisement reporting

Although mechanisms to set the port operation state have been provided in the ixEthAcc module, true operating state-link indications should be obtained from the ethAccMiiControl module.

The general principle of avoiding unnecessary copying of data for performance is adhered to in this component. The BSD-based buffering scheme is used.

## 4.3.2 Role of the Ethernet NPE Firmware

The Ethernet NPE firmware is responsible for moving data between an Ethernet MAC and external data memory, where it can be made available to the Intel XScale core. In addition, the Ethernet NPE firmware performs a number of data processing operations.

On the Ethernet receive path, the Ethernet NPE firmware performs filtering (according to the destination MAC address), learning (according to the source MAC address), and MAC statistics collection. On the Ethernet transmit path, the Ethernet NPE firmware performs priority queuing of outgoing frames and MAC statistics collection. In addition, the Ethernet NPE firmware may perform frame payload modification functions associated with fast-path operation.

It is important to note that the Ethernet NPE firmware support for Ethernet data transport (including filtering, learning, and priority queuing) does not extend to support for all Ethernet-related protocols and functions. For example, support for VLANs, the spanning tree algorithm, and the parsing of the Tag Control Information field (on inbound or outbound Ethernet frames) are not included in software releases 1.0 or 1.1of the Ethernet NPE firmware. However, the lack of NPE-level support for these features in no way inhibits the Intel XScale core-based software from implementing them.

Communication between an Ethernet NPE and the Intel XScale core is facilitated by two mechanisms: the Queue Manager (which is external to the NPE) and the APB (which the NPE accesses via its internal debug and execution control block). The primary use of these interfaces is to communicate the existence and location of network payload data and Ethernet service configuration information in external SDRAM.

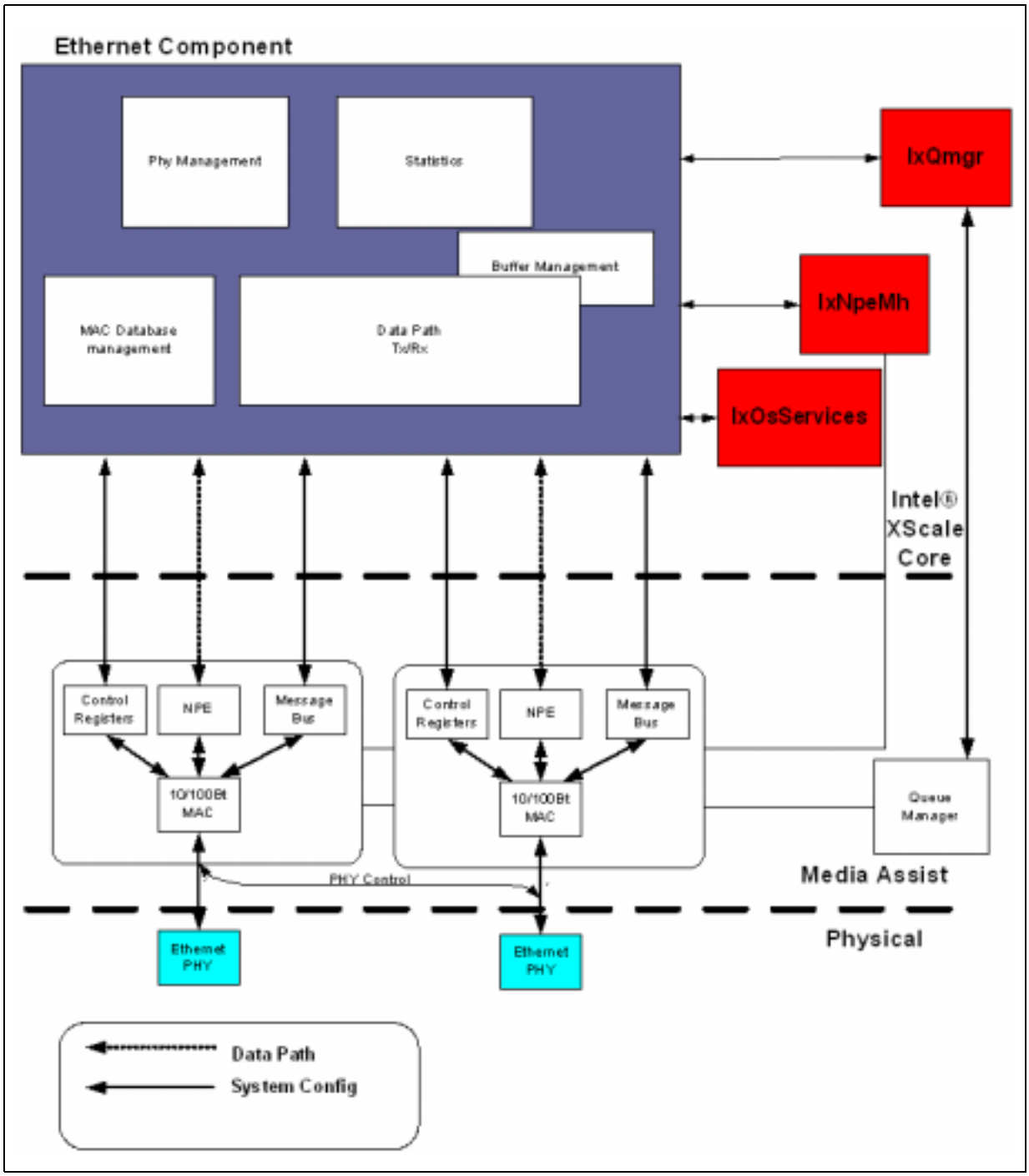
Ethernet frames are presented to an Ethernet-capable NPE via its organic, Ethernet coprocessor which serves as an interface between the Ethernet MAC and the NPE. Ethernet frame payloads are transferred from the Ethernet coprocessor to the host NPE in discrete blocks of data. The frames are buffered in NPE internal data memory, optionally filtered according to their destination MAC address, checked for errors, and then (assuming that no errors exist and that the frame is not filtered) transferred to external SDRAM. The Intel XScale core client is notified of the arrival of new frames via the queue manager interface. If the learning service is activated, the frame’s source MAC address is submitted to the learning service if the frame is received without error.

## 4.3.3 ixEthAcc Component Features

The Ethernet Access component features may be divided into three areas:

- **Data Plane —** Responsible for the transmission and reception of IEE 803.2 Ethernet frames.
- **Control Plane —** Responsible for the control of the interface characteristics.
- **Management Information —** Responsible for retrieving counter and statistical information associated with the interfaces.

**Figure 5. ixEthAcc Block Diagram**



## 4.3.4 Data Plane

The data plane is responsible the transmission and reception of Ethernet frames.

### 4.3.4.1 Port Initialization

Prior to any operation being performed on a port, the appropriate microcode must be downloaded to the NPE using the ixNpeDl component.

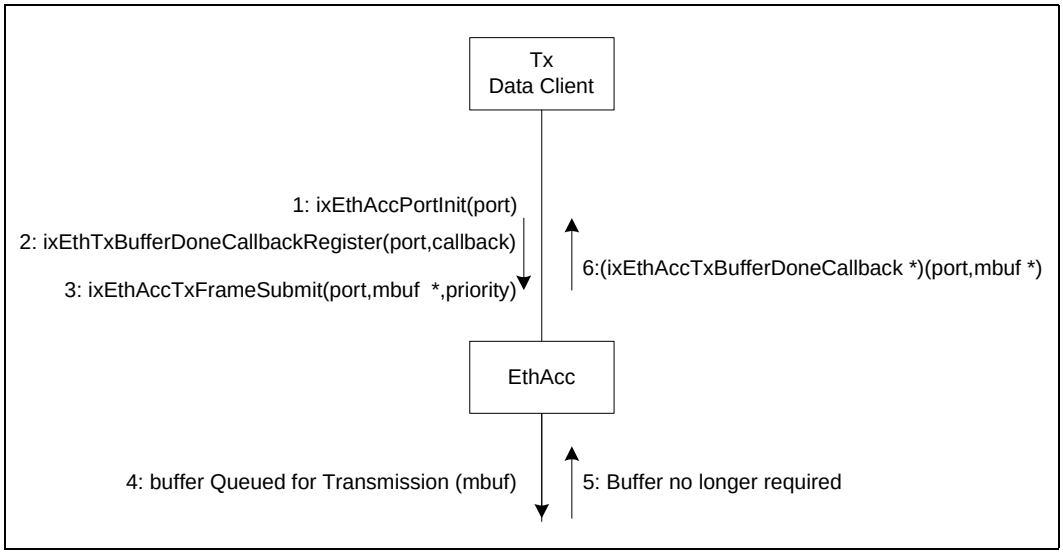
The ixEthAccPortInit() function initializes all internal data structures related to the port and checks that the port is present before initialization. The Port state shall remain disabled even after Port Init has been called. The port is enabled using the ixEthAccPortEnable() function.

### 4.3.4.2 Ethernet Frame Transmission

The Ethernet access component provides a mechanism to submit frames with a relative priority to be transmitted on a specific Ethernet MAC. Once the mBuf is no longer required by the component, it is returned from the Ethernet Access component via a free buffer callback mechanism.

The frame-transmission API is asynchronous in nature. The call is non-blocking as the transmit frame request queues the frame for transmission at a later point. There is no direct status indication as to whether the frame was successfully transmitted on the wire or not. Statistics, however, are maintained for failed transmit attempts.

**Figure 6. Ethernet Transmit Frame API Overview**



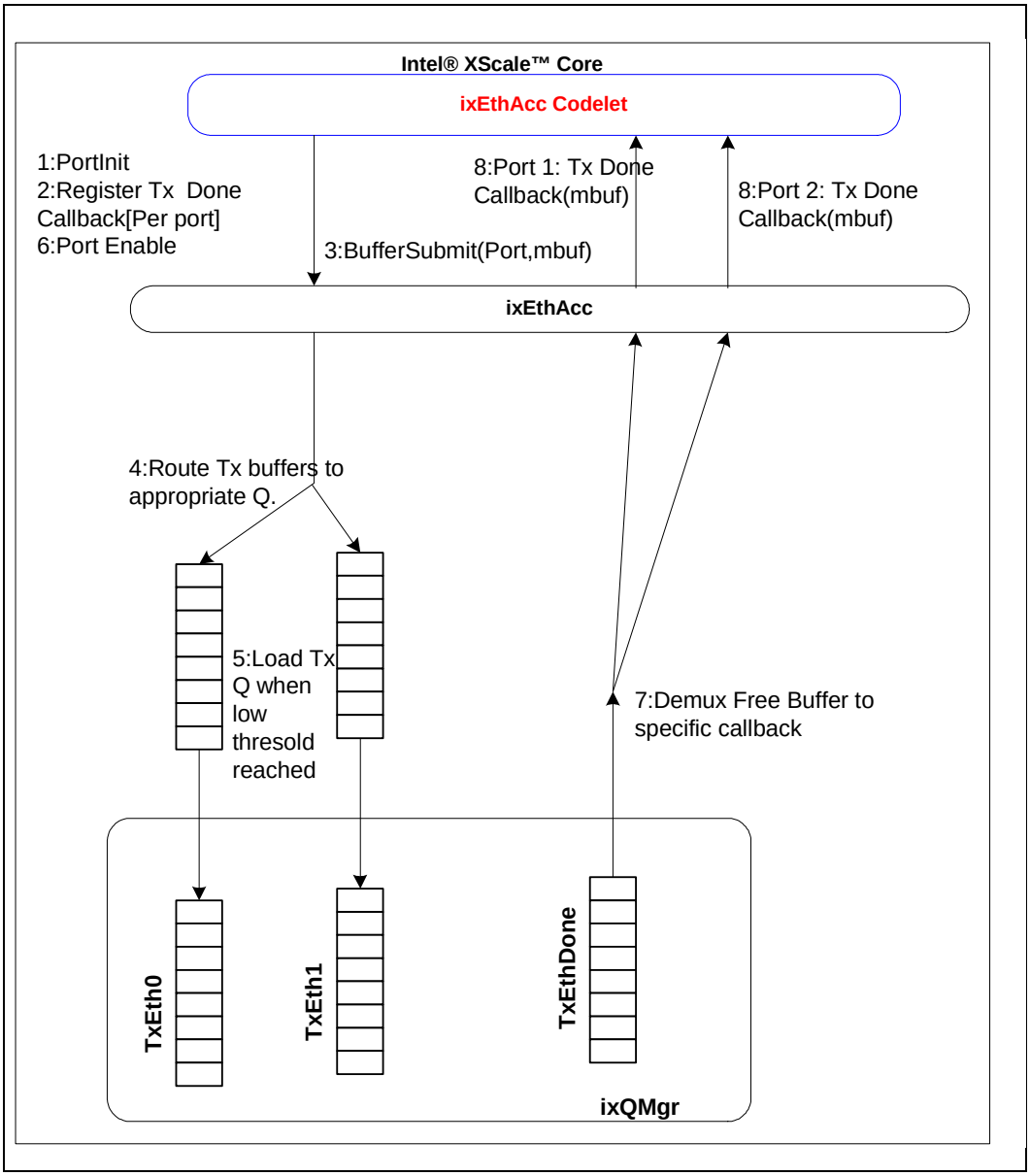

**NOTES:**
The submission of frames to be transmitted does not necessarily correspond to the callbacks to free the submitted buffers. The service shall maintain ownership of all outstanding buffers up to a limit of IX_ETHACC_MAX_OUTSTANDING_FRAMES.

Submission of chained mBuf clusters for transmission shall be supported, but excessive chaining shall have an adverse impact on performance. It is expected that chained buffers are used to add protocol headers. The payload portion of large PDUs may also use chained mBuf clusters. The suggested minimum size for the buffers within the payload portion of a packet is 64 bytes. The “transmit done” callback function is called with the head of the cluster mBuf only when the entire chain has completed transmission.

The key interface to the transmit data path NPEs is via a selection of QM queues. There are individual frame transmit queues assigned for each NPE Ethernet port. A single queue is used to multiplex buffer complete/done messages from the NPEs.

The QM queues are a maximum of 128 entries deep per port. The frame submit function must internally queue (in the ixEthAcc software) frames which are submitted in excess of a predefined limit (currently ½ max number of q entries due to fast path component requirements). For more information on the Fastpath dependency see the IX_ETH_ACC_FPATH_AWARE define in Section B.3.13. All internally queued buffers submitted for transmission but not queued to the hardware QM queue are stored in software queues. If priority FIFO queuing is being used, the frames shall be saved in individual per priority FIFOs.

**Figure 7. Ethernet Transmit Frame Data Flow**



Frames will be submitted to the port specific QM queue when a low/empty threshold is reached on the queue. Once frame transmission has completed, the buffer is placed on a buffer done QM queue. This queue contains multiplexed entries from both NPE ports. The ixEthAcc software consumes entries from this queue and returns the buffers to the client via the function previously registered by ixEthAccTxDoneCalbackRegister( ).

There are two scheduling disciplines selectable via the ixEthTxSchedulingDiscipline(). The frame submit behavior shall be different for each case. The current scheduling disciplines are **No Priority** and **Priority**.

#### 4.3.4.2.1 Tx FIFO No Priority

If the selected discipline is FIFO_NO_PRIORITY, then all frames may be directly submitted to the QM queue for that port if there is room on the port. Frames which cannot be queued in the QM queue are stored in an ixEthAcc software queue for deferred submission to the QM queue. The QM threshold in the configuration can be quite high. This allows the ixEthAcc software to burst frames into the QM queue and improve system performance due to the resultant higher cache hit rates.

#### 4.3.4.2.2 Tx FIFO Priority

If the selected discipline is FIFO_PRIORITY, then frames are queued by ixEthAcc software in separate priority queues. The threshold in the QM must be kept quite low to improve fairness among packets submitted. Once the low threshold on the QM queue is reached, frames are selected from the priority queues in strict priority order. (i.e., all frames are consumed from the highest priority queue before frames are consumed from the next lowest priority).

The priority is controlled by the IxEthAccTxPriority value in the ixEthAccPortTxFrameSubmit () function. IX_ETH_ACC_TX_PRIORITY_0 is the lowest priority submission and IX_ETH_ACC_TX_PRIORITY_7 is the highest priority submission.

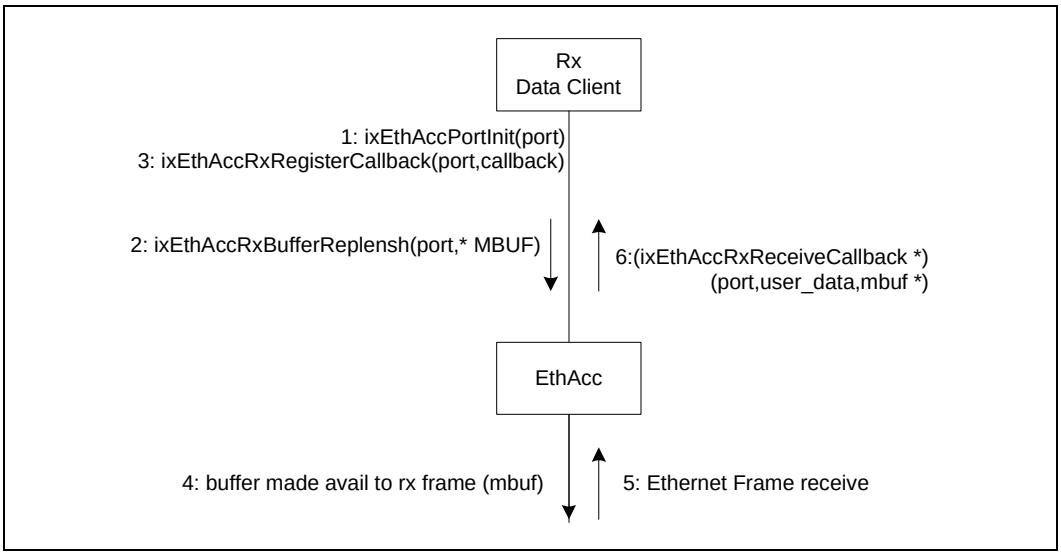
There are no fairness mechanisms applied across different priorities. Higher priority frames could starve lower-priority frames indefinitely.

### 4.3.4.3 Ethernet Frame Reception

The Ethernet access component provides a mechanism to register a callback to receive Ethernet frames from a particular MAC. The user-level callback is called for each Ethernet frame received. The Ethernet access component must be supplied with receive buffers prior to any receive activity on the Ethernet MAC. The user also must ensure that there are sufficient buffers assigned to this component to maintain wire speed Ethernet receive performance. Should a frame arrive and no buffer is available, there is no callback indication, and a rx_buffer_underrun counter is incremented.

The mBuf must contain IX_ETHACC_RX_MBUF_MIN_SIZE bytes in a single data cluster. Receive frames will not be pushed into a chained mBuf structure. Once this service calls the callback with the receive mBuf, “ownership” of the buffer is transferred to the user of the access component (i.e., the access component will not free the buffer).

**Figure 8. Ethernet Receive Frame API Overview**



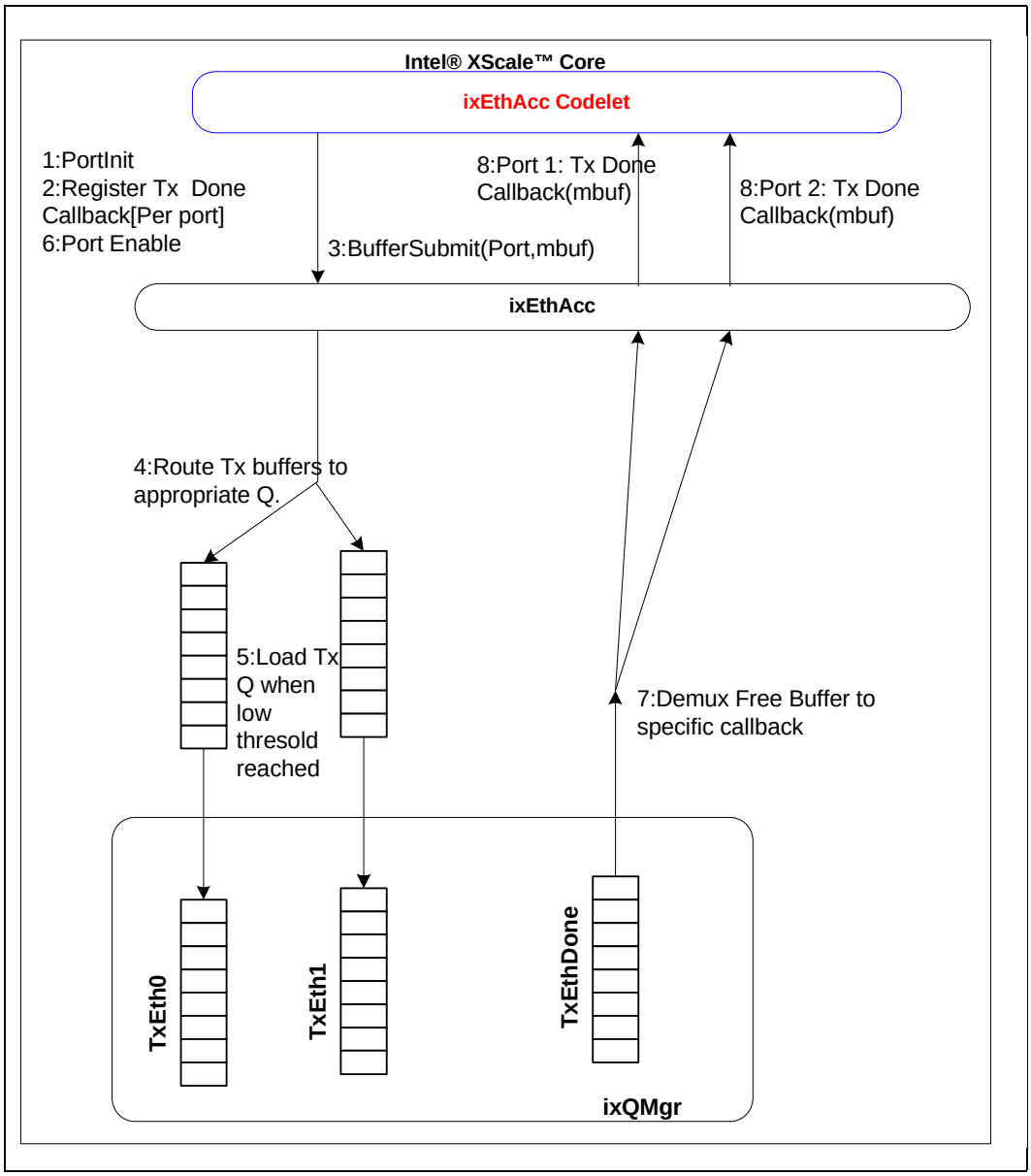

An Rx polling interface shall not be provided for the service. This can easily be extended via queuing the received frames by the access component user and subsequently providing a polling interface. Note that the receive callback is issued in frame-receive order. No receive priority mechanisms are provided.

Buffers may also be returned to the user if the user flushes the receive path via the mechanism provided.

The key interface from the NPEs to the receive data path is a selection of queues residing in the queue manager block. Received frames from both NPEs are multiplexed onto one queue manager queue (RxEthQ). The receive path s/w must de-multiplex the received frames and call the associated user level callback function registered via ixEthAccRxCallbackRegister().

The frames placed in the QM queue have already been validated to have a correct FCS. They are also free from all other types of MAC/PHY-related errors, including alignment errors and “frame too long” errors.

**Figure 9. Ethernet Receive Plane Data Flow**




The receive data plane subcomponent must provide receive buffers to the NPEs. The buffers are supplied on a per port basis via the user level interface ixEthAccRxBufferReplenish(). There are two separate free buffer QM queues allocated to providing the NPEs with receive buffers (one per port). The replenish function loads the port specific free buffer QM queue with an IX_MBUF pointer. If the port specific free buffer QM queue is full, the replenish function queues the buffer in a software queue. Once a low threshold on the specific queue is reached the software reloads the port specific free buffer queue from its software queue if available.

***Note:*** If the receive NPE does not have a receive buffer in advance of receiving an Ethernet frame, the frame will be dropped.

There is no specific port flush capability. To retrieve submitted buffers from the system, the port must be disabled. This has the result of returning all buffers (both Tx and Rx) to the user via the registered callbacks (for example, all Tx buffers submitted shall be returned via the Tx done callback).

#### 4.3.4.4 Data-Plane Endianness

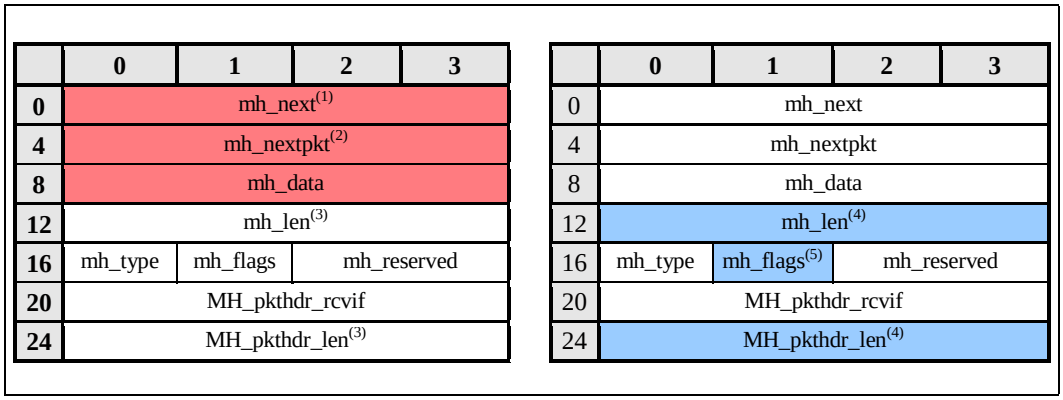
All data structures passed to the component are defined in host byte order. No changes to data structures are required in order to use the access component data path interfaces. The data pointed to by the MBUF is however expected to be in network byte order. No byte swapping takes place on the data prior to transmission to the Ethernet MAC.

#### 4.3.4.5 External Memory Requirements

The mbuf buffer descriptor is described in detail in TCP/IP Illustrated, Volume 2. The Ethernet NPE firmware expects that all such structures (i.e., m_blk structures) will be word-aligned. The NPE expects that the Intel XScale core client will always supply it (via the FreeEnet queue) with mbufs large enough to hold the largest Ethernet frame so that chaining will never need to be performed in order to buffer an entire frame. This requirement implies that the size of the mbuf data clusters to be used on the Ethernet receive path must be at least 1536 bytes, and preferably 2K bytes (which, in turn, implies that all Rx-path mbufs must have external data clusters). On the transmit side, the NPE is capable of handling chained mbufs (via the mh_next field). However, to reduce the load on NPE computational and data resources, any use of mbuf chaining should be limited to only what is absolutely necessary. The mh_nextpkt field, which facilitates the linking of multiple packets together into “queues,” is not used by the Ethernet PSM firmware; it may not be set to anything other than NULL by the Intel XScale core client (for both the receive and transmit paths).

***Note:*** If the buffer has originated in cached memory it is very important to flush the buffer prior to submission to the NPE for transmission. As the caching control of the system is a system dependency, MACROS are used to abstract this behavior to a common location within the Access Layer component code.

### Figure 10. Mbuf Fields Written by Intel® XScale™ Core (left) and NPE (right) on Ethernet Rx Path



| | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | mh_next(1) | | | |
| 4 | mh_nextpkt(2) | | | |
| 8 | mh_data | | | |
| 12 | mh_len(3) | | | |
| 16 | mh_type | mh_flags | mh_reserved | |
| 20 | MH_pkthdr_rcvif | | | |
| 24 | MH_pkthdr_len(3) | | | |

| | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | mh_next | | | |
| 4 | mh_nextpkt | | | |
| 8 | mh_data | | | |
| 12 | mh_len(4) | | | |
| 16 | mh_type | mh_flags(5) | mh_reserved | |
| 20 | MH_pkthdr_rcvif | | | |
| 24 | MH_pkthdr_len(4) | | | |

***Note:*** (1) On the Ethernet Rx path, there is an underlying assumption that no buffer chaining will ever be performed. Thus, the value of the mh_next field must always be NULL.

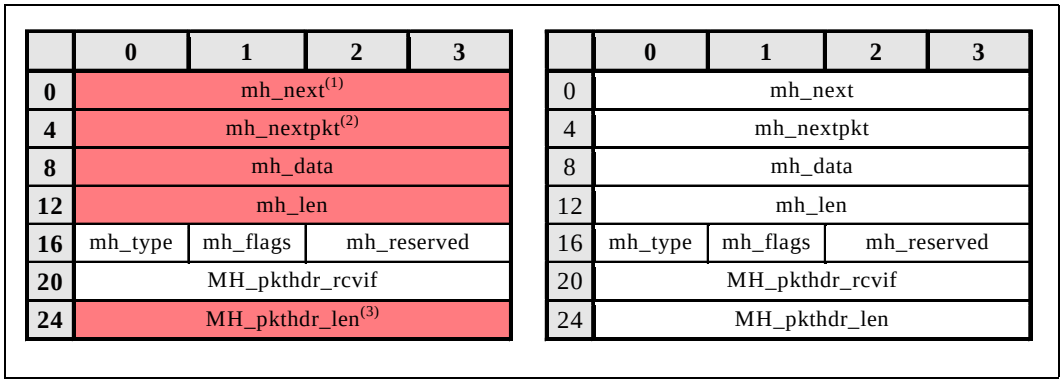
(2) All IXP42X product line NPE firmware is designed under the assumption that packet chaining will not be used. Thus, the value of the mh_nextpkt must always be NULL.

(3) While it is not required that the Intel XScale core client explicitly set the mh_len and MH_pkthdr_len fields, the Ethernet NPE firmware depends on the fact that the data cluster to which the mh_data field points is at least 1536 (and preferably 2K) bytes in length.

(4) The NPE sets both the mh_len and MH_pkthdr_len fields to the length of the received frame. The 4 bytes of the FCS field are counted toward the frame length only if the Ethernet MAC is set up to pass the FCS field through to the Ethernet coprocessor.

(5) The NPE only sets the multicast bit (bit 5) and broadcast bit (bit 4) of the mh_flags field, and only if the status word from the MAC indicates the frame is multicast and/or broadcast.

### Figure 11. Mbuf Fields Written by Intel XScale® Core (left) and NPE (right) on Ethernet Tx Path



| | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | mh_next(1) | | | |
| 4 | mh_nextpkt(2) | | | |
| 8 | mh_data | | | |
| 12 | mh_len | | | |
| 16 | mh_type | mh_flags | mh_reserved | |
| 20 | MH_pkthdr_rcvif | | | |
| 24 | MH_pkthdr_len(3) | | | |

| | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | mh_next | | | |
| 4 | mh_nextpkt | | | |
| 8 | mh_data | | | |
| 12 | mh_len | | | |
| 16 | mh_type | mh_flags | mh_reserved | |
| 20 | MH_pkthdr_rcvif | | | |
| 24 | MH_pkthdr_len | | | |

***Note:*** (1) Unlike the Ethernet Rx path, it is possible for buffers to be chained on the Ethernet Tx path. However, all FastPath Tx frames must be unchained.

(2) All Intel® IXP42X product line firmware is designed under the assumption that packet chaining will not be used. Thus, the value of the mh_nextpkt must always be NULL.

(3) In the case of a Tx frame being represented by a chain of mbufs, only the first mbuf in the chain is required to have its MH_pkthr_len field set correctly (i.e., to the length of the entire frame).

## 4.3.5 Control Plane

The control plane portion of the component covers:

- **MAC Address Learning Database Features** — All functions related to the provision and control of the Ethernet MAC learning facility of the Ethernet NPEs.
- **Ethernet MAC Control** — This portion provides functionality to provision the MAC registers with the required functionality.

### 4.3.5.1 MAC Address Learning Database Features

The NPEs provide a function whereby MAC address-source learning is performed on received (ingress) Ethernet frames. If source learning is enabled, the source MAC addresses are automatically populated in a learning database. For a frame to be filtered, there must be a filtering database entry whose MAC address matches the frame's destination MAC address and whose port ID matches that of the ingress MAC.

To minimize the unnecessary forwarding of frames, an IEEE 802.1D-compliant bridge maintains a filtering database.

***Note:*** In IXP400 software v1.2.2, the MAC Address Learning and Filtering database component was decoupled from the ixEthAcc component. This new component is called ixEthDB and provides MAC address learning/filtering database functionality that can be easily extended beyond the Ethernet NPE interfaces.

Each entry in the filtering database is composed of a MAC address and a logical port number. Whenever the bridge receives a frame, the frame is parsed to determine the destination MAC address, and the filtering database is consulted to determine the port to which the frame should be forwarded. If the destination MAC address of the frame being processed has been learned on the same interface from which it was received, it is dropped. Otherwise the frame is forwarded from the NPE to the Intel XScale core.

Although it is possible to create static entries in the filtering database via a management interface, most information in the database is created dynamically via the MAC address learning process. Whenever a bridge receives a frame through a particular port, it compares the frame's source MAC address to all addresses in that port's filtering database. If the received source MAC address does not yet exist in the database and it does not match the MAC address of the attached Ethernet port, the new MAC address is inserted. The insertion of the new MAC address is known as learning.

If a terminal (source of Ethernet traffic on the network) is moved from one NPE port to another, this component is responsible for ensuring the consistency of the learning/filtering database. The database is updated within one second of the terminal move being detected. The change is detected when traffic is first received from the terminal on the new NPE port.

An API is provided to allow the user to statically provision entries in the database. These entries are not subject to aging.

The learning/filtering database is an organized collection of MAC address information coupled with a comprehensive API and data flow event model. The main actors in this event model are the client software (accessing the database through the public API), the data path ports (NPEs), and the database itself.

Apart from common data manipulation (add, remove, search, update) the learning/filtering database model provides a clear and structured way of defining reactions to events (for example, learning and aging) and a straightforward manner of describing how ports share data and depend on each other.

The learning/filtering database was designed in consideration with these constraints:

- NPEs can share MAC information, useful for making switching decisions at an NPE level.
- Very fast search capability is required (per-packet MAC address search).
- Database must be able to extract a filtered view of entries, based on any set of port ids, and this filtered view must be represented as a balanced binary search tree to be used by NPEs or other ports (for example, PCI).
- It must be able to age entries and have static (non-removable by aging) entries.

The main learning database is stored in a hash structure, which allows maintaining a unified set of entries for all the ports and very fast search. This database is updated using one of the following mechanisms:

- Adding/removing static and dynamic entries using the public ethAcc API.
- Removing dynamic entries by aging (controlled by the ethAcc public API, see ixEthAccAgingMaintenance).
- Adding dynamic entries from NPE messages.

Following a modification to the database, a dependency list is computed to determine what trees must be reconstructed. Database views are filtered on appropriate port IDs and pushed back to the NPEs as balanced binary search trees.

Non-balanced trees pushed by NPEs for balancing are used to update the age in the main database and add any addresses not already in the main database. New trees will be extracted from the main database instead of rebalancing the given ones to ensure data integrity across ports and minimize tree exchanges between the main database and NPEs.

While each NPE has an entry limit count of 511, the main database is limited only by the size of the memory pools used to maintain the hash, which is determined at compilation time. However, for good search performance, it is not recommended to use less than eight times hash buckets compared to the expected number of entries. As it is optimal for a hash to use a prime number of buckets, it is expected to use 6421 or 12853 buckets for most cases, which allows approximately 600 and respectively 1000 entries while preserving fast search capabilities.

A high-level overview of the system is provided in Figure 12.

**Figure 12. Ethernet Receive Frame API Overview**

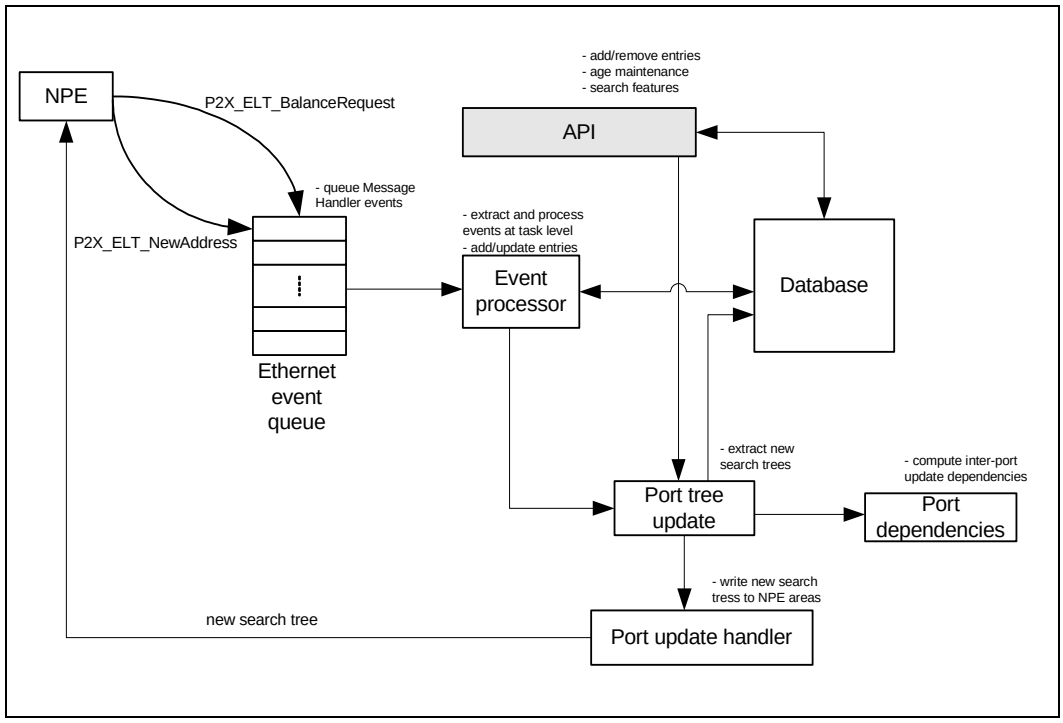



Figure 12 exemplifies the main event flows from NPEs and API, which may trigger changes to the database, which in turn determines tree updates that provides the NPEs with updated binary search trees. It should be noted that the design specifically allows for ports that are not specifically Ethernet NPEs to be integrated into the same access and update scheme. (for example, PCI)

A detailed breakdown of the learning/filtering database components is provided:

**Public API**

- Add/remove static or dynamic entries.
- Enable/disable aging.
- Search by MAC address or by port ID and MAC address.
- Maintain aging.

**NPE Services**

- Add entries signaled by P2X_ELT_NewAddress.
- Add entries signaled by P2X_ELT_BalanceRequest and update address aging.
- Provide NPEs with updated search trees.

**Hash Table Manipulation**

- Add/remove/search entries.
- Hash iteration methods.

**MAC Descriptor Handling**

- Infrastructure and support for the hash table (hashing function, address comparison, etc.).

**Binary Search Tree Manipulation**

- Tree extraction from the hash table based on port ID queries.
- Tree balancing.

**Memory Management**

- Memory pools used by the hash table, MAC descriptors, and tree nodes.
- Smart pointers for sharing data across structures.

**Event Processing**

- Event transfer between the message handler and the component internal queue.

**Port Dependency and Update Logic**

- Process port dependency rules.
- Optimal generation of learning trees.

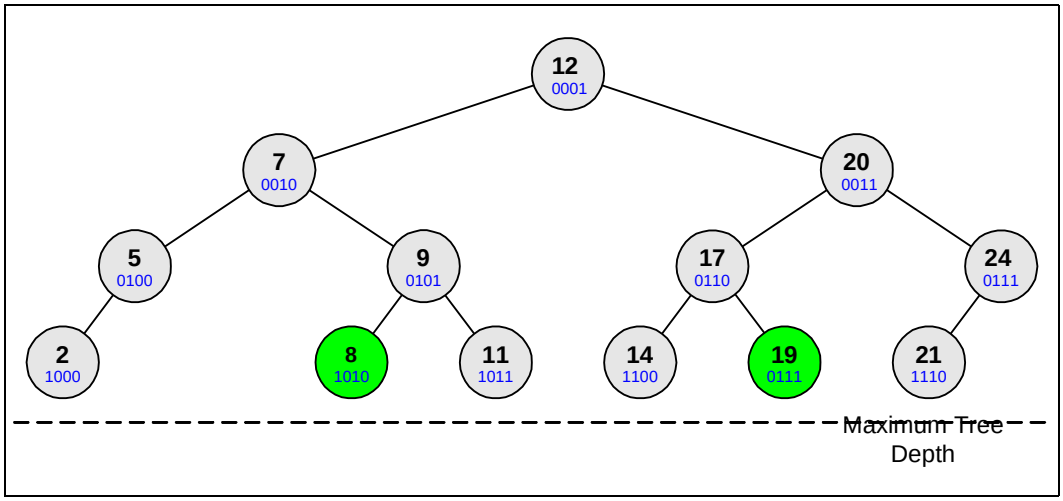
#### 4.3.5.1.1 Database Maintenance

Under certain circumstances, the learning/filtering database may require maintenance. Aging and Balancing the learning/filtering database is an example of such maintenance.

Aging is the process through which inactive MAC addresses are removed from the filtering database. At periodic intervals, the filtering database is examined to determine if any of the learned MAC addresses have become inactive during the last period (i.e., no traffic has originated from those MAC addresses for a period of roughly five minutes, as recommended by IEEE 802.1D). If so, they are removed from the filtering database.

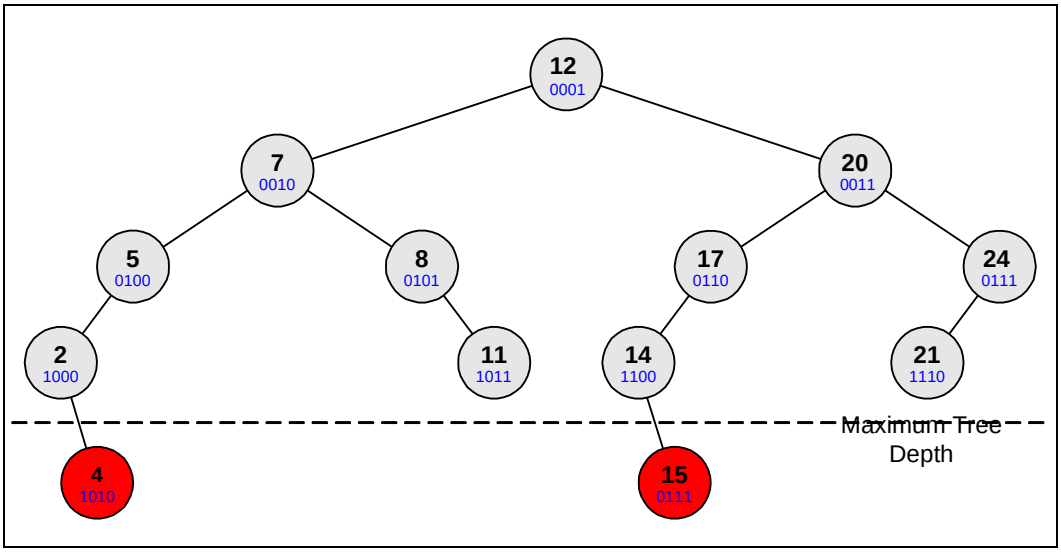
As new MAC addresses are inserted into the filtering database via the learning process, the binary tree structure used to maintain this database tends to become unbalanced.

**Figure 13. Examples of Node Insertion Not Requiring Re-Balancing**



As the tree becomes more unbalanced, the average time required for conducting a search through the tree increases. This has a negative impact on the efficiency of both the filtering and learning processes. Furthermore, if the tree becomes unbalanced — to the point that its maximum physical depth is reached along a particular branch — no new MAC addresses may be inserted on that branch. To prevent these problems, the tree must be periodically rebalanced so that the maximum depth of the populated portion of the tree varies as little as possible from one branch to another.

**Figure 14. Examples of Node Insertion Requiring Re-Balancing**




The ixEthAcc component performs all database maintenance functions. To facilitate this, the ixEthAccMaintenance() must be called with a frequency of IX_ETH_ACC_LEARNING_ENTRY_AGE_TIME.

If the maintenance function is not called, then the aging function will not run. An entry shall be aged at IX_ETH_ACC_LEARNING_ENTRY_AGE_TIME +/- IX_ETH_ACC_MAINTENANCE_TIME seconds.

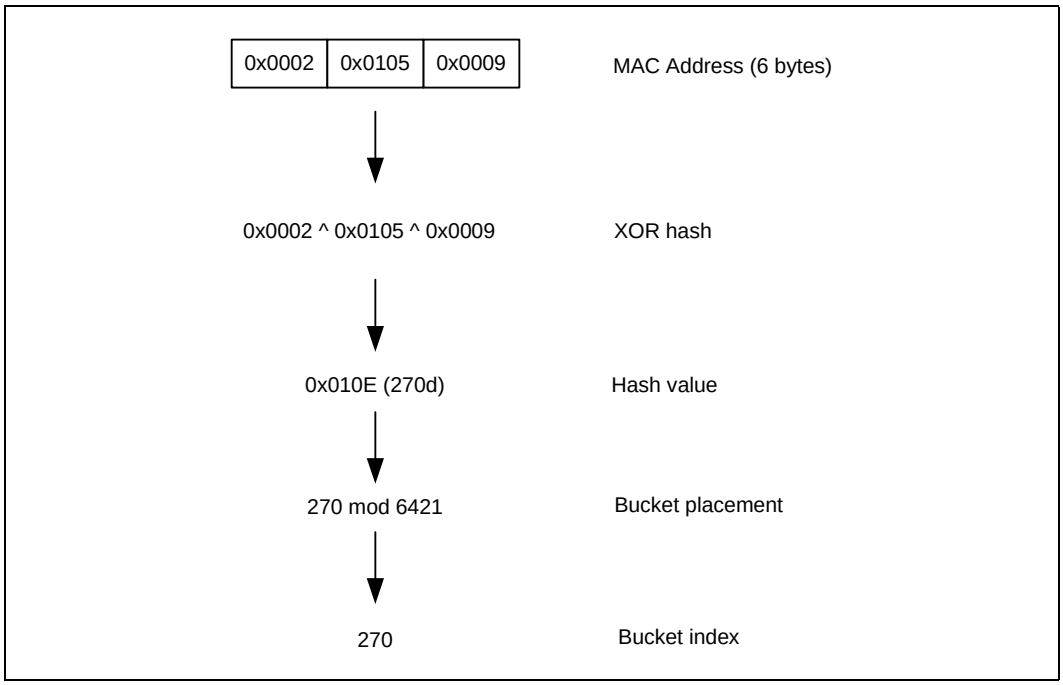
***Note:*** It is the client's responsibility to ensure the ixEthAccMaintenance() function is executed with the required frequency.

#### 4.3.5.1.2 Algorithms Used by the Ethernet Learning Tree

**Hashing**

A hash table represents the main Ethernet learning database. Entries (MAC address descriptors) will be hashed using fast XOR hashing on the three groups of 16 bytes of a MAC address, therefore rendering a good, near-random distribution. To avoid collisions (entries being placed in the same bucket), it is optimal for the hash table to have a prime number of buckets, but using a power of 2 as number of buckets will make the bucket distribution faster. Figure 15 describes the process of hashing a MAC address entry, assuming 6421 buckets are used.

Figure 15. Hashing




When two or more entries render the same bucket index value, there is a collision. Collisions are solved by linking entries within buckets. Because this makes the search within a particular bucket use linear time depending on the number of chained items, it is best to minimize collisions by using 8-10 times the expected number of entries, ideally prime number of buckets. For approximately 600 and 1000 entries, the values of 6421 and 12853 buckets (respectively) can be used with good results. The hash table uses only one pointer (4 bytes) per bucket. Therefore the memory usage is relatively small.

**Tree Balancing**

The algorithm responsible for tree rebalancing is optimal in space and time. It executes in O(n) and it requires no extra memory storage apart from the original tree. The resulting trees are route balanced and not perfectly balanced as creating the second variety is more expensive time-wise without improvements in the search tree traversal times which remains O(log(n)). The process takes place in two steps: first the tree is converted into a vine (a binary tree in which every node has at most one right child), then the vine is compacted into a balanced tree.

### 4.3.5.2 Ethernet MAC Control

The role and responsibility of this module is to enable clients to configure the Ethernet Co-Processor MACs for both NPEs. This API permits the setting and retrieval of unicast and multicast addresses, duplex mode configuration, FCS appending, and promiscuous mode configuration.

#### 4.3.5.2.1 FCS MAC Control

An API is provided to provision whether the MAC appends an IEEE-803.2 FCS to the outgoing Ethernet frame or if the data passed to the ixEthAcc component is to be transmitted without modification.

An API is also provided to provision whether the receive buffer — sent to the Intel XScale core's client — contains the frame FCS or not. The default behavior is to remove the FCS from the frame. Rx frames are still subject to FCS validity checks, and frames that fail the FCS check are dropped.

Both of these interfaces operate on a per-port basis.

#### 4.3.5.2.2 MAC Filtering (MAC Level)

The MAC is capable of operation in either promiscuous or non-promiscuous mode. An API to control the operation of the MAC is provided.

##### Promiscuous Mode

All valid Ethernet frames are forwarded to the NPE for receive processing.

##### Non-Promiscuous Mode

This allows the following frame types to be forwarded to the NPE for receive processing:

- Destination MAC address = Provisioned unicast MAC address
- Destination MAC address = Broadcast address
- Destination MAC address = Provisioned multicast MAC addresses

## 4.3.6 Management Information

The Ethernet component provides MIB II EtherObj statistics for each interface. The statistics are collected from Ethernet component counters and NPE collected statistics. Statistics are gathered for collisions, frame alignment errors, FCS errors, etc.

The statistics counters that are support by the Ethernet access component are shown in Table 18 and Table 19. For more details on these statistics object, see RFC-2665.

These APIs are provided to retrieve these statistics:

- IxEthAccStatus ixEthAccMibIIStatsGet (IxEthAccPortId portId,IxEthEthObjStats *retStats)
  This function returns the statistics maintained for a port.
- IxEthAccStatus ixEthAccMibIIStatsGetClear (IxEthAccPortId portId, IxEthEthObjStats *retStats)
  This function returns and clears the statistics maintained for a port.
- IxEthAccStatus ixEthAccMibIIStatsClear (IxEthAccPortId portId)
  This function clears the statistics maintained for a port.

**Table 18. Managed Objects for Ethernet Receive**

| Object | Increment Criteria |
|---|---|
| dot3StatsAlignmentErrors | RFC-2665 definition |
| dot3StatsFCSErrors | RFC-2665 definition |
| dot3StatsFrameTooLongs | RFC-2665 definition |
| dot3StatsInternalMacReceiveErrors | RMII_FRM_ALN_ERROR \|\| XTRA_BYTE \|\| LEN_ERR \|\| RX_LATE_COLL \|\| (MII_FRM_ALN_ERR andante!FCS_ERR) |
| LearnedEntryDiscards | Received frame dropped due to MAC destination address filtering. |

**Table 19. Managed Objects for Ethernet Transmit**

| Object | Increment Criteria |
|---|---|
| dot3StatsSingleCollisionFrames | RFC-2665 definition |
| dot3StatsMultipleCollisionFrames | RFC-2665 definition |
| dot3StatsDeferredTransmissions | RFC-2665 definition |
| dot3StatsLateCollisions | RFC-2665 definition |
| dot3StatsExcessiveCollisions | RFC-2665 definition |
| dot3StatsInternalMacTransmitErrors | RFC-2665 definition |
| dot3StatsCarrierSenseErrors | RFC-2665 definition |

## 4.3.7 Memory Requirements

### 4.3.7.1 Code Size

The code occupies approximately 70 Kbytes of code memory (text section), including static data and show-function strings.

### 4.3.7.2 Data Memory

Data memory includes the memory needed to maintain the structures needed by the algorithms and the memory required to exchange information with other components of the system. The stacks size needed is not included in the following estimations.

It is the responsibility of the user of this service to provide all buffer memory. Buffer memory of the component is likely to be in the order of:

- Rx Buffers 128 * 2,048 bytes
- Tx Buffers 128 * 2,048 bytes

This equates to 512 K per port.

**Table 20. Static Data Storage**

| Data Element | Entry Size | Entries | Total Bytes |
|---|---|---|---|
| LAN MAC Learning Tree | 4,096 | 2 | 8,192 |
| Local Statics | 4 | 32 | 128 |
| Total | | | < 10,000 |

## 4.4 IxAtmdAcc

The IxAtmdAcc software component for the IXP400 software is the ATM access-driver component, which provides a unified interface to AAL transmit and receive hardware.

While the IXP400 software is capable of supporting other ATM adaptation layers, the ixAtmdAcc component of Intel® IXP42X product line Software Release 1.0 supports AAL-5 only. IXP400 software v1.2.2 adds support for AAL-0 and OAM.

IxAtmdAcc is a component of the IXP400 software. The phrase ‘IxAtmdAcc’ stands for “ATM Data Driver Access.” This component provides an abstraction to the IXP42X product line’s ATM cell-processing hardware. It is designed to support ATM transmit and receive services for multiple ports and VCs.

This section describes the configuration, control, and transmit/receive flow of ATM PDU data through the IxAtmdAcc component.

The general principle of avoiding unnecessary copying of data for performance is adhered to in this component. The BSD-based buffering scheme is used.

Since AAL-0 is conceptually a raw cell service, the concept of an AAL-0 PDU can be somewhat misleading. In the context of IXP400 software v1.2.2, an AAL-0 PDU is defined as containing an integral number of 48-byte (cell payload only) or 52-byte (cell payload and cell header without HEC field) cells.

### 4.4.1 IxAtmdAcc Component Features

The services offered by the ixAtmdAcc component are:

- Support the configuration and activation of up to 12 ports on the UTOPIA Level 2 interface.
- Support for AAL-5 CPCS PDUs transmission service, which accepts fully formed PDUs for transmission on a particular port and VC. AAL-5 CRC calculation is performed by hardware.(PDUs may be comprised of single or chained ix_mbufs.)
- Support for AAL-0-48 PDU transmission service, which accepts PDUs containing an integral number of 48-byte cells for transmission on a particular port and VC. (PDUs may be comprised of single or chained ix_mbufs.)
- Support for AAL-0-52 PDU transmission service, which accepts PDUs containing an integral number of 52-byte cells for transmission on a particular port and VC. (PDUs may be comprised of single or chained ix_mbufs.)
- Support for OAM PDU transmission service, which accepts PDUs containing an integral number of 52-byte OAM cells for transmission on a particular port independent of the VC. (PDUs may be comprised of single or chained ix_mbufs.)
- Support for ATM traffic shaping
  - — Scheduler Registration: Allows registration of ATM traffic-shaping entities on a per-ATM-port basis. A registered scheduler must be capable of accepting per-VC-cell demand notifications from AtmdAcc.
  - — Transmission Control: Allows ATM traffic-shaping entities to determine when cells are sent and the number of cells sent from each VC at a time.
- Support for setting or viewing the CLP for AAL-5 CPCS SARed PDUs.
- Support for setting the transmit CLP CUP in all cells of an AAL-0-48 PDU.
- Support for the client to set the transmit GFC, PTI, or CLP in any cell of an AAL-0-52/OAM PDU.

  IxAtmdAcc does not process cell headers for AAL-0-52/OAM, thus GFC, PTI, and CLP must be set in the cell headers in the PDU by the client.(note the HEC is not included)
- Supports delivery of fully formed AAL-5 CPCS PDUs received on a particular port and VC with error detection for CRC errors, priority queuing, and corrupt packet delivery.

  (PDUs may be comprised of single or chained ix_mbufs).
- Support delivery of AAL-0 PDU containing 48-byte cells (with good HEC), received on a particular port and VC.
- Support delivery of AAL-0 PDU containing 52-byte cells, received on a particular port and VC.
- Support delivery of an OAM PDU containing a single, 52-byte OAM cell (with good HEC, and good CRC-10), received on any port and any VC.
- For all client service types, allow the client to determine the port on which the PDU was received
- Support the ability to the view the receive CLP of an AAL-0-48 PDU (logical OR of the CLP value in each cell contained in the PDU).
- Allow the client to view the GFC, PTI, or CLP of any cell in a received AAL-0-52/OAM PDU.

  The component does not process cell headers for AAL-0-52/OAM, CLP may be read from the header cells in the PDU by the client.

- Support for up to 32 VCC channels for transmit and up to 32 channels for AAL-0/AAL-5 receive services. One client per channel is supported.
- Support one dedicated OAM transmit channel (OAM-VC) per port. This channel supports transmission of OAM cells on any VC.
- Support one dedicated OAM receive channel (OAM-VC) for all ports. This channel supports reception of OAM cells from any port on any VC.
- Provide an interface to retrieve statistics unavailable at the client layer.

  These statistics include the number of cells received, the number of cells receive with an incorrect cell size, the number of cells containing parity errors, the number of cells containing HEC errors, and the number of idle cells received.
- Provides an interface to use either a threshold mechanism, which allows the client actions to be driven by events, or a polling mechanism by which the client decides where and when to invoke the functions of the interface.
- Support for fast-path-exception packet processing.
- Support the possibility to be used in a complete user environment, or to run in a complete-interrupt environment, or a mixture of both.

  This is done by providing the control over the rx and txDone dispatch functions, transmit and replenish functions. The user may trigger them from interrupts, or poll them, or both, assuming an exclusion mechanism is provided as needed.

The ixAtmdAcc component communicates with the NPEs ATM over UTOPIA component through entries placed on Queue Manager queues, mbufs and associated descriptors located in external memory, and through the message bus interface.

## 4.4.2 Configuration Services

IxAtmdAcc supports three configuration services:

- UTOPIA port configuration
- ATM traffic shaping
- VC configuration

### 4.4.2.1 UTOPIA Port Configuration Service

The UTOPIA is the IXP42X product line processors' interface by which ATM cells are sent to and received from external PHYs. In order to configure the UTIOPIA interface, IxAtmdAcc provides an interface that allows a configuration structure to be sent to and/or retrieved from the UTIOPIA interface.

IxAtmdAcc provides the interface to configure the hardware, and enable/disable traffic on a per-port basis.

### 4.4.2.2 ATM Traffic Shaping Services

An ATM scheduling entity provides a mechanism where VC traffic on a port is shaped in accordance with its traffic parameters. IxAtmdAcc itself does not provide such a traffic-shaping service, but can be used in conjunction with external scheduling services.

The scheduler registration interface allows registration of ATM traffic-shaping entities on a per-port basis. These entities, or proxies thereof, are expected to support the following callbacks on their API:

- Function to exchange VC identifiers.

  A VC identifier identifies a port, VPI, and VCI and is usually specific to layer interface. IxAtmdAcc has an identifier known as a connId and the scheduling entity is expected to have its own identifier known as a scheduler VcId. This callback also serves to allow the scheduling entity to acknowledge the presence of VC.

- Function to submit a cell count to the scheduling entity on a per-VC basis.

  This function is used every time the user is submitting a new PDU for transmission.

- Function to clear the cell count related to a particular VC.

  This function is used during a disconnect to stop the scheduling services for a VC.

No locking or mutual exclusion is provided by the IxAtmdAcc component over these registered functions.

The transmission control API expects to be called with an updated transmit schedule table on a regular basis for each port. This table contains the overall number of cells, the number of idle cells to transmit, and for each VC) the number of cells to transmit to the designated ATM port.

The ATM Scheduler can be different for each logical port and the choice of the ATM scheduler is a client decision. ATM scheduler registrations should be done before enabling traffic on the corresponding port. Once registered, a scheduler cannot be unregistered. If no ATM scheduler is registered for one port, transmission for this port is done immediately.

#### 4.4.2.3 VC Configuration Services

IxAtmdAcc provides an interface for registering VCs in both Tx and Rx directions. The ATM VC is identified by a logical PHY port, an ATM VPI, and an ATM VCI. The total number of ATM AAL-5 or AAL-0 VCs supported on all ports in both directions is 32. IxAtmdAcc supports up to 32 Rx channels, and up to 32 Tx channels on all ports. For AAL-5 and AAL-0, the number of logical clients supported per-VC is one.

In addition to the 32 VCs mentioned above, one dedicated OAM transmit VC per port and one dedicated OAM receive VC are supported. These dedicated OAM VCs behave like an “OAM interface” for the OAM client, and are used to carry OAM cells for any VPI/VCI (even if that VPI/VCI is one of the 32 connected for AAL services).

In the Tx direction, the client has to register the ATM traffic characteristics to the ATM scheduler before invoking the IxAtmDAcc “connect” function. The TxVcConnect function does the following actions:

- Checks if the PHY port is enabled.
- Checks for ATM VC already in use in an other TX connection.
- Checks if the service type is OAM and, if so, checks that the VC is the dedicated OAM-VC for that port.
- Checks the registration of this VC to the registered ATM scheduler.
- Binds the VC with the scheduler associated with this port.
- Registers the callback by which transmitted buffers get recycled.

- Registers the notification callback by which the hardware will ask for more data to transmit.
- Allocates a connection ID and return it to the client.

In the Rx directions, the RxVcConnect steps involve the following actions:

- Check if the PHY port is enabled.
- Check for ATM VC already in use in an other RX connection.
- Check if the service type is OAM and, if so, check that the VC is the dedicated OAM-VC.
- Register the callback by which received buffers get pushed into the client's protocol stack.
- Register the notification callback by which the hardware will ask for more available buffers.
- Allocate a connection ID and return it to the client.

When connecting, a connection ID is allocated and it must be used to identify the VC in all calls to the API. The connection IDs for Receive and Transmit on the same ATM VC are different.

The client has the choice to use a threshold mechanism provided by IxAtmdAcc or to poll the different resources. When using the threshold mechanism, the client needs to register a callback function and supply a threshold level. As a general rule, when configuring threshold values for different services, the lower the threshold value is, the higher the interrupt rate will be.

### 4.4.3 Transmission Services

The IxAtmAcc transmit service currently supports AAL-5, AAL-0-48, AAL-0-52, and OAM only, and operates in scheduled mode.

In scheduled mode, buffers are accepted and internally queued in IxAtmdAcc until they are scheduled for transmission by a scheduling entity. The scheduling entity determines the number cells to be transmitted from a buffer at a time, this allows cells from different VCs to be interleaved on the wire.

AtmdAcc accepts outbound ATM payload data for a particular VC from its client in the form of chained ix_mbufs. For AAL-5 an ix_mbuf chain represents an AAL-5 PDU which can contain 0-65535 payload octets. A PDU is, however, a multiple of 48 octets, when padding and the AAL-5 trailer are included. For AAL-0-48/AAL-0-52/OAM an ix_mbuf chain represents a PDU where the maximum length is limited to 256 chained ix_mbufs and/or 65,535 octets.

The submission rate of buffers for transmission should be based on the traffic contract for the particular VC and is not known to IxAtmdAcc. However, there will be a maximum number of buffers that IxAtmdAcc can hold at a time and a maximum number of buffers that the underlying hardware can hold before and during transmission. This maximum is guaranteed to facilitate the port rate saturation at 64-byte packets.

Under the ATM Scheduler control (scheduled mode), IxAtmdAcc interprets the schedule table and builds and sends requests to the underlying hardware. For AAL-5/AAL-0-48 these will be segmented into 48-byte cell payloads and transmitted with ATM cell headers over the UTOPIA bus. For AAL-0-52/OAM these will be segmented into 52-byte cells, HEC added, and transmitted "as is" over the UTOPIA bus.

Once the transmission is complete, IxAtmdAcc passes back the mbufs to its client (on a per-connection basis). The client can free them or return them to the pool of buffers. The preferred option is to reuse the buffers during the next transmission. Processing of transmit done buffers from IxAtmdAcc is controlled by the client.

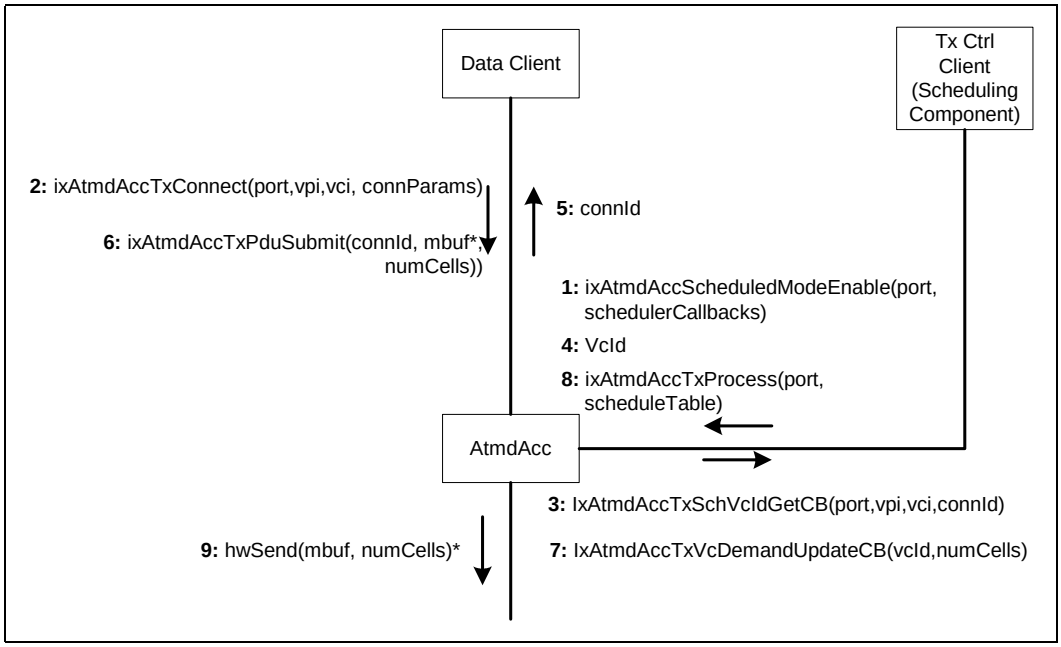
Transmit Done is a system-wide entity which provides a service to multiple ports. A system using multiple ports with very different transmit activity results in latency effects for low-activity ports. The user needs to tune the number of buffers needed to service a low-rate port or channel if the overall user application involves a port configured with a VC supporting a very different traffic rate. This tuning is at the client's discretion and hence is beyond the scope of this document

In the case of OAM, a PDU containing OAM cells for any port, VPI, or VCI must be submitted for transmission on the dedicated OAM-VC for that port. This is true regardless of whether an AAL-5/ AAL-0-48/AAL-0-52 transmit service connection exists for the given VPI, VCI. The dedicated OAM-VC will be scheduled just like any other VC.

#### 4.4.3.1 Scheduled Transmission

The scheduling entity controls the VC from which cells are transmitted and when they are transmitted. Buffers on each VC are always sent in the sequence they are submitted to IxAtmdAcc. However, cells from different VCs can be interleaved. Figure 16 shows VC connection and buffer transmission for a scheduled port.

**Figure 16. Buffer Transmission for a Scheduled Port**




1. A control client wants to use an ATM traffic shaping entity that will control the transmission of cells on a particular port, ensuring VCs on that port conform to their traffic descriptor values. The client therefore calls ixAtmdAccScheduledModeEnable() passing the port and some callback functions as parameters. IxAtmdAcc has no client connections active for that port and accepts the scheduler registration.

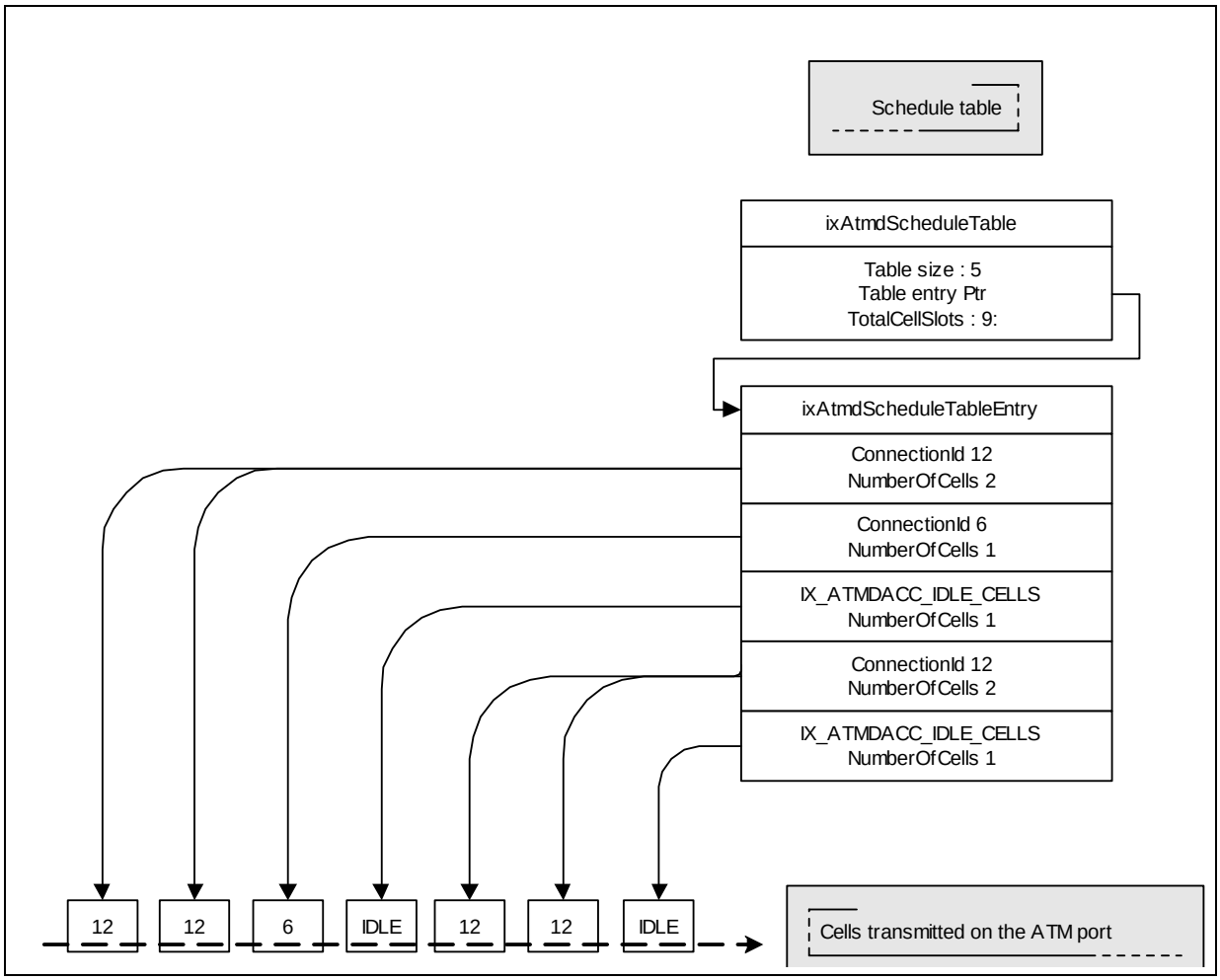
2. Later a data client wants to use the IxAtmdAcc AAL-5/AAL-0-48/AAL-0-52/OAM transmit service for a VC on the same port, and therefore calls ixAtmdAccTxVcConnect(). Note that in the case of the OAM transmit service, the connection will be on the dedicated OAM-VC for that port.
3. IxAtmdAcc calls the IxAtmdAccTxSchVcIdGetCallback () callback registered for the port. By making this call, IxAtmdAcc is asking the traffic shaping entity if it is OK to allow traffic on this VC. In making this callback, ixAtmdAcc is also providing the AtmScheduler VC identifier that should be used when calling IxAtmdAcc for this VC.
4. The shaping entity acknowledges the validity of the VC, stores the IxAtmdAcc connection ID and issues a VcId to IxAtmdAcc.
5. IxAtmdAcc accepts the connection request from the data client and returns a connection ID to be used by the client in further IxAtmdAcc API calls for that VC.
6. Sometime later, the data client has a fully formed AAL-5/AAL-0-48/AAL-0-52/OAM PDU in an ix_mbuf ready for transmission. The client calls ixAtmdAccTxPduSubmit() passing the ix_mbuf and numbers of cells contained in the chained ix_mbuf as parameters. Note:
   — In the case of AAL 5, the CRC in the AAL-5 trailer does not have to be pre-calculated.
   — In the case of OAM, the CRC 10 does not have to be pre-calculated.
7. IxAtmdAcc ensures the connection is valid and submits new demand in cells to the shaping entity by calling ixDemandUpdateCallback() callback. The shaping entity accepts the demand and IxAtmdAcc internally enqueues the ix_mbufs for later transmission.
8. The traffic-shaping entity decides at certain time, either by its own timer mechanism or by using the “Tx Low Notification” service provided by IxAtmdAcc component for this port, that cells should be transmitted on the port based on the demand it has previously obtained from AtmdAcc. It creates a transmit schedule table and passes it to the IxAtmdAcc by calling ixAtmdAccTxProcess().
9. IxAtmdAcc takes the schedule, interprets it, and sends scheduled cells to the hardware. In the case of hardware queue being full (only possible if the “Tx Low Notification” service is not used), the ixAtmdAccTxProcess call returns an overloaded status so that the traffic shaping entity can retry this again later.

#### 4.4.3.1.1 Schedule Table Description

IxAtmAcc uses a schedule table when transmitting cell information to the hardware. This schedule table drives the traffic on one port.

The schedule table is composed of an array of table entries, each of which specifies a ConnectionID and a number of cells (up to 16) to transmit from that VC. Idle cells are inserted in the table with the ConnectionID identifier set to IX_ATMDACC_IDLE_CELLS. Figure 17 shows how this table is translated into an ordered sequence of cells transmitted to one ATM port.

Figure 17. IxAtmdAccScheduleTable Structure and Order Of ATM Cell




## 4.4.3.2 Transmission Triggers (Tx Low Notification)

In Scheduled Mode, the rate and exact point at which the ixAtmdAccTxProcess() interface should be called by the shaping entity is at the clients discretion and hence is beyond the scope of this document.

However, ixAtmdAcc transmit service does provide a Tx Low Notification service which can be configured to execute a client supplied notification callback when the number of cells not yet transmitted by the hardware reaches a certain low level. The service only supports a single client per port and the maximum default cell threshold is 8 cells.

### 4.4.3.2.1 Transmit Done Processing

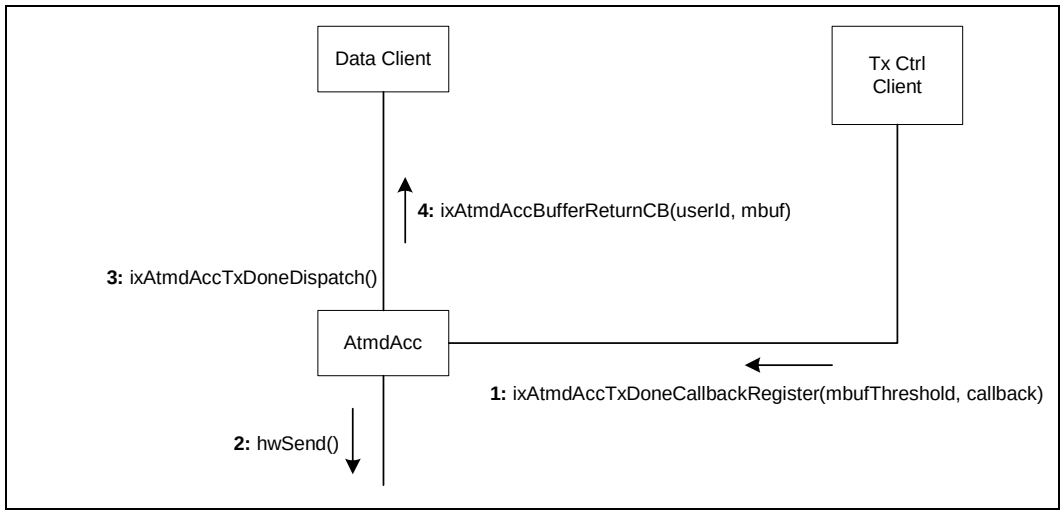
When buffers have been sent on a port they are placed in a single transmit complete stream, which is common to all ports. IxAtmdAcc does not autonomously process this stream, but instead the client decides when and how many buffers will be processed.

Processing primarily involves handing back ownership of buffers to clients. The rate at which this is done must be sufficient to ensure that client buffer starvation does not occur. The details of the exact rate at which this must be done is implementation dependent and not within the scope of this document. Because the TX Done resource is a system wide resource, it is important to note that failing to poll will cause transmission to be suspended on all ports.

### Transmit Done Based on a Threshold Level

IxAtmdAcc does provide a notification service whereby a client can choose to be notified when the number of outstanding buffers in the transmit done stream has reached a configurable threshold, as shown in Figure 18.

**Figure 18. TX Done Recycling Using a Threshold Level**

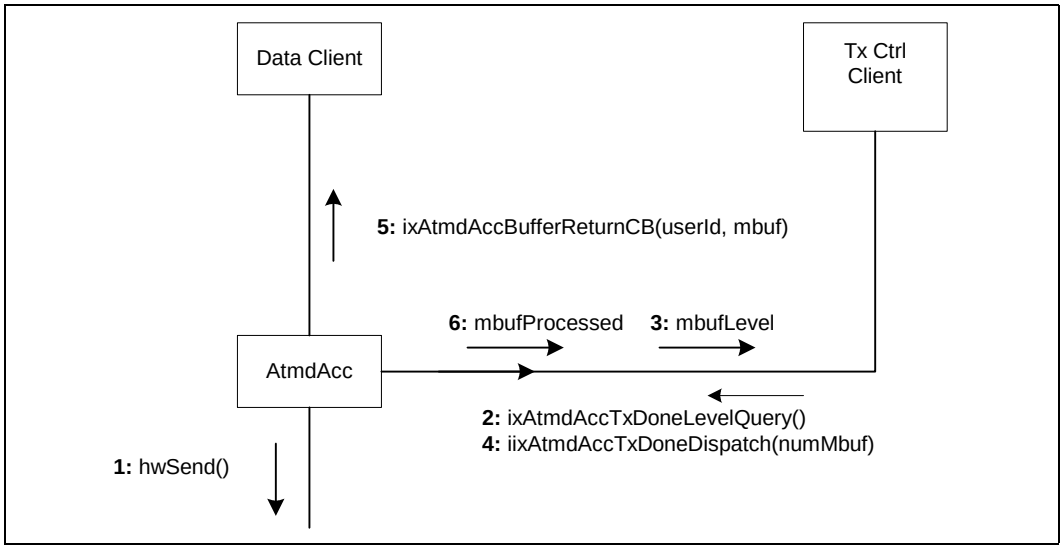



1. The control client wants to use the threshold services to process the transmitted buffers. The ixAtmdAccTxDoneCallbackRegister() function is called to set a buffer threshold level and register a callback. IxAtmdAcc provides the function ixAtmdAccTxDoneDispatch() to be used by the control client. This function itself can be used directly as the callback. IxAtmdAccTxDoneCallbackRegister allows the client to register its own callback. From this callback the IxAtmdAccTxDoneDispatch() function must be called. An algorithm can also be used to decide the number of ix_mbufs to service, depending on system load or any other constraint.
2. Sometimes earlier, the data client sent data to transmit. Cells are now sent over the UTOPIA interface and the ix_mbufs are now available.
3. At a certain point in time, the threshold level of available buffers is reached and the control client's callback is invoked by IxAtmAcc. In response to this callback, the control client calls ixAtmdAccTxDoneDispatcher(). This function gets the transmitted buffer and retrieves the connId associated with this buffer.
4. Based on connId, ixAtmdAccTxDoneDispatcher identifies the data client to whom this buffer belongs. The corresponding data client's TxDoneCallback function, as registered during a TxVcConnect, is invoked with the ix_mbuf. This TxDoneCallback function is likely to free or recycle the ix_mbuf.

### Transmit Done Based on Polling Mechanism

A polling mechanism can be used instead of the threshold service to trigger the recycling of the transmitted buffers, as shown in Figure 19.

**Figure 19. TX Done Recycling Using a Polling Mechanism**




1. Sometime earlier, the data client sent data to transmit. Cells are now sent over the UTOPIA interface and the ix_mbufs are now available.

2, 3. A control client does not want to use the threshold services to process the transmitted buffers. Therefore, the ixAtmdAccTxDoneQueryLevel() function can optionally be called to get the current number of ix_mbufs already transmitted.

4. The control client requests the ixAtmdAcc to do more processing and provides a number of buffers to process as a parameter of the ixAtmdAccTxDoneDispatch() function. This function gets the transmitted buffer and retrieves the connId associated with this buffer.

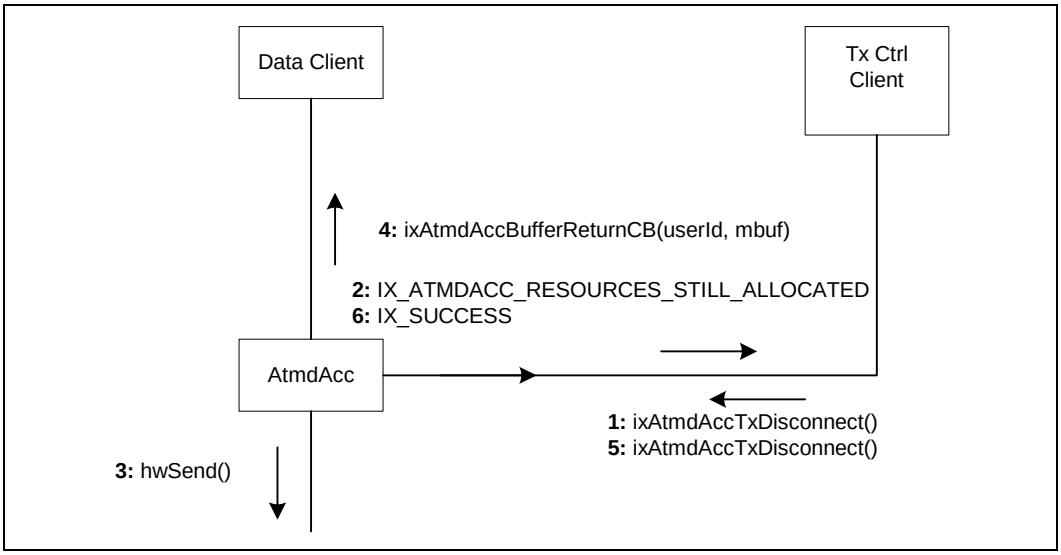
5. Based on connId, ixAtmdAccTxDoneDispatch identifies the data client to whom this buffer belongs. The corresponding data client's TxDoneCallback function, as registered during a TxVcConnect, is invoked with the ix_mbuf. This TxDoneCallback function is likely to free or recycle the chained ix_mbufs.

6. The client gets the number of buffer processed from the control client. This number may be different to the number requested when multiple instances of the ixAtmdAccTxDoneDispatch() function are used at the same time.

#### 4.4.3.2.2 Transmit Disconnect

Before a client disconnects from a VC, all resources must have been recycled, as shown in Figure 20. This is done by calling the ixAtmdAccTxVcDisconnect() function until all PDUs are transmitted by the hardware and all buffers are sent back to the client.

**Figure 20. TX Disconnect**




1. The Data Client sends the last PDUs and the Control Client wants to disconnect the VC. IxAtmdAccTxVcDisconnect() invalidates further attempts to transmit more PDUs. Any call to ixAtmdAccPduSubmit() will fail for this VC.
2. If there are resources still in use, the IxAtmdAccTxVcDisconnect() functions returns IX_ATMDACC_RESOURCES_STILL_ALLOCATED. This means that the hardware has not finished transmitting and there are still ix_mbufs pending transmission, or ix_mbufs in the TxDone stream.

3,4. Transmission of remaining traffic is running (no new traffic is accepted through ixAtmdAccPduSubmit())

5. The client waits a certain delay, depending on the TX rate for this VC, and asks again to disconnect the VC.
6. There are no resources still in use, the IxAtmdAccTxVcDisconnect() functions returns IX_SUCCESS. This means that the hardware did finish transmitting all cells and there are no ix_mbufs either pending transmission or in the txDone stream.

#### 4.4.3.3 Receive Services

IxAtmdAcc processes inbound AAL payload data for individual VCs, received in ix_mbufs. In the case of AAL 5, ix_mbufs may be chained. In the case of AAL 0-48/52/OAM, chaining of ix_mbufs is not supported. In the case of OAM, an ix_mbuf contains only a single cell.

In the case of AAL 0, Rx cells are accumulated into an ix_mbuf under supervision of an Rx timer. The ix_mbuf is passed to the client when either the ix_mbuf is passed to the client when either the ix_mbuf is filled or when the timer expires. The Rx timer is implemented by the NPE-A.

In order to receive a PDU, the client layer must allocate ix_mbufs and pass their ownership to the IxAtmdAcc component. This process is known as replenishment. Such buffers are filled out with cell payload. Complete PDUs are passed to the client. In the case of AAL 5, an indication about the validity of the PDU — and the validity of the AAL-5 CRC — is passed to the client.

In the case of AAL 0, PDU completion occurs either when an ix_mbuf is filled, or is controlled by a timer expiration. The client is able determine this by the fact that the ix_mbuf will not be completely filled in the case that completion was due to a timer expiring.

Refer to the API for details about the AAL-0 timer.

IxAtmdAcc supports prioritization of inbound traffic queuing by providing two separate receive streams. The algorithms and tuning required to service these streams can be different so management of latency and other priority constraints on receive VCs is allowed. As an example, one stream can be used for critical-time traffic (e.g., voice) and the other stream for data traffic.

The streams can be serviced in two ways:

- Setting a threshold level (when there is data available)
- Polling mechanism

Both mechanisms pass buffers to the client through a callback. Once the client is finished processing the buffer, it can either ask to replenish the channel with available buffers or free the buffer back directly to the operating-system pool.

##### 4.4.3.3.1 Receive Triggers (Rx-Free-Low Notification)

IxAtmdAcc receive service does provide a Rx-free-low notification service that can be configured to execute a client supplied notification callback when the number of available buffers reaches a certain low level. The service is supported on a per-VC basis and the maximum threshold level is 16 unchained ix_mbufs.

##### 4.4.3.3.2 Receive Processing

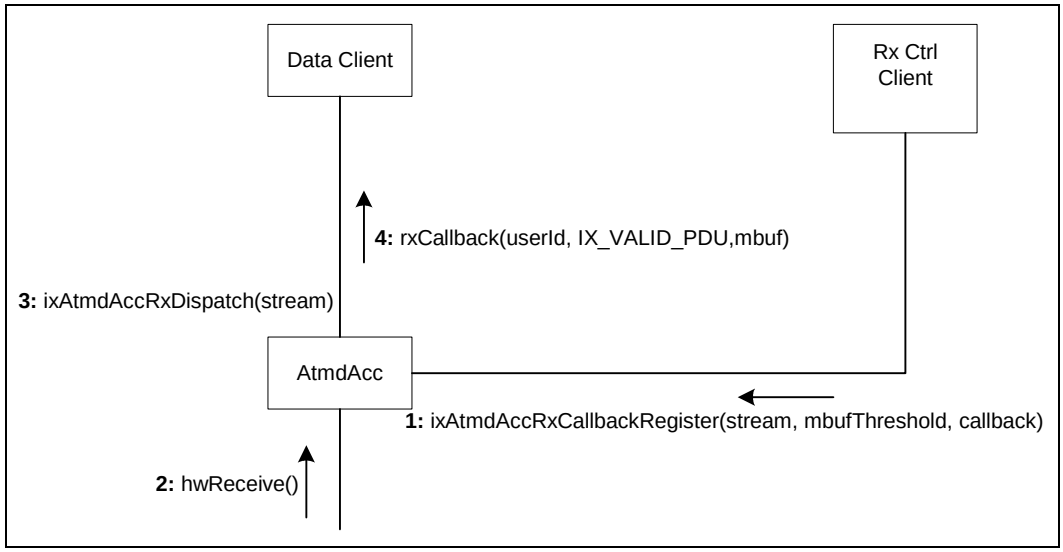
When buffers have been received on a port, they are placed in one of two Rx streams common to the VCs sharing this resource as decided by the client when establishing a connection. IxAtmdAcc does not autonomously process this stream, but instead the client decides when and how many buffers will be processed.

Processing primarily involves handing back ownership of buffers to clients. The rate at which this is done must be sufficient to ensure that client requirements in terms of latency are met. The details of the exact rate at which this must be done is implementation dependent and not within the scope of this document.

### Receive Based on a Threshold Level

IxAtmdAcc provides a notification service where a client can choose to be notified when incoming PDUs are ready in a receive stream as shown in Figure 21.

**Figure 21. RX Using a Threshold Level**

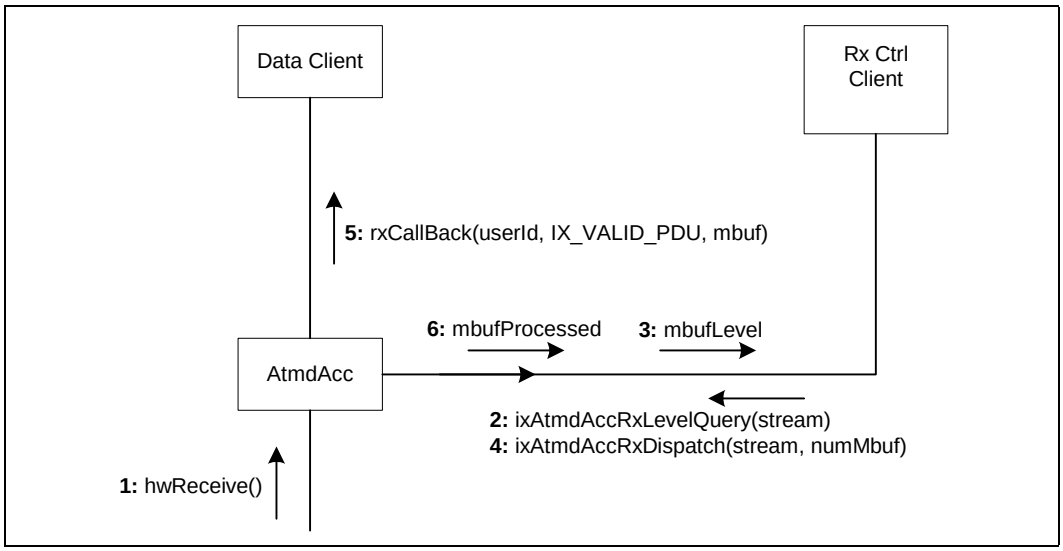



1. A control client wants to use the threshold services to process the received PDUs. The ixAtmdAccRxThresholdSet() function is called to register a callback. IxAtmdAcc provides the ixAtmdAccRxDispatch() function to be used by this callback. This function itself can be used directly as the callback. IxAtmdAccRxThresholdSet allows the client to register its own callback. From this callback (where an algorithm can be used to decide the number of mbufs to service, depending on system load or any user constraint), the user has to call the IxAtmdAccRxDispatch() function.
2. Cells are now received over the UTOPIA interface and there is a PDU available.
3. When a complete PDU is received, the callback is invoked and the function ixAtmdAccRxDispatch() runs. This function interates through the received buffers and retrieve the connId associated with each buffer.
4. Based on connId, ixAtmdAccRxDispatch identified the data client to whom this buffer belongs. The corresponding data client's RxCallback function, as registered during a RxVcConnect, is invoked with the first ix_mbuf of a PDU. This RxCallback function is likely to push the received information to the protocol stack, and then to free or recycle the ix_mbufs. The RxCallback will be invoked once per PDU. If there are many PDUs related to the same VC, the RxCallback will be called many times.

### Received Based on a Polling Mechanism

A polling mechanism can also be used to collect received buffers as shown in Figure 22.

**Figure 22. RX Using a Polling Mechanism**

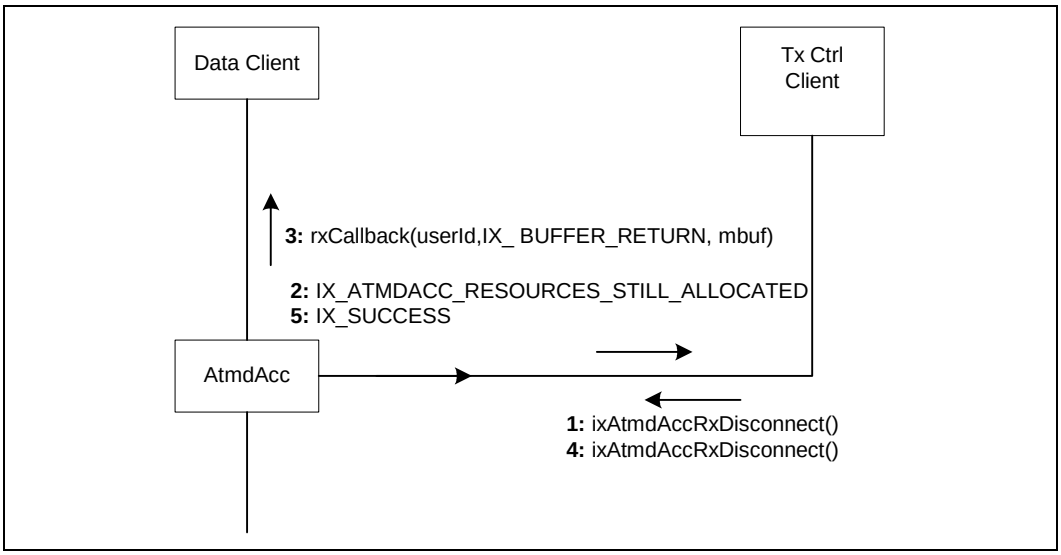



1. Cells are now received over the UTOPIA interface and a complete PDU is now available.
2,3. The control client does not want to use the threshold services. Therefore, the client can optionally query the current number of PDUs already received in one of the receive streams, using the ixAtmdAccRxLevelQuery() function.
4. The control client asks IxAtmdAcc to process an amount of PDUs from one of the streams using the function ixAtmdAccTxDoneDispatch().
5. IxAtmdAcc get the requested number of PDUs from the underlying hardware. Based on connId, ixAtmdAccRxDispatch() identifies the data clients to whom the buffers belong. The corresponding data client's RxCallback functions, as registered during a ixAtmdAccRxVcConnect, is invoked with the first ix_mbuf a PDU. This RxCallback function is likely to push the received information to the protocol stack, and then to free or recycle the ix_mbufs. The RxCallback will be invoked once per PDU. If there are many PDUs related to the same VC, the RxCallback will be called many times.
6. IxAtmdAcc returns the number of PDUs processed.

#### 4.4.3.3.3 Receive Disconnect

Before a client disconnects from a VC all resources must have been recycled as shown in Figure 23.

**Figure 23. Rx Disconnect**




1,2. The control client wants to disconnect the VC. IxAtmdAccRxVcDisconnect() tell IxAtmdAcc to discard any rx traffic and, if resources are still in use, the IxAtmdAccRxVcDisconnect() function returns IX_ATMDACC_RESOURCES_STILL_ALLOCATED.

3. Reception of remaining traffic is discarded.

4. The client waits a certain delay, depending on the RX drain rate for this VC, and asks again to disconnect the VC. If resources are still in use, the IxAtmdAccRxVcDisconnect() function returns IX_ATMDACC_RESOURCES_STILL_ALLOCATED

5. Because there are no resources still in use, the IxAtmdAccRxVcDisconnect() function returns IX_SUCCESS. This means that there are no resources or ix_mbufs pending for reception or in the rxFree queue for this VC.

### 4.4.3.4 Buffer Management

The IxAtmdAcc Interface is based on BSD4.4 ix_mbufs.

#### 4.4.3.4.1 Buffer Allocation

Ix_mbufs used by IxAtmdAcc are allocated and released by the client through the appropriate Operating System functions. During disconnect steps, pending buffers will be released by the IxAtmDAcc component using the callback functions provided by the client on a per-VC basis.

#### 4.4.3.4.2 Buffer Contents

For performance reasons, the data pointed to by an ix_mbuf is not accessed by the IxAtmDAcc component.

The ix_mbuf fields required for transmission are described in Table 21. These fields will not be changed during the Tx process.

### Table 21. Ix_mbuf Fields Required for Transmission

| Field | Description |
|---|---|
| m_next | Required. When ix_mbufs are chained to build a PDU. In the last ix_mbuf of a PDU, this field value has to be 0. |
| m_nextpkt | Not used. |
| m_data | Required. This field should point to the part of PDU data. |
| m_len | Required. This field is the length of data pointed to by mh_data. |
| m_type | Not used. |
| m_flags | Not used. |
| m_reserved | Not used. |
| pkt.rcvif | Not used. |
| pkt.len | Required in the first ix_mbuf of a chained PDU. This is the total length of the PDU. |

The ix_mbuf fields of available ix_mbufs used by the receive service are described in Table 22. They are set by the client who wants to provide available buffers to IxAtmdAcc Rx service.

### Table 22. Ix_mbuf Fields of Available Buffers for Reception

| Field | Description |
|---|---|
| m_next | This field value has to be 0. Buffer chaining is not supported when providing available buffers. |
| m_nextpkt | Not used. |
| m_data | This field is the pointer to PDU data. |
| m_len | This field is the length of data pointed to by mh_data. |
| m_type | Not used. |
| m_flags | Not used. |
| m_reserved | Not used. |
| pkt.rcvif | Not used. |
| pkt.len | Set to 0. |

The ix_mbuf fields in received buffers that are set during traffic reception are described in Table 23.

### Table 23. Ix_mbuf Fields Modified During Reception (Sheet 1 of 2)

| Fields | Description |
|---|---|
| m_next | Modified when ix_mbufs are chained to build a PDUro point to the next ix_mbuf. In the last ix_mbuf of a PDU, this field value has to be 0. |
| m_nextpkt | Not used. |
| m_data | This field is the pointer to PDU data. |
| m_len | Modified. This field is the length of data pointed to by mh_data. |
| m_type | Not used. |
| m_flags | Not used. |

**Table 23. Ix_mbuf Fields Modified During Reception (Sheet 2 of 2)**

| Fields | Description |
|---|---|
| m_reserved | Not used. |
| pkt.rcvif | Not used. |
| pkt.len | Not used. |

#### 4.4.3.4.3 Buffer Size Constraints

Any ix_mbuf size can be transmitted, but a full PDU *must* be a multiple of a cell size (48/52 bytes depending on AAL type). Similarly, the system can receive and chain mbufs that are a multiple of a cell size.

When receiving and transmitting AAL PDUs, the overall packet length is indicated in the first ix_mbuf header. For AAL 5, this length includes the AAL-5 PDU padding and trailer.

Buffers with an incorrect size are rejected by IxAtmDAcc functions.

#### 4.4.3.4.4 Buffer Chaining Constraints

Ix_mbufs can be chained to build PDUs up to 64 Kbytes of data plus overhead. The number of mbufs that can be chained is limited to 256 per PDU.

To submit a PDU for transmission, the client needs to supply a chained ix_mbuf. When receiving a PDU, the client gets a chained ix_mbuf.

Similarly, the interface to replenish the Rx queuing system and supporting the Tx-done feature are based on unchained ix_mbufs.

### 4.4.3.5 Error Handling

#### 4.4.3.5.1 API Usage Errors

The AtmdAcc component detects the following misuse of the API:

- Inappropriate use of connection IDs
- Incorrect parameters
- Mismatches in the order of the function call; for example, use start() after disconnect()
- Use of resources already allocated for an other VC; for example, port/VPI/VCI

Error codes are reported as the return value of a function API.

The AAL client is responsible for using its own reporting mechanism and for taking the appropriate action to correct the problem.

#### 4.4.3.5.2 Real-Time Errors

Errors may occur during real-time traffic. Table 24 shows the different possible errors and the way to resolve them.

**Table 24. Real-Time Errors**

| Cause | Consequences and Side Effects | Corrective Action |
|---|---|---|
| Rx free Queue underflow | • System is not able to store the inbound traffic, which gets dropped.<br>• AAL-5 CRC errors<br>• PDU length invalid<br>• Cells missing<br>• PDUs missing | • Use the replenish function more often<br>• Use more and bigger ix_mbufs |
| TX Done overflow | The hardware is blocked because the Tx-done queue is full. | • Poll the TxDone queue more often.<br>• Change the TxDone threshold. |
| IxAtmdAccPduSubmit() reports IX_ATMD_OVERLOADED | System is unable to transmit a PDU. | • Increase the scheduler-transmit speed.<br>• Slow down the submitted traffic. |
| RX overflow | • Inbound traffic is dropped.<br>• AAL-5 CRC errors<br>• PDU length invalid | Poll the Rx streams more often. |

## 4.4.3.6 Memory Requirements

### 4.4.3.6.1 Code Size

The ixAtmdAcc code size is approximately 51 KBytes of code memory.

### 4.4.3.6.2 Data Memory

Data memory includes the memory required to maintain the structures needed by the algorithms and the memory required to exchange information with other components of the system. The stacks size needed is not included in the following estimations.

### 4.4.3.6.3 AtmdAcc Data Memory

ixAtmdAcc internal data size is approximately 13 Kbytes with an additional data memory requirement of up to 644 Kbytes which is used to exchange information with other components in the system.

### 4.4.3.6.4 Client Data Memory

The size and number of ix_mbufs are under the responsibility of the client. Using 2 Kbytes ix_mbufs, and 32 ix_mbufs per channel and 32 channels together, this can amount to 2 Mbytes of memory. The number of ix_mbufs depends on the time the client retains them for its own processing and the traffic rate. The size of the ix_mbuf depends on the application.

## 4.4.3.7 Performance

The IxAtmdAcc component is used to exchange between the Intel XScale core and UTOPIA bus. Table 25 describes some of the traffic rates supported by the ixAtmdAcc component of IXP400 software.

**Table 25. ixAtmdAcc Supported Traffic Rates**

| Technology | Downstream B/W (Mbps) | Upstream B/W (Mbps) |
|---|---|---|
| ADSL | 8 | 1.5 |
| 2xADSL | 16 | 3 |
| 3xADSL | 24 | 4.5 |
| 4xADSL | 32 | 6 |
| G.SHDSL | 2.3 | 2.3 |
| 2xG.SHDSL | 4.6 | 4.6 |
| 3xG.SHDSL | 6.9 | 6.9 |
| 4xG.SHDSL | 9.2 | 9.2 |
| VDSL | 52 | 6.4 |

# 4.5 IxAtmSch

## 4.5.1 IxAtmSch Overview

This section describes the IxAtmSch software component. IxAtmSch is an “example” IXP400 software component, an ATM scheduler component supporting ATM transmit services on IXP42X product line.

This section describes these details of the IxAtmSch:

- Functionality and services
- Interfaces to use the services
- Conditions and constraints for using the services
- Its dependency on other IXP400 software components
- Its performance and resource usage estimates

IxAtmSch is a simplified scheduler with limited capabilities. See Table 26 on page 85 for details of scheduler capabilities.

The IxAtmSch API is specifically designed to be compatible with the IxAtmdAcc transmission control interface. However, if a client decides to replace this scheduler implementation, they are urged to reuse the API presented on this component.

IxAtmSch conforms to interface definitions for the IXP42X product line processors’ ATM transmission control schedulers.

## 4.5.2 IxAtmSch Component Features

The IxAtmSch component is provided as a demonstration ATM scheduler for use in the IXP42X product line processors' ATM transmit. It provides two basic services for managing transmission on ATM ports:

- Outbound (transmission) virtual connection admission control on serving ATM ports.
- Schedule table to the ATM transmit function that will contain information for ATM cell scheduling and shaping.

IxAtmSch implements a fully operational ATM traffic scheduler for use in the IXP42X product line processors' ATM software stack. It is possible (within the complete IXP400 software architecture) to replace this scheduler with one of a different design. In such an event, this component is valuable as a model of the interfaces that another scheduler implementation requires to be compatible with the IXP400 software ATM stack. IxAtmSch complies with the type interfaces for an IXP400 software compatible ATM scheduler as defined by the IxAtmdAcc software component.

The IxAtmSch service model consists of two basic concepts: ATM port and VCC. Instead of dealing with these real hardware and software entities in the IXP42X product line processors' chip and software stack, IxAtmSch models them. Because of this, there is no limit of how many ATM ports it can model and schedule given enough runtime computational resources.

IxAtmSch does not currently model or schedule Virtual Paths (VPs) or support any VC aggregation capability.

In order to use IxAtmSch services, the clients shall first ask IxAtmSch to establish the model for an ATM port. Virtual connections can then be attached to the port. IxAtmSch models the virtual connections and controls the admission of a virtual connection based on the port model and required traffic parameters. IxAtmSch schedules and shapes the outbound traffic for all VCs on the ATM port. IxAtmSch generates a scheduling table detailing a list of VCs and number of cells of each to transmit in a particular order.

The two basic services provided by IxAtmSch are related. If a VC is admitted on the ATM port, IxAtmSch is committed to schedule all outbound cells for that VC, so that they are conforming to the traffic descriptor. The scheduler does not reject cells for transmission as long as the transmitting user(s) (applications) do not over-submit. Conflict may happen on the ATM port because multiple VCs are established to transmit on the port.

If a scheduling commitment cannot be met for a particular VC, it is not be admitted. Note that IxAtmSch admits a VC only based on the port capacity, current-port usage, and required-traffic parameters.

The current IXP42X product line processors' resource requirements are for a maximum of eight ports and a total of 32 VCs across all ports. This may increase in the future.

Table 26 shows the ATM service categories that are supported in the current scheduler model.

**Table 26. Supported Traffic Types**

| Traffic Type | Supported | Num VCs | CDVT | PCR | SCR | MCR | MBS |
|---|---|---|---|---|---|---|---|
| rt-VBR | Yes | Single VC per port | Yes | Yes† | Yes | No | Yes |
| nrt-VBR | Yes | Single VC per port | No | Yes | Yes | No | No |
| UBR | Yes | Up to 32 VC | No | Yes | No | No | No |
| CBR | Yes — simulated | Single VC per port | Yes | Yes | = PCR | No | No |

† This scheduler implementation is special purpose and assumes SCR = PCR.

## 4.5.3 Connection Admission Control (CAC) Function

IxAtmSch makes outbound virtual connection admission decisions based a simple ATM port reference model. Only one parameter is needed to establish the model: outbound (upstream) port rate R in terms of (53 bytes) ATM cells per second. IxAtmSch assumes the real world ATM port is a continuous pipe that draws the ATM cells at the constant cell rate. IxAtmSch does not rely on a hardware clock to get the timing. Its timing information is derived from the port rate. It assumes T = 1/R seconds pass for sending every ATM cell.

IxAtmSch determines if a new (modeled) VC admission request on any ATM port is acceptable using following information supplied by its client:

- Outbound port rate
- Required traffic parameters for the new VC
- Traffic parameters of existing VCs on that port

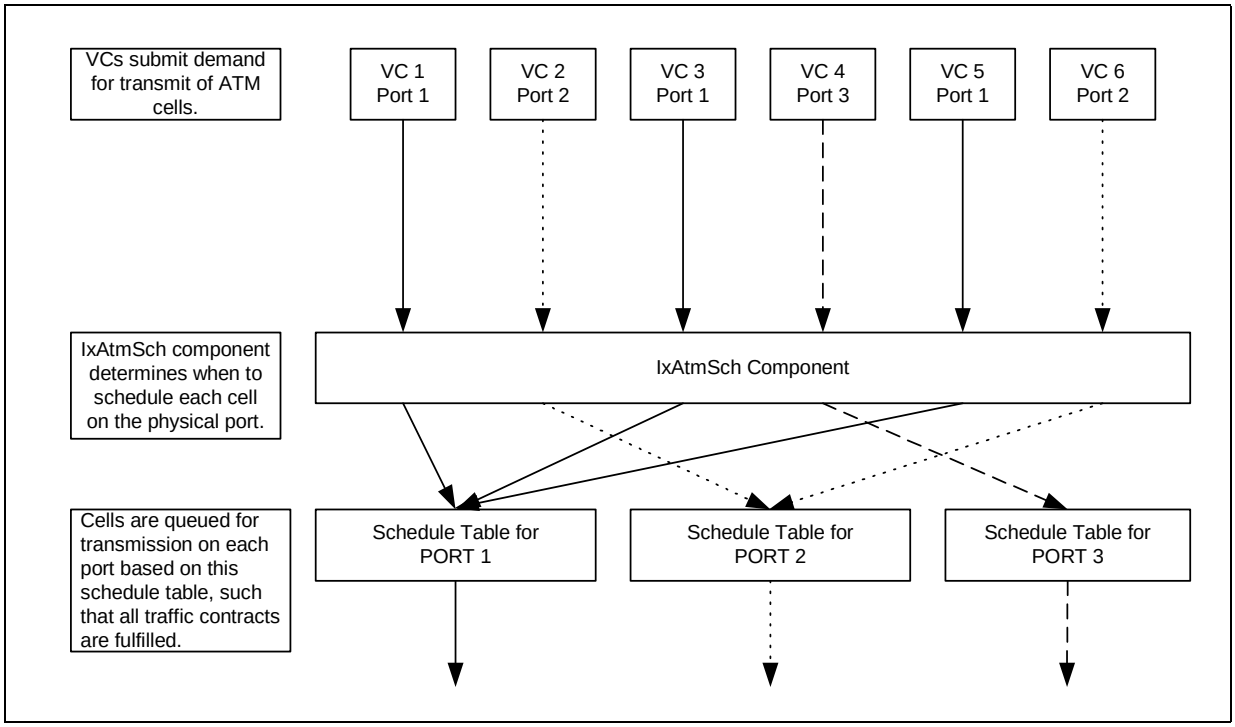
IxAtmSch works on a first-come-first-served basis. For example, if three existing CBR VCs on the ATM port use ¼ port capacity each (PCR = R/4), the fourth CBR VCC asking for 1/3 of the port capacity (PCR = R/3) will be rejected. IxAtmSch issues a globally unique VCC ID for each accepted VCC.

For non-CBR real time VCs — i.e., where the SCR and PCR values are different — only the SCR value is used to determine the required capacity for the VC. This is based on the principle that, over a long term, the required capacity of the VC will be equal to the SCR value, even if the VC may burst at rates above that rate for short periods.

Upon a successful registration via the CAC function, each VC is issued a port-unique identifier value. This value is a positive integer. This value is used to identify the VC to the IxAtmSch during any subsequent calls. The combination of port and VC ID values will uniquely identify any VC in the IXP42X product line processors device to the IxAtmSch component.

## 4.5.4 Scheduling and Traffic Shaping

**Figure 24. Multiple VCs for Each Port, Multiplexed onto Single Line by the ATM Scheduler**




### 4.5.4.1 Schedule Table

Once an ATM port is modeled and VCs are admitted on it, the client can request IxAtmSch to publish the schedule table that indicates how the cells on all modeled VCs over the port shall be interleaved and transmitted.

IxAtmSch publishes a scheduling table each time its scheduling function is called by a client for a particular port. The schedule table data structure returned specifies an ordering on which cells should be transmitted from each VCs on the port for a forthcoming period. The client is expected to requests a table for a port when the transmit queue is low on that port.

The number of cells that are scheduled by each call to the scheduling function will vary depending on the traffic conditions. The schedule table contains an element, totalCellSlots, which specifies how many cell slots are scheduled in this table returned, including idle cells.

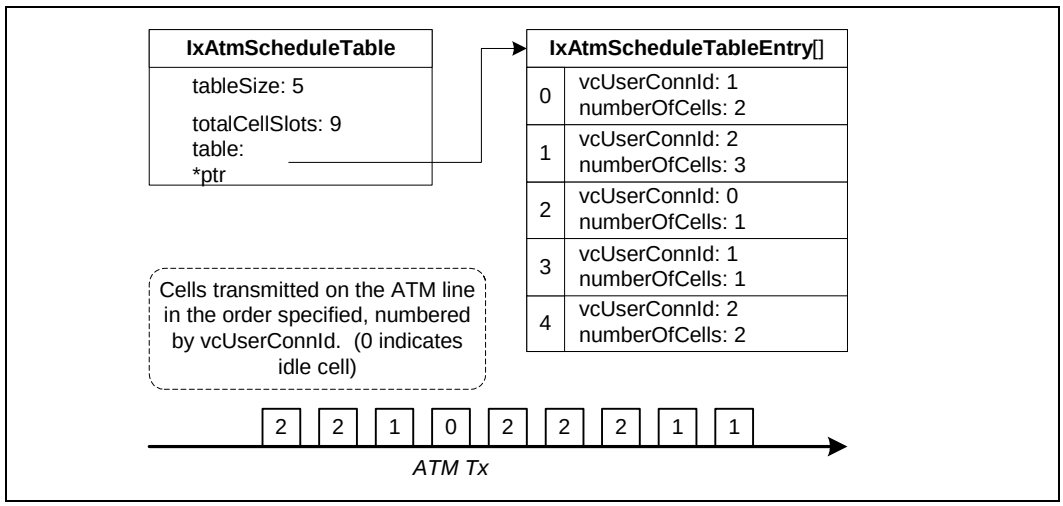
When the client calls the schedule function, the scheduler assumes that all previously scheduled cells on this port have been transmitted and that it may overwrite the previous schedule table with the new table. The client, therefore, must not be dependent on the integrity of the previous table when a request is made for a new schedule table and hence should ensure the current schedule table has been processed by the transmit mechanism before it requests for a new table.

The schedule table is composed of an array of table entries, each of which specifies a VC ID and a number of cells to transmit from that VC. The scheduler explicitly inserts idle cells into the table where necessary in order to fulfill the traffic contract of the VCs registered in the system. Idle cells are inserted in the table with the VC identifier set to 0.

The exact format of the schedule table is defined in `IxAtmTypes.h`.

Figure 25 shows how this table is translated into an ordered sequence of cells transmitted to the ATM port.

**Figure 25. Translation of IxAtmScheduleTable Structure to ATM Tx Cell Ordering**




### Minimum Cells Value (minCellsToSchedule)

When a port model is created the minimum number of cells (minCellstoSchedule) that the scheduler should schedule per table is specified. Therefore, as long as there is at least one cell available to schedule the scheduler will guarantee to generate a table containing a minimum totalCellSlots value of minCellsToSchedule. If the number of outstanding cells available for scheduling is less than minCellsToSchedule then idle cells are scheduled to make up the difference. This value is setup once per port and cannot be modified.

***Note:*** The minCellstoSchedule facility is provided to simplify the transmission control code in the case where queue threshold values are used to drive scheduling. The threshold value in cells can be matched to the minCellsToSchedule so that scheduler is always guaranteed to schedule enough cells to fill the Tx Q above its threshold value.

### Maximum Cells Value (maxCells)

The maximum number of cells that the scheduler produces in a table can be limited by the maxCells parameter. This can controllable on a table by table basis. The actual number of cells scheduled will be the lesser of maxCells and minCellsToSchedule.

## 4.5.4.2 Schedule Service Model

IxAtmSch provides schedule service through two functional interfaces: “VC queue update” and “Schedule table update.”

The client calls the VC queue update interface whenever the user of the VC submits cells for transmission. The structure of the VC queue update interface is compatible with the requirements of the IxAtmdAcc component.

The client calls the Schedule table update interface whenever it needs a new table. Internally, IxAtmSch maintains a transmit queue for each VC.

IxAtmSch also provides a “VC queue clear” interface for use when the client wishes to cancel pending demand on a particular VC. This interface is useful, for example, when the client wishes to remove a VC from the system.

#### 4.5.4.3 Timing and Idle Cells

IxAtmSch does not rely on a hardware clock for timing, instead it derives timing information from the supplied port transmit rate for each modeled ATM port. IxAtmSch assumes that T = 1/R seconds pass for sending every ATM cell. IxAtmSch also assumes that all cells scheduled in a schedule table are transmitted immediately following the cells previously scheduled by the scheduler on that port (i.e., no cells other than those scheduled by IxAtmSch are being transmitted on the port).

The client is responsible for calling “update table” in the following timely fashion if the demand is always there. Suppose the “update table” calls for a port corresponds to time spending T(1), T(2),…, where one T(n) is the time needed to transmit cells scheduled in the n’th updated table. Then, if the demand is always there, the client must call the n’th “update table” before T(1)+T(2)+…+T(n-1) has passed, assuming the client’s first such call is at time 0. This can be easily achieved by making sure that port transmission is never empty when the demand is continuously pouring in.

When all registered VC transmit queues are exhausted, an empty schedule table is returned by the ixAtmSchTableUpdate interface. It is assumed that the client will instruct the lower layers to transmit idle cells until new cells are submitted for transmit on a registered VC. IxAtmSch is not aware of the number of idle cells transmitted in this situation and will reset its internal clock to its starting configuration when new cells are queued.

A further interface is provided to allow the client to update the transmit port rate of an ATM port which has already been registered with the IxAtmSch device, and may have established VCs with pending transmit demand. This interface is provided to cater for the event of line rate drift, as can occur on transmit medium. In the event that the new port rate is insufficient to support all established VC transmit contracts, IxAtmSch will refuse to perform this modification. The client is expected to explicitly remove or modify some established VC in this event such that all established contracts can be maintained, and then resubmit the request to modify the ATM port transmit rate.

***Note:*** If UBR VCs are registered and they specify a PCR that is based on the initial line rate and the line rate subsequently changes to below the PCR values supplied for the UBR connections, the scheduler will still allow the port rate change.

### 4.5.5 Dependencies

The IxAtmSch component has an idealized local view of the system and is not dependent on any other IXP400 software component.

Some function interfaces supplied by the IXP400 software component adhere to structure requirements specified by the IxAtmdAcc component. However, no explicit dependency exists between the IxAtmSch component and the IxAtmdAcc component.

### 4.5.6 Error Handling

IxAtmSch returns an error type to the user when the client is expected to handle the error. Internal errors will be reported using standard IXP42X Product Line processor error-reporting techniques.

### 4.5.7 Memory Requirements

Memory estimates have been sub-divided into two main areas, performance critical and not performance critical.

#### 4.5.7.1 Code Size

The ixAtmSch code size is approximately 35 Kbytes.

#### 4.5.7.2 Data Memory

There are a maximum of 32 VCs per port and eight ports supported by the IxAtmSch component. These multipliers are used in Table 27.

**Table 27. IxAtmSch Data Memory Usage**

| | Per VC Data | Per Port Data | Total |
|---|---|---|---|
| Performance Critical Data | 36 | 44 + (32 * 36) = 1,196 | 9,568 |
| Non Critical Data | 40 | 12 + (40 * 32) = 192 | 10,336 |
| Total | 76 | 2,488 | 19,904 |

### 4.5.8 Performance

The key performance measure for the IxAtmSch component is the rate at which it can generate the schedule table, measured by time per cell. The rate at which queue updates are performed is also important. As this second situation will happen less frequently, however — because a great many cells may be queued in one call to the update function — it is of secondary importance.

The remaining functionality provided by the IxAtmSch is infrequent in nature, being used to initialize or modify the configuration of the component. This situation is not performance-critical as it does not affect the data path of the IXP42X Product Line processor.

#### 4.5.8.1 Latency

The transmit latency introduced by the IxAtmSch component into the overall transmit path of the IXP42X product line processors will be zero under normal operating conditions. This is due to the fact that — when traffic is queued for transmission — scheduling will be performed in advance of the cell slots on the physical line becoming available to transmit the cells that are queued.

## 4.6 ixAtmm

IxAtmm is an example IXP400 software component. The phrase Atmm stands for “ATM Management.” This section describes the following details of ixAtmm:

- Functionality and services
- Interfaces to use these services
- Conditions and constraints for using the services

- Dependency on other IXP400 software components
- Performance and resource usage

## 4.6.1 IxAtmm Overview

The IXP400 software's IxAtmm component is a demonstration ATM configuration and management component intended as a "point of access" for clients to the ATM layer of a IXP42X product line processors device.

This component, supplied only as a demonstration, encapsulates the configuration of ATM components in one unit. It can be modified or replaced by the client as required.

## 4.6.2 IxAtmm Component Features

ixAtmm is an ATM-port, virtual-connection (VC), and VC-access manager. It does not provide support for ATM OAM services and it does not directly move any ATM data.

IxAtmm provides services to:

- Configure and track the usage of the (physical) ATM ports on IXP42X product line processors. In IXP400 software v1.2.2, up to eight parallel logical ports are supported over UTOPIA Level 2. IxAtmm configures the UTOPIA device for a port configuration supplied by the client.
- IxAtmm initializes the IxAtmSch ATM Scheduler component for each active port. IxAtmm assumes that the client will supply initial upstream port rates once the capacity of each port is established.
- IxAtmm ensures traffic shaping is performed for each registered port. IxAtmm acts as transmission control for a port by ensuring cell demand is communicated to the IxAtmSch ATM Scheduler from IxAtmdAcc and cell transmission schedules produced by IxAtmSch are supplied at a sufficient rate to IxAtmdAcc component.
- IxAtmm determines the policy for processing transmission buffers recycled from the hardware. In IXP400 software v1.2.2, the component will ensure this processing is done on an event driven basis. That is, a notification of threshold number of outstanding recycled buffers will trigger processing of the recycled buffers.
- IxAtmm controls processing of receive buffers via IxAtmdAcc. IxAtmdAcc supports two incoming Rx buffer streams termed high- and low-priority streams. The high-priority stream will be serviced in an event driven manner; i.e., as soon a buffer is available in the stream it will be serviced. The low-priority stream will be serviced on a timer basis.
- IxAtmm allows clients register VCC (Virtual Channel Connections) on all serving ATM port for transmitting and/or receiving ATM cells. IxAtmm will check the validity (type of service, traffic descriptor, etc.) of the registration request and will reject any request that presents invalid traffic parameters. IxAtmm does not have the capability to signal, negotiate, and obtain network admission of a connection. The client will make certain that the network has already admitted the requested connection before registering a connection with IxAtmm. IxAtmm also may reject a connection registration that exceeds the port capacity on a first-come-first-serve basis, regardless of whether the connection has already been admitted by the network.
- Allow query for the ATM port and registered VCC information on the port.
- Allow the client to modify the port rate of any registered port after initialization.

### 4.6.3 UTOPIA Level-2 Port Initialization

IxAtmm is responsible for the initial configuration of the IXP42X Product Line processor's UTOPIA Level-2 device. This is performed through a user interface that will facilitate specification of UTOPIA-specific parameters to the IxAtmm component. IxAtmm supports up to eight logical ports over the UTOPIA interface.

The data required for each port to configure the UTOPIA device is the five-bit address of the transmit and receive PHY interfaces on the UTOPIA bus.

The UTOPIA device can also be initialized in loop-back mode. Loop-back is only supported, however, in a single-port configuration.

All other UTOPIA configuration parameters are configured to a static state by the IxAtmm and are not configurable through the functional interface of this component. Clients who wish a greater level of control over the UTOPIA device should modify and recompile the IxAtmm component with the new static configuration, or use the interface provided by the IxAtmdAcc component.

### 4.6.4 ATM-Port Management Service Model

IxAtmm can be considered an "ATM-port management authority." It does not directly perform data movement, although it does control the ordering of cell transmission through the supply of ATM cell-scheduling information to the lower levels.

IxAtmm manages the usage of registered ATM ports and will allow or disallow a VC to be established on these ports, depending on existing active-traffic contracts and the current upstream port rate.

Once a connection is established, a client can begin to use it. The client makes data transfer requests directly to corresponding AAL layer through the IxAtmdAcc component. The AAL layer passes the request to the IXP42X Product Line processor though the appropriate hardware layers under direction from IxAtmm.

The IxAtmm service model consists of two basic concepts:

- ATM port
- VC/VCC (virtual channel/virtual channel connection) connections that are established over this port

A VC is a virtual channel through a port. A VC is *unidirectional* and is associated with a unique VPI/VCI value. Two VCs — in opposite direction on the same port — can share the same VPI/VCI value. A VCC is an end-to-end connection through linked VCs originated from the local ATM port to another device across the ATM network.

Initially a port is "bare" or "empty." A VC must be attached (register) to a port. Registration means, "to let IxAtmm know that, from now on, the VC can be considered usable on this port."

IxAtmm is not responsible for signaling and obtaining admission from network for a VCC. A client needs to use other means to obtain network admission of a VCC, where necessary. A client specifies to IxAtmm the traffic descriptor for the requested VCC. IxAtmm will accept or deny this request based only on the port rate available and the current usage of the port by VCCs already registered with the system. This CAC functionality is provided by the IxAtmSch component.

IxAtmm presumes that the client has already negotiated, or will negotiate, admission of the VCC with the network.

**Figure 26. Services Provided by Ixatmm**

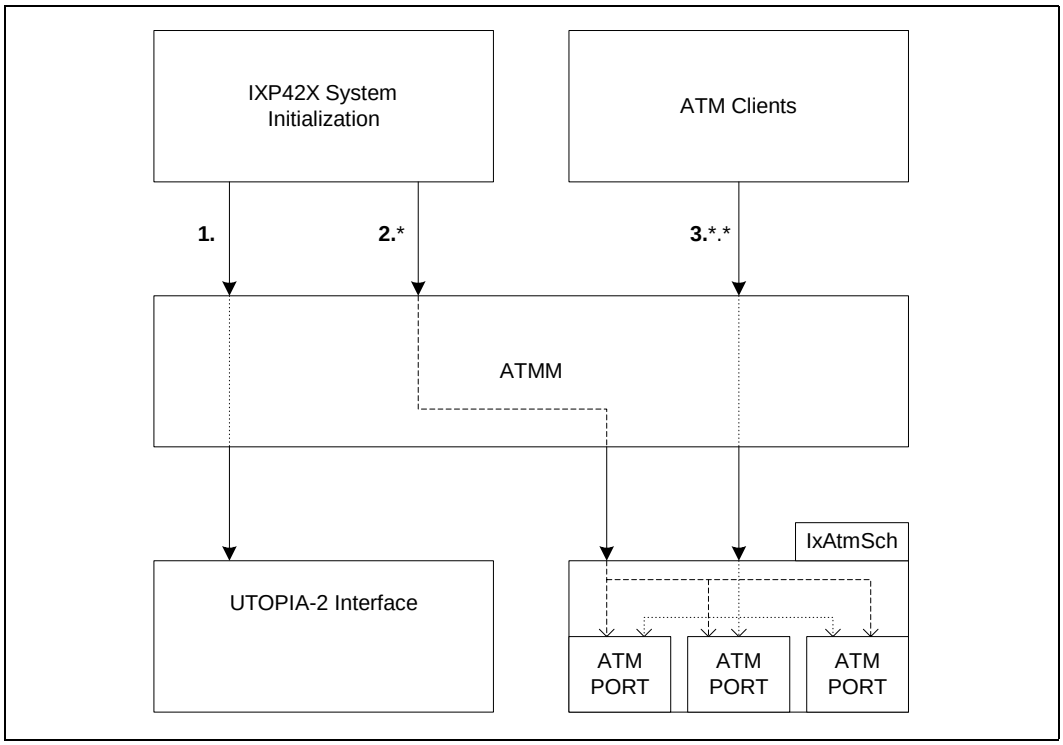



Figure 26 shows the main services provided by the IxAtmm component. In this diagram, the three services outlined are:

- IXP42X product line processors system-initialization routine will invoke an IxAtmm interface function to initialize the UTOPIA Level-2 device for all active ATM ports in the system. This function call is only performed once, encompassing the hardware configuration of all ports in a single call to the interface.
- For each active port, once the link is established and the line rates are known to the system, IxAtmm is informed of the upstream and downstream rate for each port. The upstream rate is required by the ATM scheduler component in order to provide traffic shaping and admission services on the port. The port rates must be registered with IxAtmm before any VCs may be registered. In addition, once the scheduling component is configured, it is bound to IxAtmdAcc. This ensures shaped transmission of cells on the port.
- Once the port rate has been registered, the client may register VCs on the established ports. Upstream and downstream VCs must be registered separately. The client is assumed to have negotiated any required network access for these VCs before calling IxAtmm. IxAtmm may refuse to register upstream VCs — the ATM scheduler refusing admission based on port capacity. Once IxAtmm has allowed a VC, any future transmit and receive request on that VC will not pass through IxAtmm. Instead, they go through corresponding AAL layer directly to the IXP42X Product Line processor's hardware.

Further calls to IxAtmDAcc must be made by the client following registration with IxAtmm to fully enable data traffic on a VC.

IxAtmm does *not* support the registration of Virtual Path Connections (VPCs). Registration and traffic shaping is performed by IxAtmm and IxAtmSch on the VC/VCC level only.

### 4.6.5 Tx/Rx Control Configuration

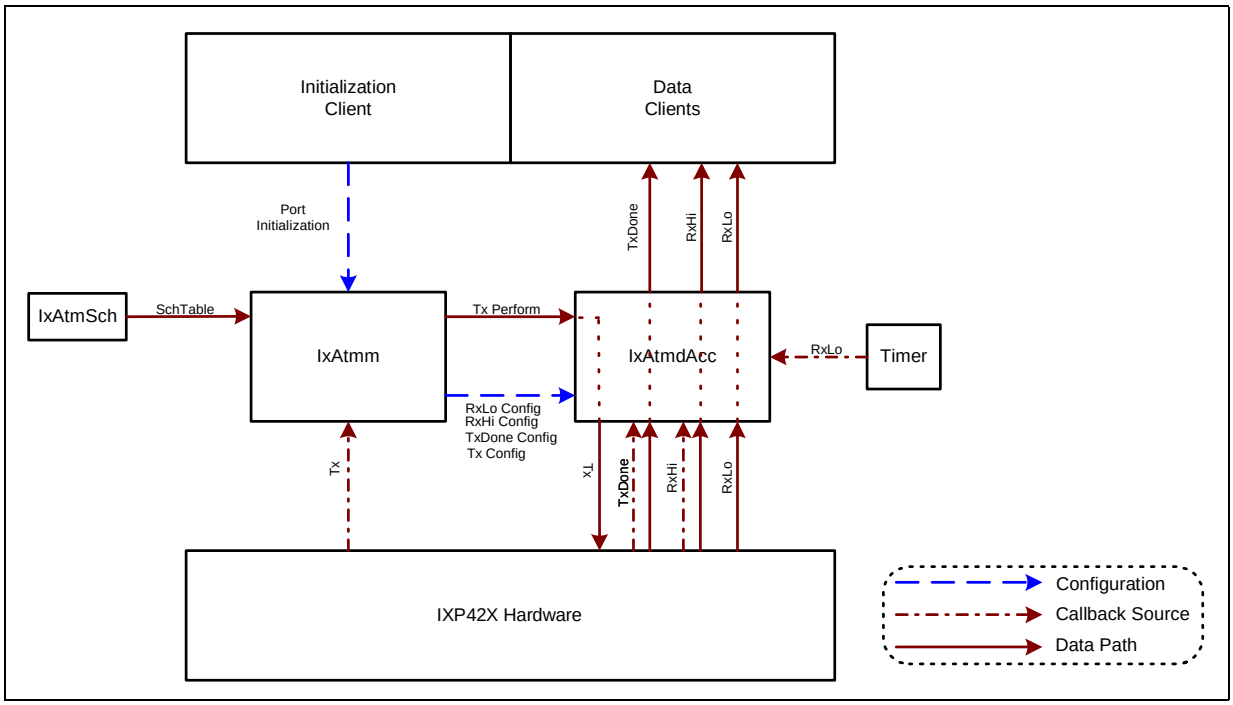
The IxAtmm application is responsible for the configuration of the mechanism by which the lower layer services will drive transmit and receive of traffic to and from the IXP42X Product Line processor hardware. This configuration is achieved through the IxAtmdAcc component interface. Configuration of these services will be performed when the first active port is registered with IxAtmm.

IxAtmm will configure IxAtmdAcc for the following traffic events:

- **Transmit Required** — The IXP42X Product Line processor hardware requires more cells to be scheduled for transmission on a particular port. IxAtmm will implement a callback function that will be registered as a target for the low queue notification callback with IxAtmdAcc. When invoked, this function will generate a transmit schedule table for the port through the IxAtmSch component and pass this table to the IxAtmdAcc interface to cause more cells to be transmitted to the hardware, according to the generated schedule table.
- **Transmit Done** — When all data from a particular buffer has been transmitted, it is necessary for the IXP42X Product Line processor hardware to return the buffer to the relevant client. IxAtmm will configure the IXP42X Product Line processor such that the processing of these buffers will be performed whenever there are a specific number of buffers ready to be processed. IxAtmm will configure the system such that the default IxAtmdAcc interface to return these buffers to the appropriate clients will then be invoked automatically.
- **High Priority Receive** — Data received on the any high priority receive channels is required to be supplied to the client in a timely manner (e.g., voice traffic). IxAtmm will configure the IxAtmdAcc component to process the receipt of data on high priority channels using a low threshold value on the number of received data packets. The default IxAtmdAcc receive processing interface will be invoked whenever the number of data packets received by the IXP42X Product Line processor reaches the supplied threshold. These packets will then be dispatched to the relevant clients by the IxAtmdAcc component.
- **Low Priority Receive** — Data received on low priority receive channels is not as urgent for delivery as the high-priority data and is, therefore, expected to be tolerant of some latency when being processed by the system (e.g., data traffic). IxAtmm will configure the IXP42X Product Line processor such that the receive processing of low-priority data will be handled based on a timer. This will cause the processing of this data to occur at regular time intervals, each time returning all pending low-priority data to the appropriate clients.

It should be noted that IxAtmm is responsible only for the configuration of this mechanism. Where possible the targets of threshold and timer callbacks are the default interfaces for the relevant processing mechanism as supplied by IxAtmdAcc. The exception is the processing of cell transmission, which is driven by an IxAtmm callback interface that passes ATM scheduling information to the IxAtmDAcc component as required to drive the transmit function. As a result, once configured, all data buffers in the system will pass directly through IxAtmdAcc to the appropriate clients. No data traffic will pass through the IxAtmm component at any stage.

### Figure 27. Configuration of Traffic Control Mechanism

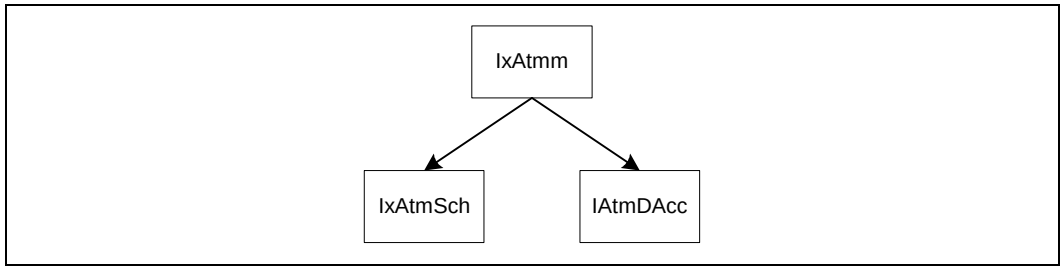



Only transmit traffic which has already been queued by the client with IxAtmdAcc when the request for more traffic is made will be scheduled and sent to the hardware (i.e., no callback to the data client will be made in the context of the transmit processing). IxAtmdAcc makes IxAtmSch aware of the existence of this pending traffic when it is queued by the client through the use of a previously registered callback interface.

The supply of empty buffers to the hardware for use in the receive direction is the responsibility of the individual client on each active VC. As a result, the target callback for this event on each VC is outside of the visibility of the IxAtmm component, being part of the client logic. It is the responsibility of each client, therefore, to ensure that the supply mechanism of free buffers for receive processing is configured correctly before traffic may begin passing on the system.

## 4.6.6 Dependencies

**Figure 28. Component Dependencies of IxAtmm**




IxAtmm configures the IXP42X Product Line processor's UTOPIA Level-2 device through an interface provided by the IxAtmdAcc component.

IxAtmm is also responsible for configuring VC registrations with the IxAtmSch demo ATM scheduler component and relaying CAC decisions to the client in the event of VC registration failure.

IxAtmm is responsible for port traffic shaping by conveying traffic and scheduling information between the ATM scheduler component and the cell transmission control interface provided by the IxAtmdAcc component.

## 4.6.7 Error Handling

IxAtmm returns an error type to the user when the client is expected to handle the error. Internal errors will be reported using standard IXP42X product line processors error-reporting techniques. The established state of the IxAtmm component (i.e., registered ports, VCs, etc.) is not affected by the occurrence of any error.

## 4.6.8 Management Interfaces

No management interfaces are supported by the IxAtmm component. If a management interface is required for the ATM layer, the IxAtmm is the logical place for this interface to be implemented, as the component is intended to provide an abstract public interface to the non-data path ATM functions.

## 4.6.9 Memory Requirements

IxAtmm code is approximately 26 Kbytes in size.

IxAtmm data memory requirement under peak cell-traffic load is approximately 20 Kbytes.

## 4.6.10 Performance

The IxAtmm does not operate on the data path of the IXP42X Product Line processor. Because it is primarily concerned with registration and deregistration of port and VC data, IxAtmm is typically executed during system initialization.

# 4.7 ixHSSAcc

## 4.7.1 Overview

The HssAccess component provides the clients with Intel XScale core driver-level access to the HSS and HDLC coprocessors and supports HSS port configuration.

The HSS-access component is required for an interface to configure both the HSS and HDLC coprocessors. Its primary responsibilities are:

- Configure the HSS coprocessor through the NPE.
- Configure the HDLC coprocessor through the NPE.
- Provide access to the HDLC coprocessor's transmit and receive engines through the NPE packetised service.

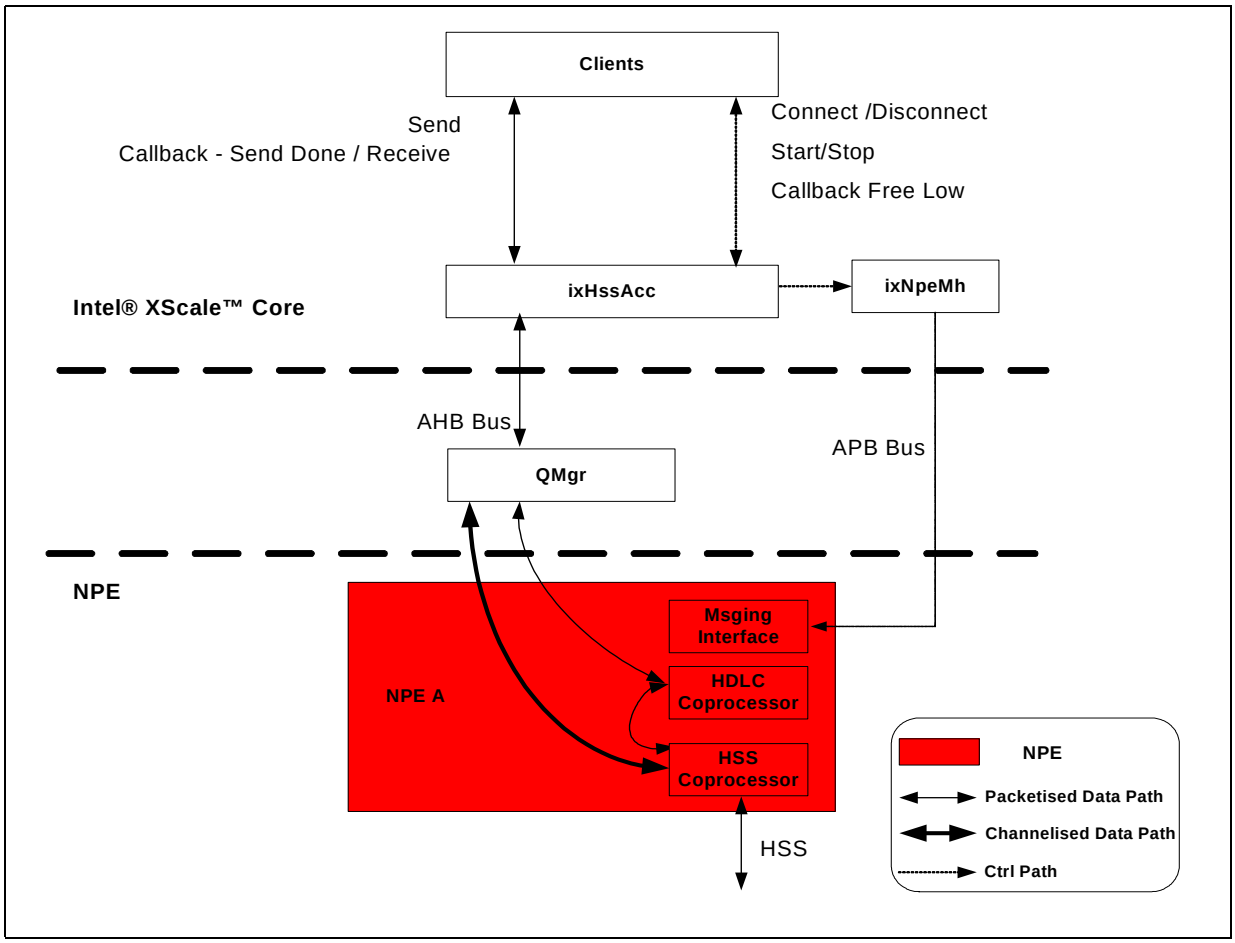
  HssAccess will support up to 4-Kbyte HDLC packets for transmission [1] in the form of chained mbufs. HssAccess will support the receipt of HDLC packets with a recommended maximum size of 16 Kbytes.

  These packets will be passed to the client in the form of chained mbufs.

  ***Note:*** Any packet transmitted by a client through the packetised service will be transmitted out a HSS port.
- Support up to four clients of the NPE packetised service per HSS port — One client per E1/T1 trunk.
- Provide access to the NPE channelised service.
- Support one client of the NPE Channelised Service per HSS port.
- Support up to 32 simultaneous, channelised, TDM slots (64 Kbps or 56 Kbps) per HSS port.

An overview of the data and control flow for the HSS/HDLC is shown in Figure 29.

**Figure 29. HSS/HDLC Access Overview**




## 4.7.2 Features

The HssAccess component provides:

- Access to one of the two HSS ports for the Intel® IXP42X Product Line Processors.
- Support for TDM signals up to a rate of 8 Mbps (4 E1/T1) on a HSS port.
- Channelised access to N x 64 Kbps channels within a TDM stream, where 1<=N<=32.
- HDLC termination points – Up to four, one per E1/T1 trunk.

### 4.7.3 Functional Interface Description

The HssAccess service provides the following interface:

- Primitives to configure an HSS port.
- Primitives to connect/disconnect to or from the packetised service on HDLC port basis.
- Primitives to connect/disconnect to or from the channelised service on a HSS port basis.
- Primitives to enable and disable the receive service for both packetised and channelised.
- Primitives to send and receive buffers.
- Primitives to manipulate the outstanding free buffers Q for a particular HDLC port:
  - Primitive to replenish the Free Q
  - Primitive to register call-backs when the Free Q crosses a particular threshold
  - Primitive to set the Q threshold and call-back source

### 4.7.4 Key Assumptions

The HSS service is predicated on the following assumptions:

- Only one of the two HSS port will be supported.
- Packetised (HDLC) service is coupled with the HSS port.

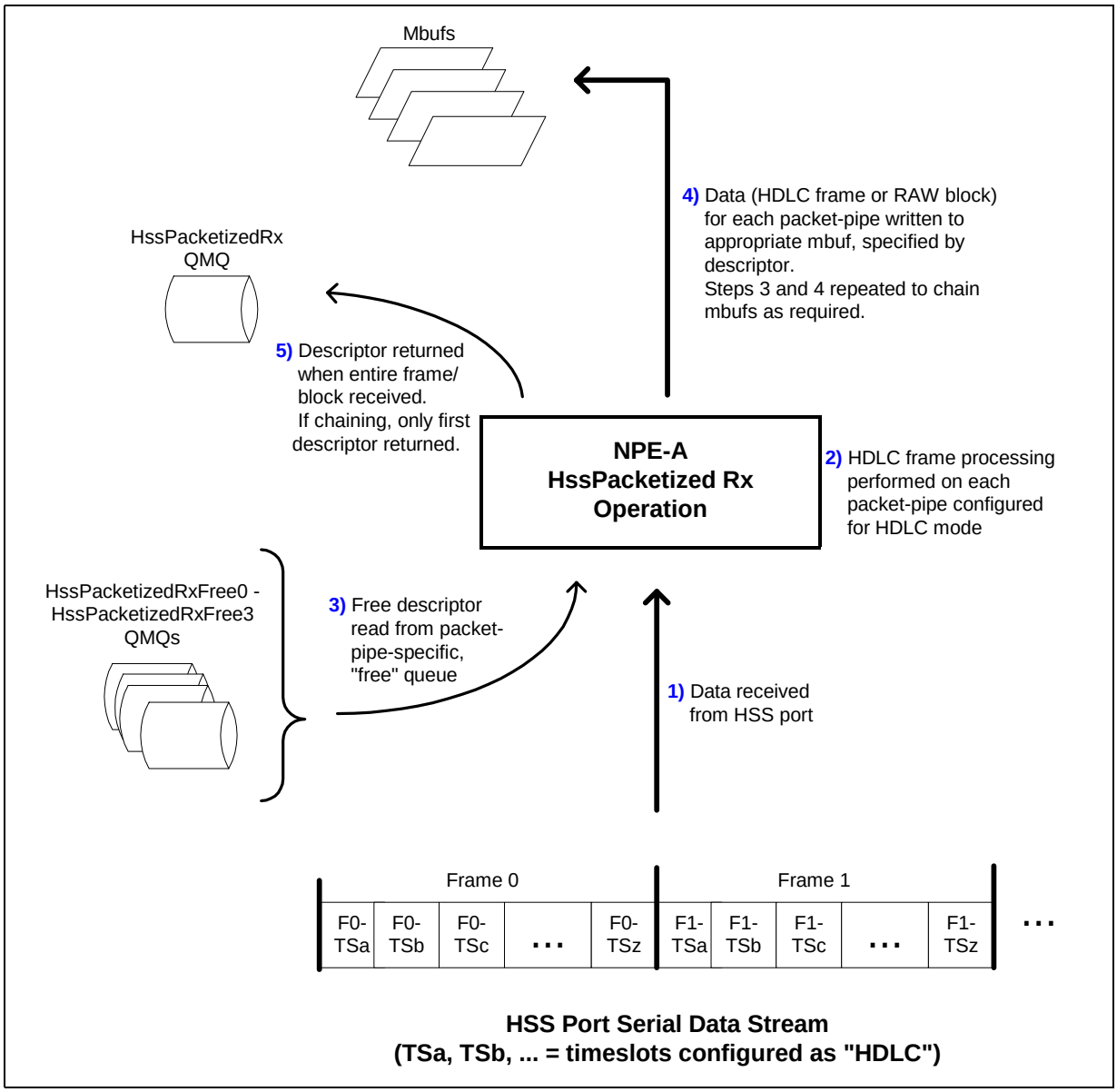
  Packets transmitted using the packetised service access interface will be sent through the HDLC coprocessor and on to the HSS coprocessor.
- Tx and Rx TDM slot assignments are identical.
- Packetised services will use BSD 4.4 mbufs.
- Channelised services will use raw buffers.

All mbufs provided by the client to the packetised receive service will contain 2,048-byte data stores.

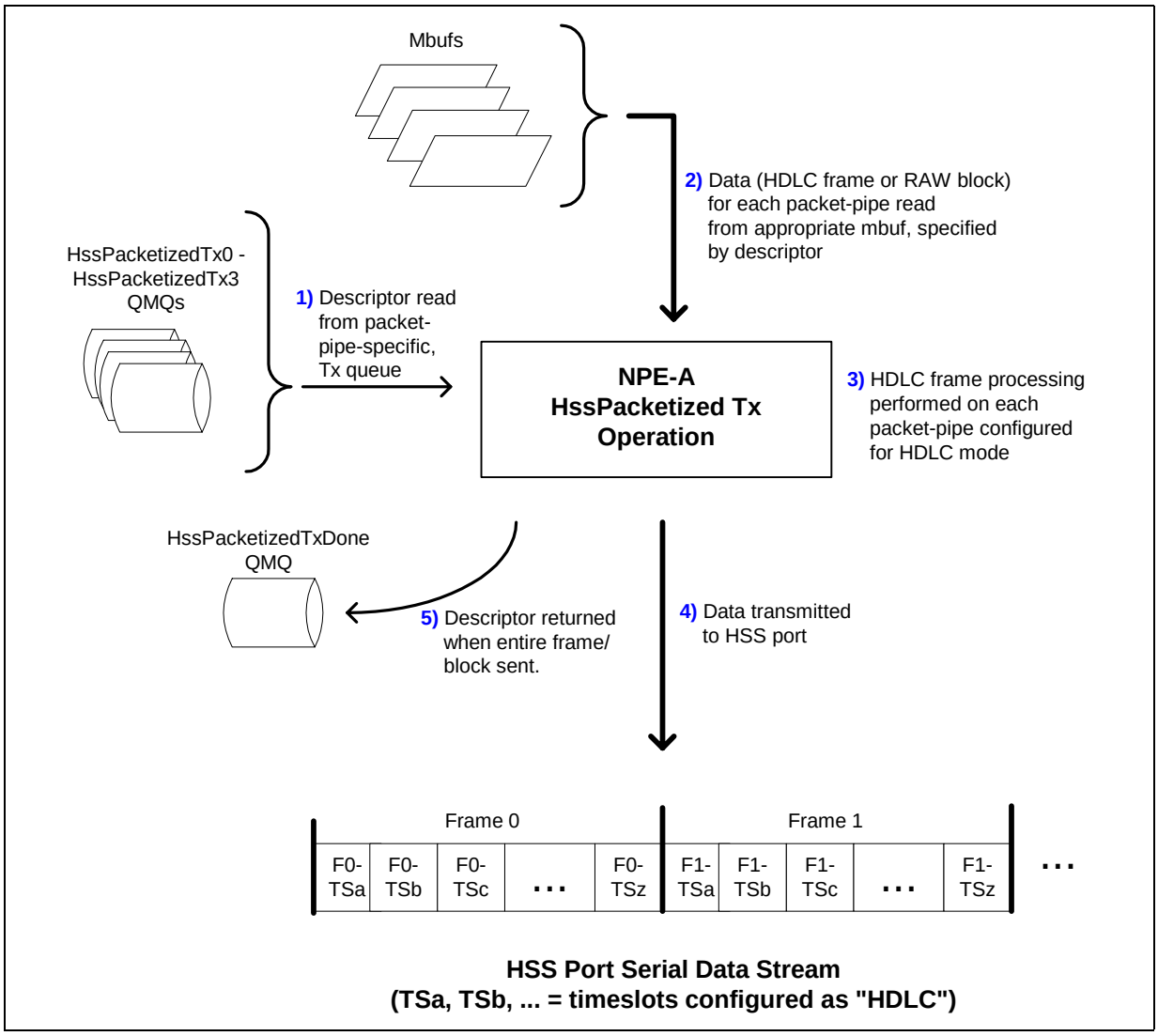
### 4.7.5 Data-Flow Overview

For the packetised access, the client provides a receive call-back function to accept packets received through the HSS. The client will also provide a transmit-done call-back function to accept return of buffers passed down for transmission. It is the client's responsibility to ensure there is always an adequate supply of mbufs for the receive direction. It can achieve this in two ways. A call-back can be registered to HssPktAccess to be called back, when the free mbufs queue is running low. Alternatively, the client can use its own timer for regular supply of mbufs.

### Example 1. HSS Packetised Receive Buffering



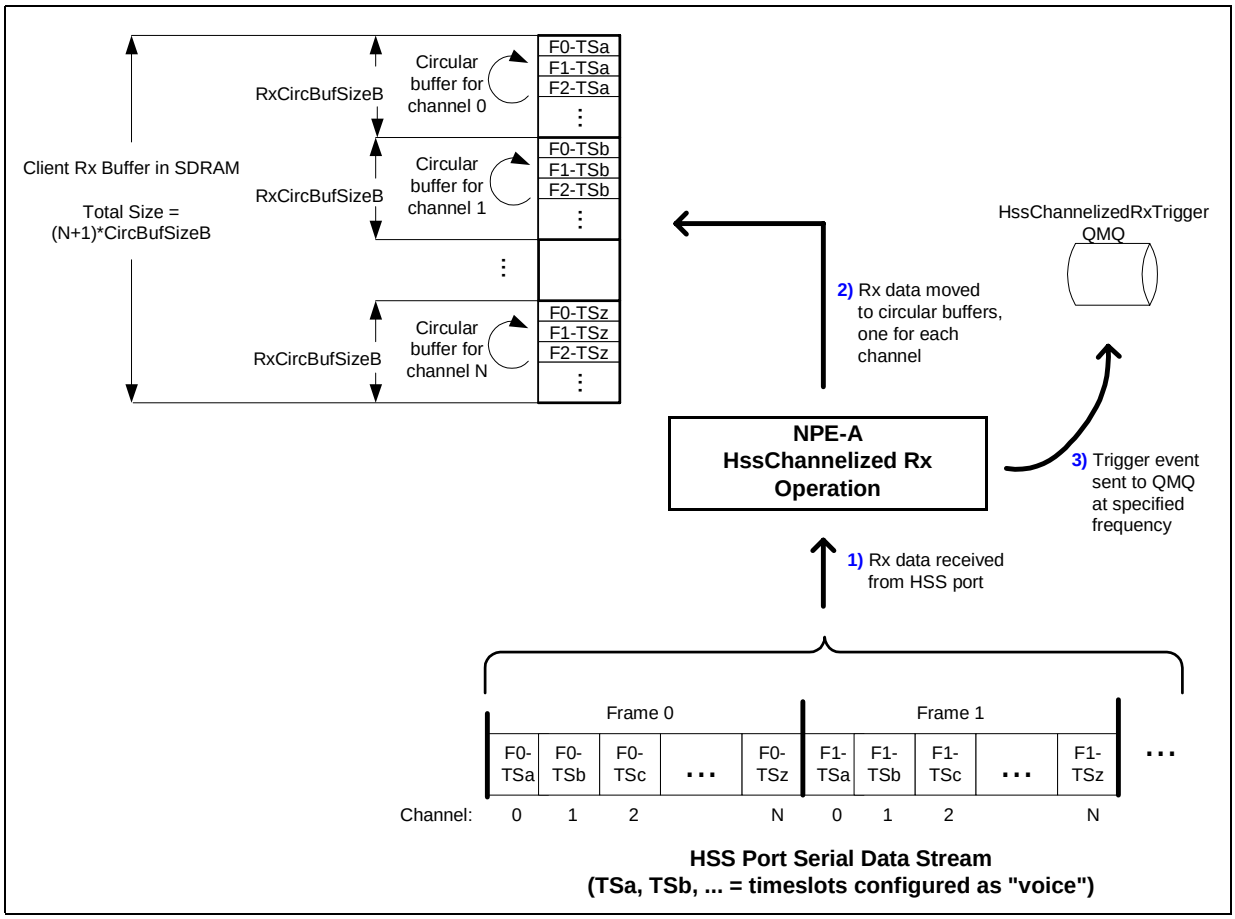
### Example 2. HSS Packetised Transmit Buffering



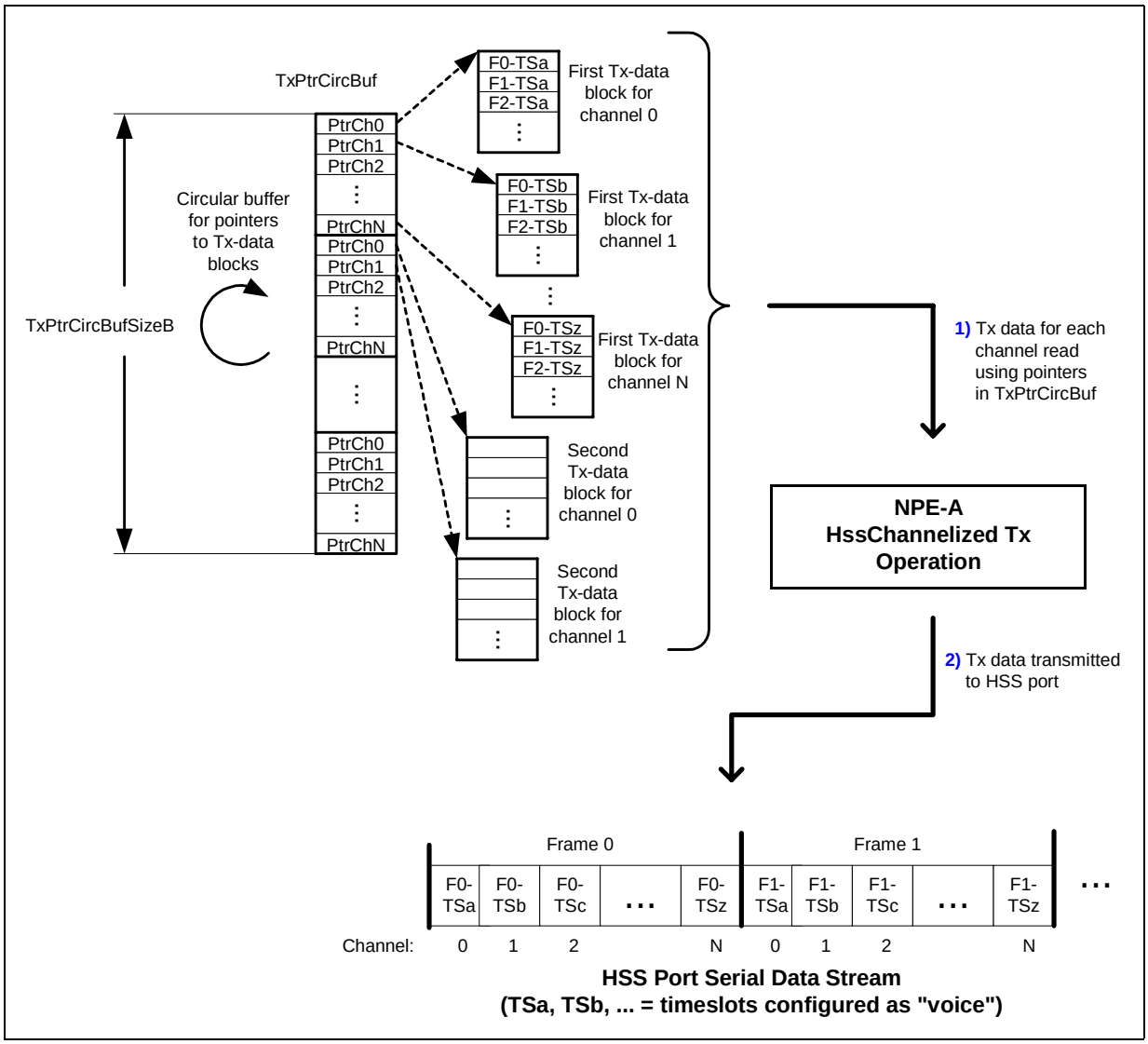

For channelised access, the client provides HssChanAccess with a pointer to a pool of contiguous memory to be filled with data received through the HSS. NPE_A treats this pool as a circular pool of buffers and uses a write rxOffset to index into this memory. When NPE_A writes to the receive memory, it assumes that the address it writes to is no longer in use by the client.

For transmission, the client provides HssChanAccess with a pointer to a pool of contiguous memory, to hold pointer lists for transmission through the HSS. The client expects to fill these transmit pointer lists. NPE_A uses a read txOffset to index into this memory. At regular intervals, the client triggers through a QMgr interrupt. The client is passed the rxOffset and txOffset in the context of the interrupt.

### Example 3. HSS Channelised Receive Operation



### Example 4. HSS Channelised Transmit Operation



## 4.7.6 HSS Port Configuration

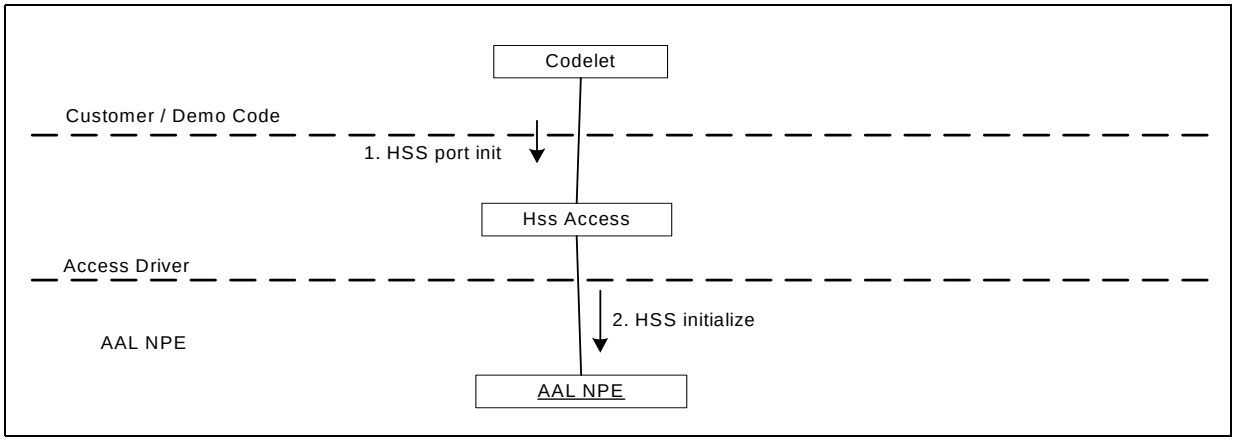
### 4.7.6.1 Description

The client wants to configure a HSS port for packetised/channelised access. The client configures a HSS port to match the configuration of any connected phy.

### 4.7.6.2 Pre-Conditions

The client has no packetised/channelised connections open when trying to configure an HSS port.

**Figure 30. HssAccess**




1. The codelet configures the HSS through HssAccess. This is done once per HSS port. The configuration details include framing characteristics, clock characteristics, data characteristics, number of HDLCs (1, 2, or 4), total number of channelised time slots and a per-time-slot mapping of traffic type – HDLC/Voice/56kVoice/Unassigned.
2. The HssAcces layer configures the NPE appropriately. When complete, the HSS-CoP will be running, although no access is given to the codelet until a connect occurs followed by an enable.

## 4.7.7 HSS Channelised Operation

### 4.7.7.1 Channelised Connect

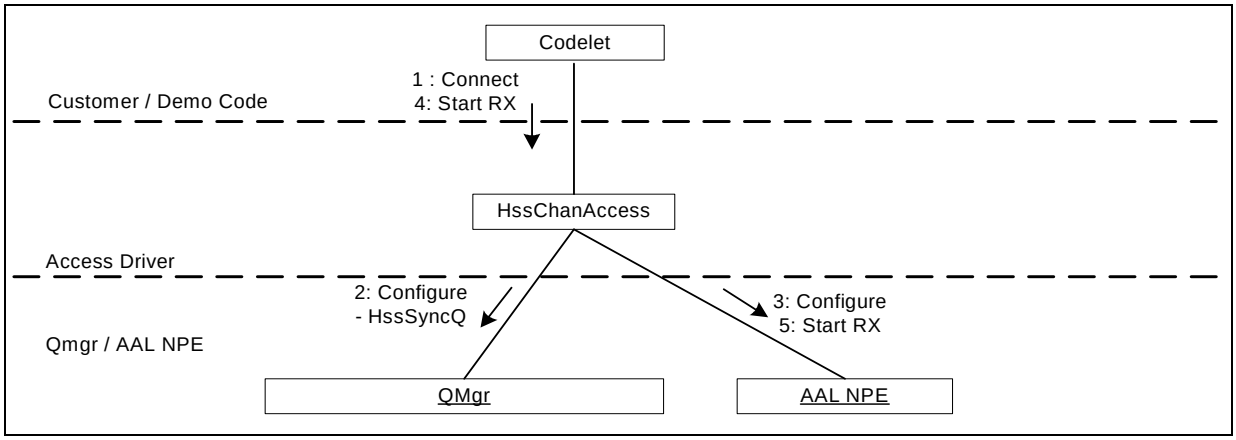
#### Description

The client wants to enable the reception or the transmission of channelised data on a HSS port.

#### Pre-Conditions

- Queue-manager initialization is done.
- HSS is configured appropriately.

**Figure 31. hssChannelisedAccess — Connect**




1. The codelet issues a connect request to HssChanAccess.
2. If blkXferPtr, passed through the connect call, is not NULL, then the codelet expects to be triggered by events to drive the tx and rx block transfers. HssChanAccess registers the function pointer to the QMgr to be called back in the context of an ISR when hssSyncQMQ is not empty. If the blkXferPtr pointer is NULL, the codelet polls HssChanAccess for hssSyncQMQ status.
3. HssChanAccess configures the NPE appropriately.
4. The codelet enables the receive service through the HssChanAccess.
5. HssChanAccess enables the NPE Rx flow.

### 4.7.7.2 Channelised Tx/Rx

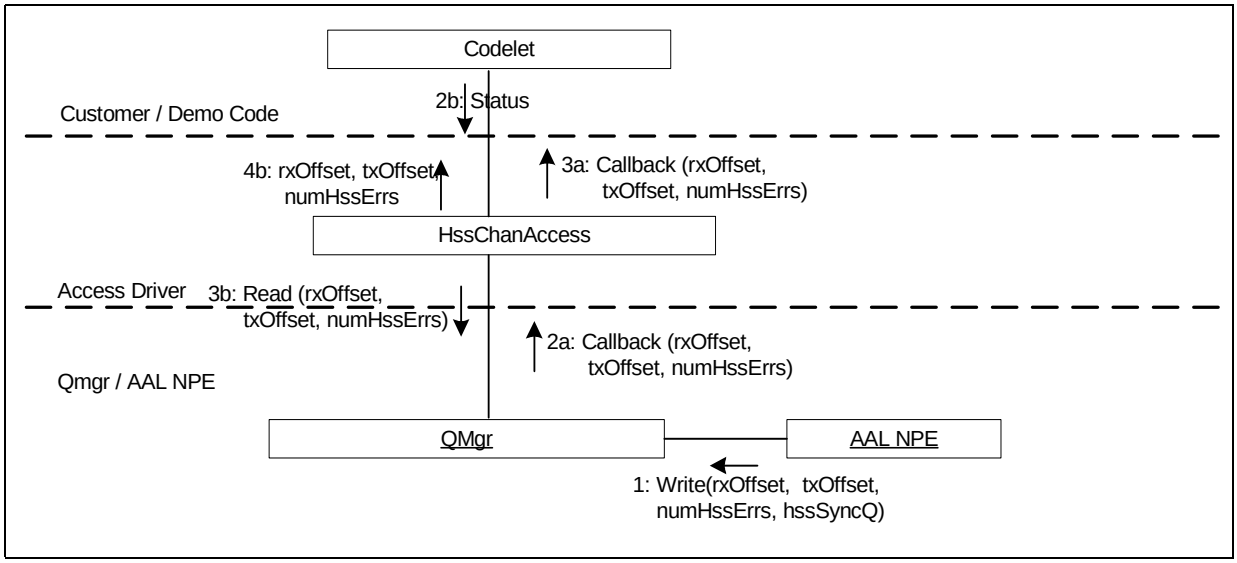
#### Description

The client wants notification of the reception of channelised data on a HSS port. Pointers to buffers are supplied by the client for transmission out the HSS port.

#### Pre-Conditions

- Client should have established a connection and enabled the service for a specific HSS port.
- Pointer to the Rx memory and a pointer to the Tx pointer lists have already been passed down to the NPE through the connect.
- Client can interpret from the data passed to it from HssChanAccess, where in Rx memory the NPE has written to, and where in the list of Tx pointer lists the NPE is currently transmitting.
- Client is responsible for populating the Tx pointer lists for transmission.

**Figure 32. hssChannelisedAccess — Tx/Rx**




1. After reading a configurable amount of data from the HSS and writing the same amount of data to the HSS, the NPE writes to the hssSyncQ.

   There are two possible paths after that depending on how the codelet is connected.

2a. Through an interrupt, the QMgr will call-back HssChanAccess with details of the hssSyncQ entry.

3b. HssChanAccess will call-back the codelet.

2b. The codelet will poll HssChanAccess.

3b. HssChanAccess will in turn poll the QMgr hssSyncQ for status.

4b. If HssChanAccess read an entry from the hssSyncQ, it returns the details to the codelet.

### 4.7.7.3 Channelised Disconnect

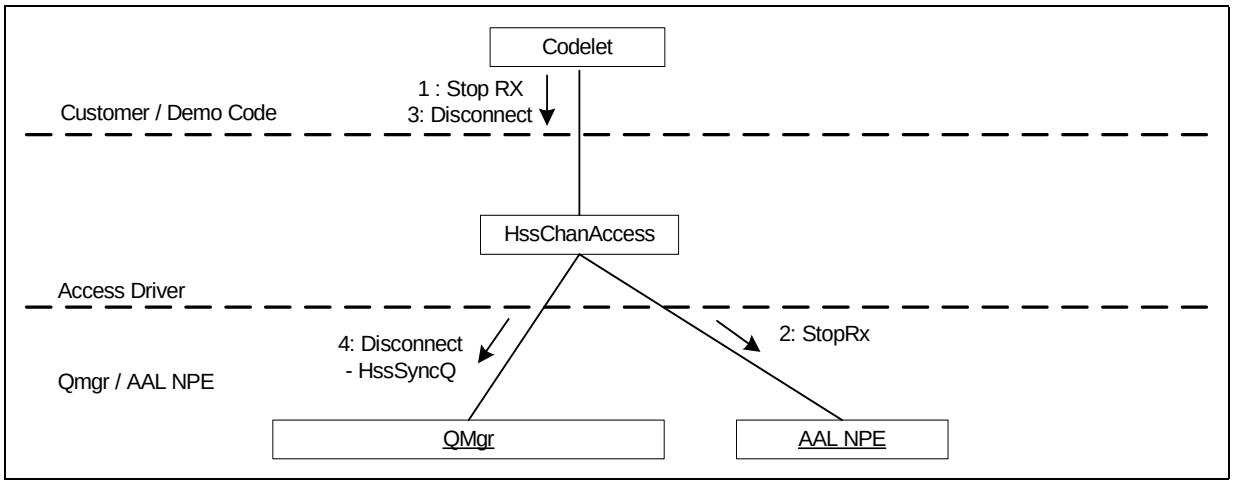
#### Description

The client wants to disable the reception or the transmission of channelised data on an HSS port.

#### Pre-Conditions

- A connection ID should have been allocated during a successful connect step.
- The client will not call any other channelised interfaces during this disconnect.

**Figure 33. hssChannelisedAccess — Disconnect**




1. The codelet requests a disconnect of receive service.
2. The NPE is notified to disconnect processing any data received for this access.
3. The codelet attempts to disconnect the channelised access.
4. If a function pointer was registered to the QMgr for the trigger to transfer tx/rx blocks, it is de-registered.

## 4.7.8 HSS Packetised Operation

### 4.7.8.1 Packetised Connect

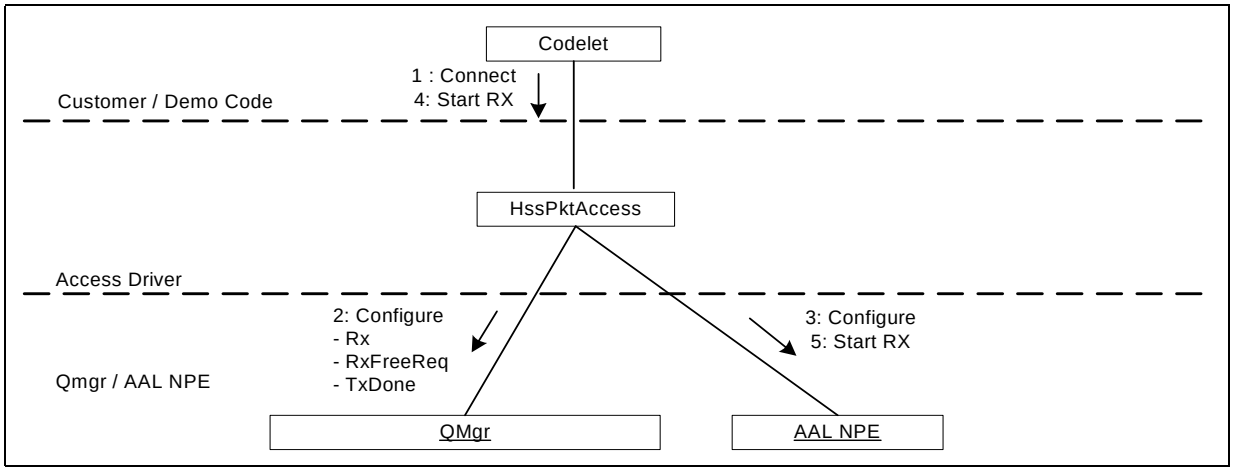
#### Description

The client wants to enable the reception or the transmission of packetised data on a particular number of HDLC ports on a specified HSS port. The client can specify whether the data will be HDLC data, or whether they would like to receive raw data (i.e. no HDLC processing performed).

#### Pre-Conditions

- The queue-manager initialization is done.
- The HSS is configured appropriately.
- The client will have sufficiently filled the Rx FREE Q before starting the Rx service.
- The client will continue to sufficiently replenish the Rx FREE Q while the service is running.

**Figure 34. hssPacketisedAccess — Connect**




1. The codelet issues a connect request to HssPktAccess
2. HssPktAccess configures the QMgr queues and register call-backs to called in the context of its ISR.
3. HssPktAccess configures the NPE with the HDLC parameters passed by the codelet
4. The codelet enables the receive service.
5. HssPktAccess enables the NPE Rx flow.

### 4.7.8.2 Packetised Tx

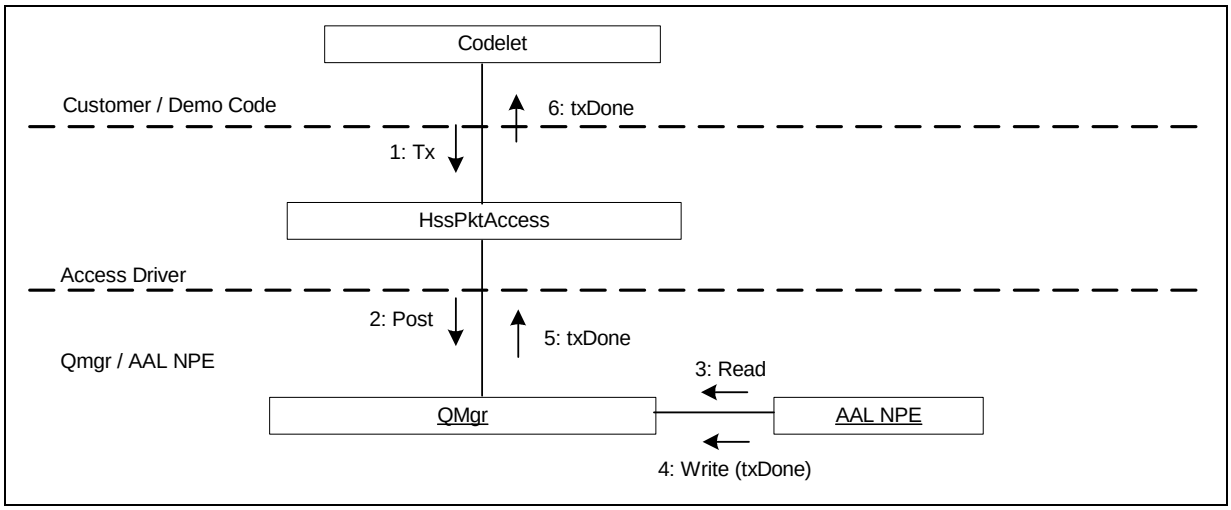
#### Description

The client wants to transmit a HDLC packet (frame) through a HSS port.

#### Pre-Conditions

The client has successfully connected and enabled a packetised service.

**Figure 35. hssPacketisedAccess — Tx**




1. The codelet present san mbuf to HssPktAccess for transmission.
2. HssPktAccess gets a transmit descriptor from its transmit descriptor pool, fills in the descriptor, and writes the address of the descriptor to the Tx Q.
3. The NPE reads a transmit descriptor from the Tx Q on HSS port transmit demand.
4. On completion of transmission, the NPE writes the descriptor to the Tx DONE Q.
5. HssPktAccess is triggered by this action, and the registered call-back executed. The descriptor is freed internally.
6. HssPktAccess calls back the codelet, passing it back its mbuf pointer.

### 4.7.8.3 Packetised Rx

#### Description

The client wants to receive a HDLC packet through a HSS port.

Reception is done in two main steps:

1. The AAL NPE gets free descriptors and mbuf structures from Rx free queues and fills them with the contents of received data.

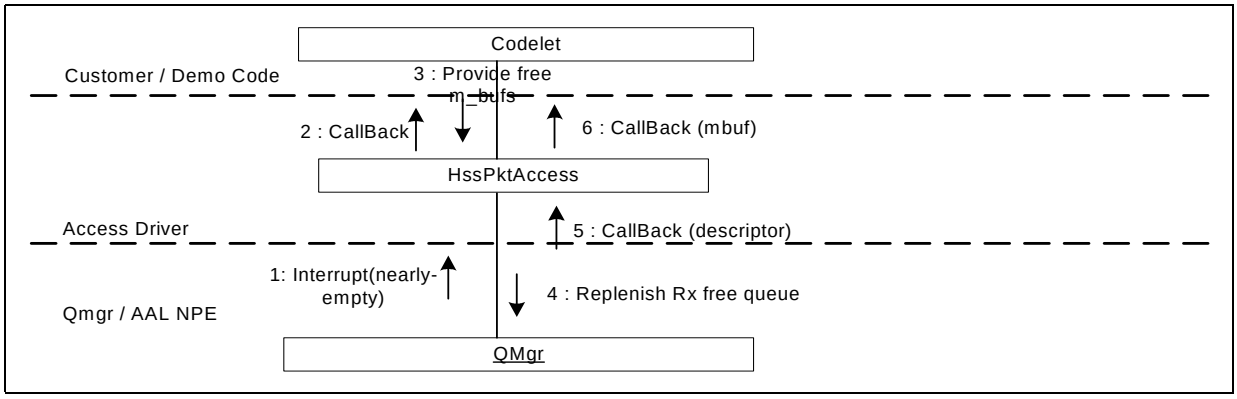
   The client is responsible for replenishing these Rx Free queues.

2. The hssPktAccess component reads the Rx queue (having been triggered by a queue threshold of one), recycles the descriptors entries, and passes the mbufs to the client using either the Rx call-back functions or a polling mechanism.

#### Pre-Conditions

- The client has successfully connected and enabled a packetised service.
- The Rx FREE Qs are sufficiently full.

**Figure 36. hssPacketisedAccess — HDLC Rx (Call-Back Implementation)**




1. The codelet can register a function call-back on connection, to be triggered through an ISR when the Rx FREE Q is low.
2. The codelet is responsible for providing free mbufs for specific HDLC ports.
3. These get stored in the Rx FREE Q within HssPktAccess Rx descriptors.
4. When a complete HDLC packet is received, the Rx Q call-back function is invoked in an interrupt.

   The descriptor is pulled from the Rx Q and the call-back for this channel is invoked with the descriptor. The descriptor gets recycled.

5. The mbuf is transmitted to the codelet.

### 4.7.8.4 Packetised Disconnect

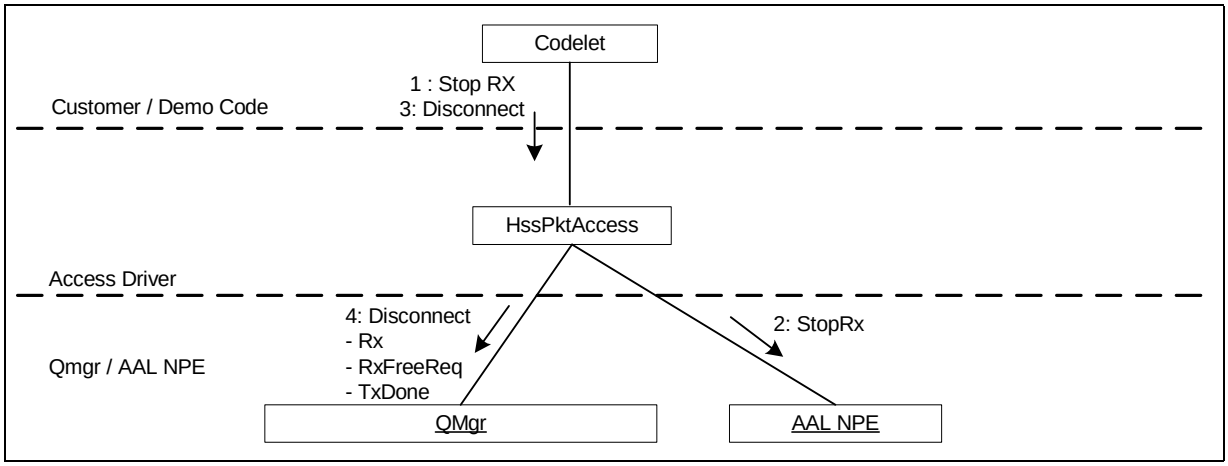
#### Description

The client wants to disable the reception or the transmission of packetised data on particular HDLC ports on a specified HSS port.

#### Pre-Conditions

- The client has successfully connected and enabled a packetised service.
- The client is responsible for ensuring all transmit activity ceases prior to disconnecting.
- The client is responsible for ensuring that the replenishment of the rxFreeQMQ ceases before trying to disconnect.
- No other connect or disconnect is called during this disconnect.

**Figure 37. hssPacketisedAccess — Disconnect**




1. The codelet disables the receive service.
2. HssPktAccess disables NPE service.
3. The codelet asks to disconnect from a HDLC port.
4. HssPktAccess disconnects from its queues associated at time of connection.

## 4.8 ixNpeMh

The IxNpeMh component is responsible for sending messages from software components on the Intel XScale core to the three NPEs (NPE-A, NPE-B, and NPE-C). The component also receives messages from the NPEs and passes them up to software components on the Intel XScale core. This encapsulates the details of NPE messaging in one place and provides a consistent approach to NPE messaging across all components.

Message handling will be a collaboration of Intel XScale core software (IxNpeMh) and the NPE software.

When sending a message that solicits a response from the NPE, the client must provide a callback to the IxNpeMh component to hand the response back. For unsolicited messages, the client should register appropriate callbacks with the IxNpeMh component to hand the messages back.

The IxNpeMh component relies on the IDs of solicited and unsolicited messages to avoid "over-lapping" and determine if a received message is solicited or unsolicited.

Each NPE has two associated data structures — one for unsolicited message callbacks and another for solicited message callbacks.

Messages are sent to the NPEs as in-FIFOs, while messages are received from the NPEs out-FIFOs. Both the in-FIFO and out-FIFO have a depth of two messages.

When sending a message that solicits a response, the solicited callback is added to the end of the list of solicited callbacks. For solicited messages, the first ID-matching callback in the solicited callback list is removed and called. For unsolicited messages, the corresponding callback is retrieved from the list of unsolicited callbacks.

The solicited callback list contains the list of callbacks corresponding to solicited messages not yet received from the NPE. As a given ID's solicited messages is received in the same order that those soliciting messages are sent, the first ID-matching callback in the list always corresponds to the next solicited message that is received.

***Note:*** ***Application software does not need to use ixNpeMh directly. The underlying Access Layer software components (for example, ixEthAcc, ixHssAcc, etc.) take care of all NPE message handling. The only ixNpeMh function that client software is required to execute is ixNpeMhInitialize.***

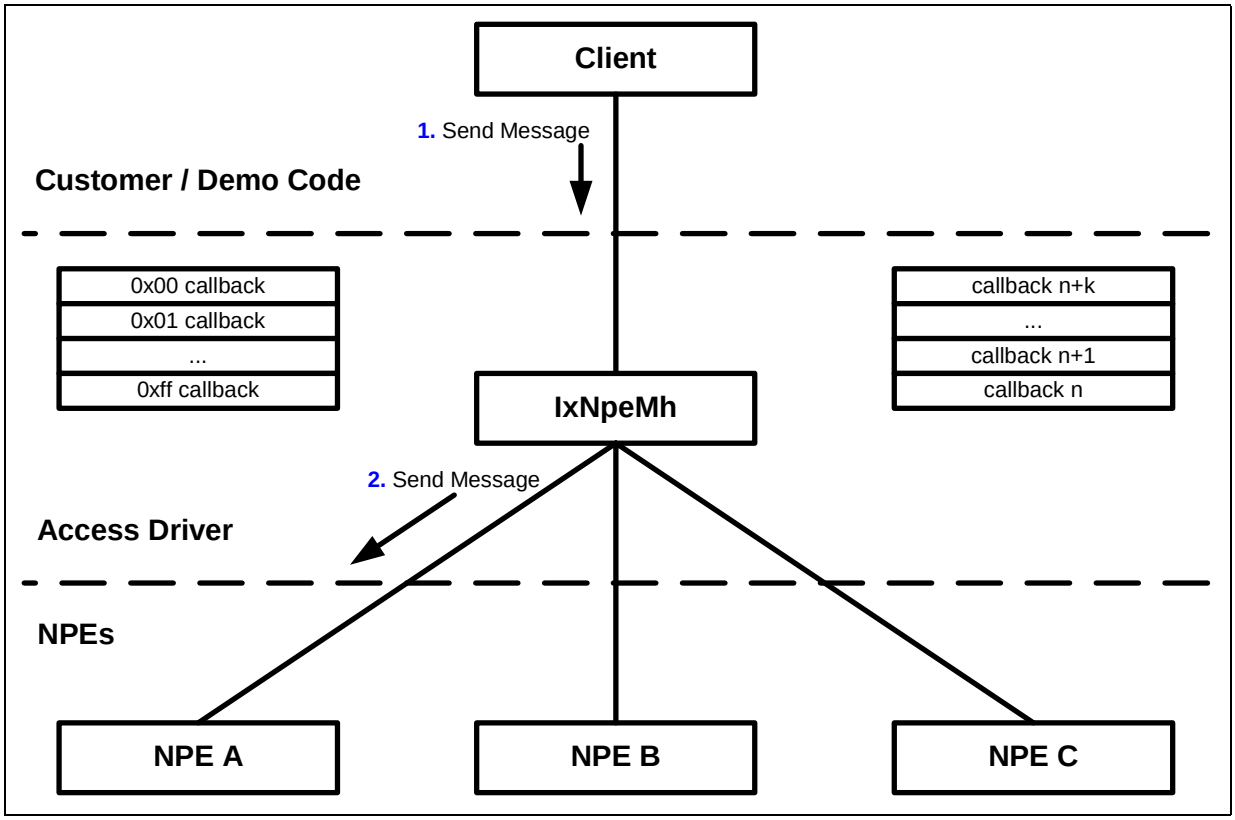
## 4.8.1 Sending Messages from an Intel XScale® Core Software Client to an NPE

### 4.8.1.1 Sending an NPE Message

The scenario of sending a messages from an Intel XScale core software client to an NPE (as shown in Figure 38) is:

- The client sends a message to the IxNpeMh component, specifying the destination NPE.
- The IxNpeMh component checks that the NPE can accept a message
  If not, the send will fail.
- The IxNpeMh component sends the message to the NPE.

**Figure 38. Message from Intel XScale® Core Software Client to an NPE**




### 4.8.1.2 Sending an NPE Message with Response

In this case, the client's message requires a response from the NPE. The scenario (as shown in Figure 39) is:

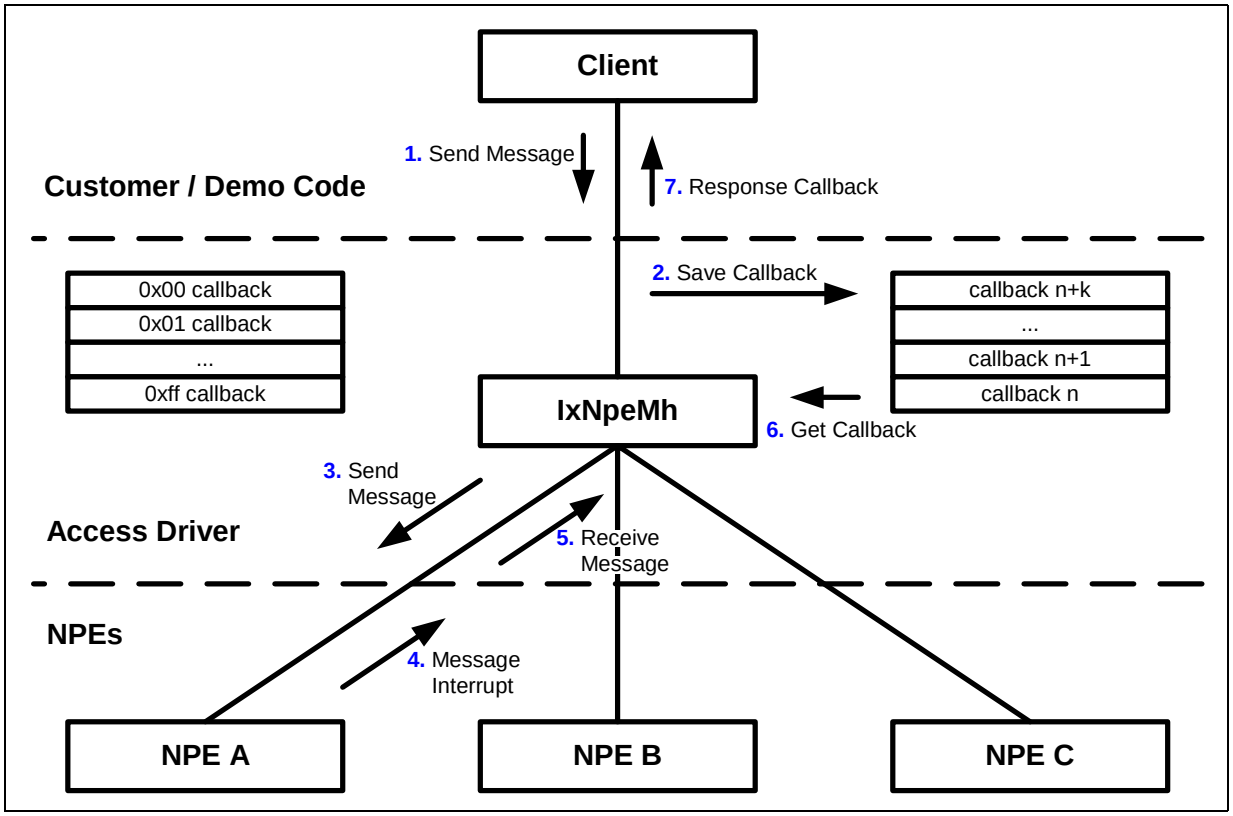
- The client sends a message to the IxNpeMh component, specifying the destination NPE and a response callback.
- The IxNpeMh component checks that the NPE can accept a message.
  If the component cannot accept a message, the send fails.

- The IxNpeMh component adds the response callback to the end of the solicited callback list and sends the message to the NPE.
- After some time, the NPE's "outFIFO not empty" interrupt invokes the IxNpeMh component's ISR.
- Within the ISR, the IxNpeMh component receives a message from the specific NPE.
- The IxNpeMh component checks if this message ID has an unsolicited callback registered for it.

  If the messages has an unsolicited callback registered, the message is unsolicited. (See "Receiving Unsolicited Messages from an NPE to Client Software" on page 114.)
- Because this is a solicited message, the first ID-matching callback is removed from the solicited callback list and invoked to pass the message back to the client.

  If no ID-matching callback is found, the message is discarded and an error reported.

**Figure 39. Message with Response from Intel XScale® Core Software Client to an NPE**



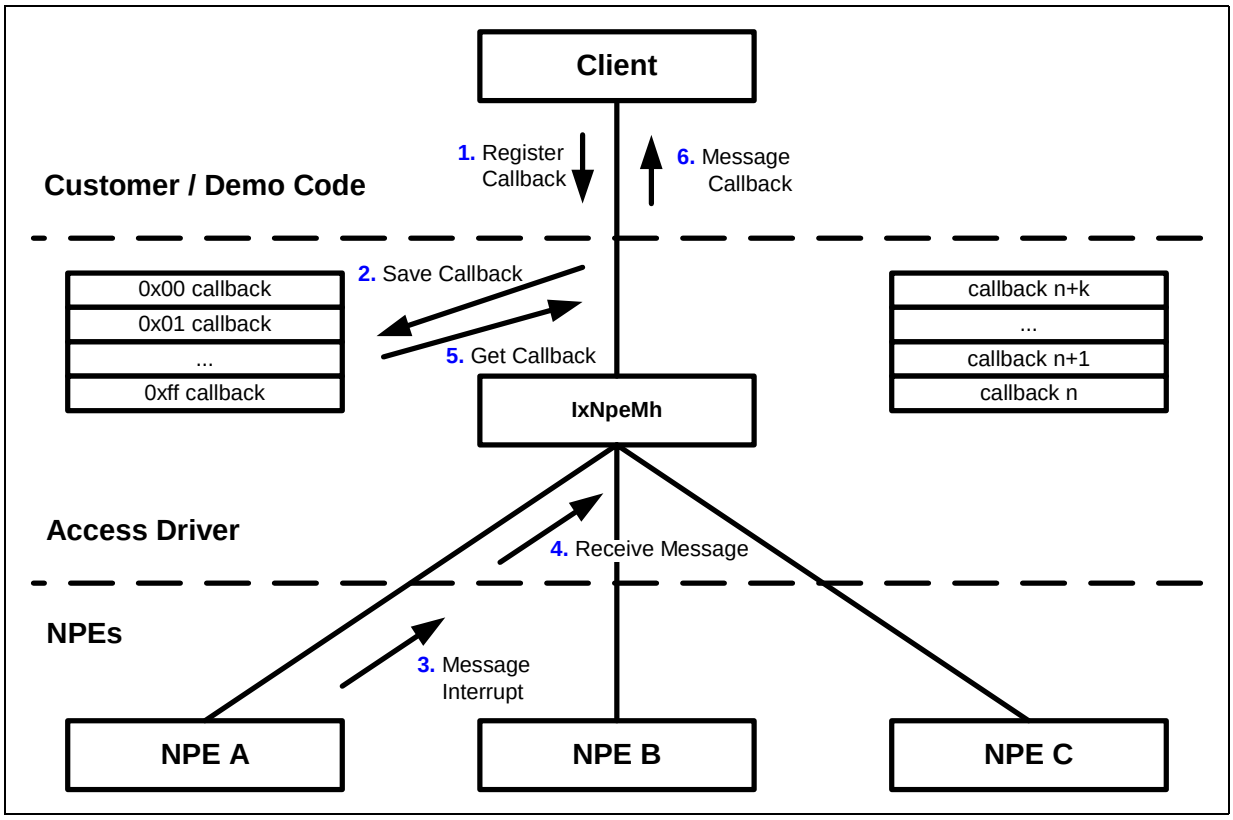
## 4.8.2 Receiving Unsolicited Messages from an NPE to Client Software

The scenario of receiving unsolicited messages from an NPE to client software (as shown in Figure 40) is:

- At initialization, the client registers an unsolicited callback for a particular NPE and a message ID.
- After some time, the NPE's "outFIFO not empty" interrupt invokes the IxNpeMh component's ISR.
- Within the ISR, the IxNpeMh component receives a message from the specific NPE.
- The IxNpeMh component determines if this message ID has an unsolicited callback registered for it.

  If the message ID does not have a registered unsolicited callback, the message is solicited. (See "Sending an NPE Message with Response" on page 112.)
- Since this is an unsolicited message, the IxNpeMh component invokes the corresponding unsolicited callback to pass the message back to the client.

**Figure 40. Receiving Unsolicited Messages from NPE to Software Client**




The IxNpeMh component does not interpret message IDs. It only uses message IDs for comparative purposes, to be able to pass a received message to the correct callback function. This makes the IxNpeMh component immune to changes in message IDs.

The IxNpeMh component relies on the message ID being stored in the most-significant byte of the first word of the two-word message (IxNpeMhMessage).

***Note:*** It is the responsibility of the client to create messages in the format expected by the NPEs.

Multiple clients may use the IxNpeMh component. Each client should take responsibility for handling its own range of unsolicited message IDs. (See the ixNpeMhUnsolicitedCallbackRegister.)

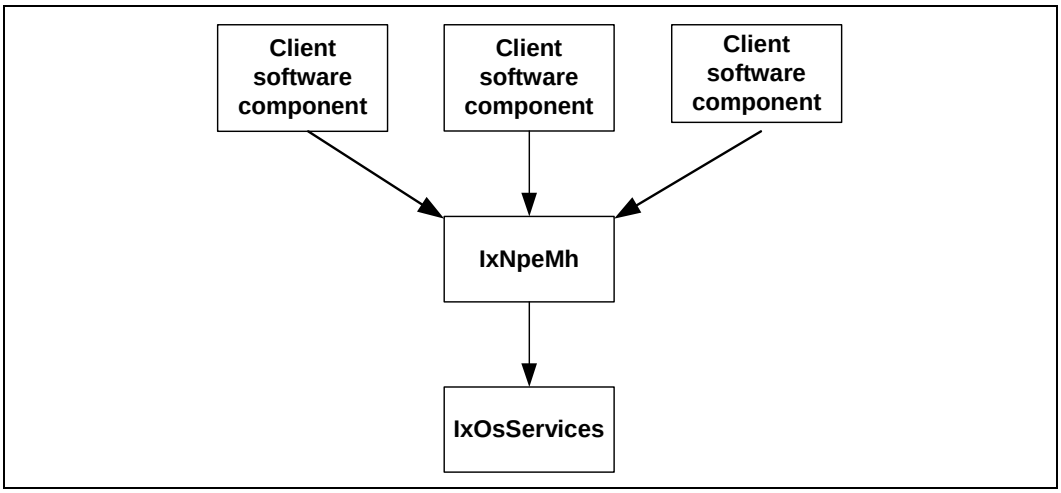
The IxNpeMh component handles messaging for the three NPEs independently. A problem or delay in interacting with one NPE will not impact interaction with the other NPEs.

## 4.8.3 Dependencies

The IxNpeMh component's dependencies (as shown in Figure 41) are:

- Client software components must use the IxNpeMh component for messages to and from the NPEs.
- The IxNpeMh component must use IxOsServices for error-handling, resource protection, and registration of ISRs.

**Figure 41. ixNpeMh Component Dependencies**




## 4.8.4 Error-Handling

The IxNpeMh component uses IxOsServices to report errors and warnings. Parameters passed to the IxNpeMh component are error-checked whenever possible. Interface functions of the IxNpeMh component return a status to the client, indicating success or failure.

The two most important error scenarios when using the IxNpeMh are:

- Failure to send a message if the NPE is unable to accept one.
- Failure to receive a message if no suitable callback can be found.

Failure to send a message implies there is some problem with the NPE. Failure to receive a message means the message will be discarded.

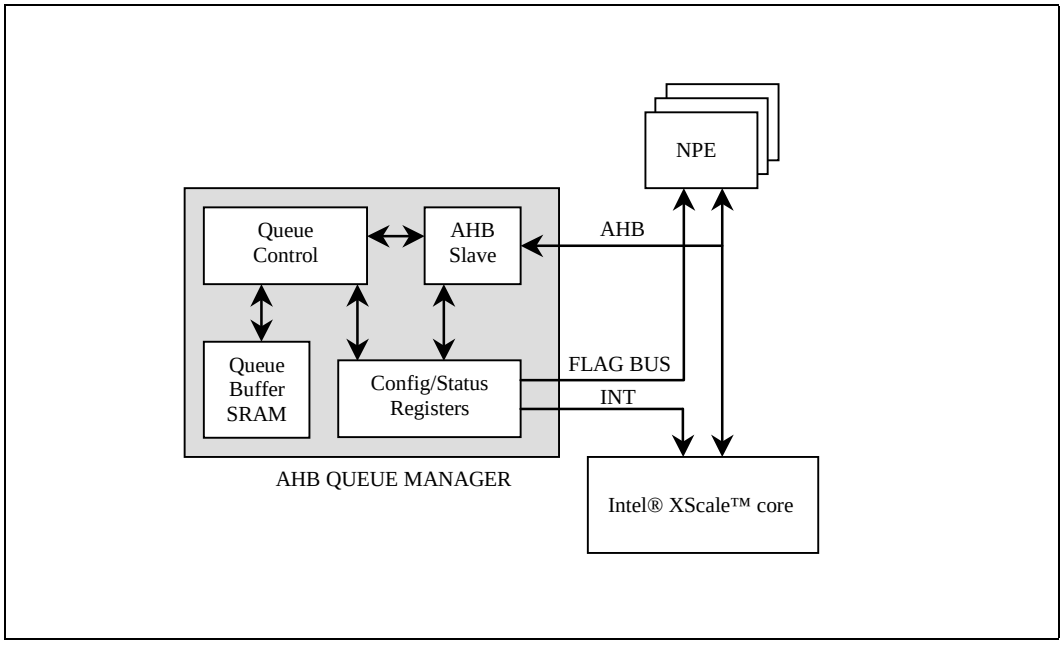
To avoid message loss, clients should ensure that unsolicited callbacks are registered for all unsolicited message types.

## 4.9 ixQmgr

The AHB queue manager hardware block of the Intel® IXP42X product line provides 64 hardware queues to facilitate communication between the Intel XScale core and the NPEs. The IxQMgr (QM) provides an interface so that other Intel XScale core components can use these queues to pass and receive data to and from the NPEs.

The IxQmgr is a collection of software services responsible for managing messages, between the NPEs and client Intel XScale core software, and configuring the AHB queue manager.

**Figure 42. AHB Queue Manager Hardware Block**




The AHB queue manager provides 64 independent queues in which messages or pointers to data, in external memory, are contained. Each queue is configurable for buffer and entry size and is allocated a status register for indicating relative fullness.

It maintains these queues as circular buffers in an internal, 8-Kbyte SRAM. Status flags are implemented for each queue to indicate relative fullness of each queue. The status flags for the lower 32 queues are transmitted to the NPEs via the flag data bus. Two interrupts — one for the lower 32 queues and one for the upper 32 queues — are used as queue status interrupts.

The AHB interface provides for complete queue configuration, queue access, queue status access, interrupt configuration, and SRAM access.

IxQMgr provides the following services:

- Configures AHB queue manager hardware queues.

  Configuration of a queue includes queue size, entry size, watermark levels, and interrupt-source-select flag. IxQMgr checks the validity of the configuration parameters and rejects any configuration request that presents invalid parameters.

- Provides register-notification callbacks for a queue.

  Notification callbacks are registered on a per-queue basis.

- Enables and disables notifications for each AHB-interface queue.
- Sets the priority of a dispatcher callback.
- Provides queue-notification source-flag select.
  - For queues 0-31, the notification source is programmable as the assertion or de-assertion of one of four status flags: Empty, Nearly Empty, Nearly Full, and Full.
  - For queues 32-63, the notification source is the assertion or de-assertion of the Nearly Empty flag and cannot be changed.
- Performs queue-status query.
  - For queues 0-31, the status consists of the flags Nearly Empty, Empty, Nearly Full, and Full, Underflow and Overflow.
  - For queues 32-63, the status consists of the flags Nearly Empty and Full.
- Determines the number of full entries in a queue.
- Determines the size of a queue in entries.
- Reads and writes entries from and to AHB-queue-manager hardware queues.
- Dispatches queue notification callbacks registered by clients.

  These are calls in a defined order, based on a set of conditions.

### 4.9.1 Queue Configuration

The IxQMgr provides queue configuration functionality in two categories:

- Static — Queue configuration that is known at compile time.

  A particular application image is expected to have a static configuration for queues. The queue configuration may vary between images, but is stable for a particular image.

  All static configuration is executed prior to the first queue access. The IxQMgr must be initialized — by calling ixQMgrInit() — before any queue is configured. The queue configuration that is known at compile time includes queue size, queue-entry size, and the queue base address in the AHB queue manager SRAM.

  The static configuration of all queues must be complete before the first access to a queue is made. A compile-time check will be performed on the queue configuration to ensure that the amount of SRAM required by the configuration does not exceed the amount available.

- Dynamic — Queue configuration that may need to change at run-time.

  This configuration includes queue watermarks, interrupt enable/disable, and callback registration.

## 4.9.2 Queue Identifiers

An AHB-queue-manager hardware queue is identified by one of 64 unique identifiers. Each IxQMgr interface function, that operates on a queue, takes one of these identifiers as a parameter. It is the client's responsibility to provide the correct identifier.

The mapping of queue number to queue usage is done in the queue configuration header file.

## 4.9.3 Configuration Values

Table 28 details the attributes of a queue that can be configured and the possible values that these attributes can take on. These values are specified in ref [1](word = 32 bits).

**Table 28. AHB Queue Manager Configuration Attributes**

| Attribute | Description | Values |
|---|---|---|
| Queue Size | The maximum number of words that the queue can contain.<br>Equals the number of entries x queue entry size (in words). | 16, 32, 64, or 128 words |
| Queue Entry Size | The number of words in a queue entry. | 1, 2, or 4 words |
| NE Watermark | The maximum number of occupied entries for which a queue is considered nearly empty. | 0, 1, 2, 4, 8, 16, 32, or 64 entries |
| NF Watermark | The maximum number of empty entries for which a queue is considered to be nearly full. | 0, 1, 2, 4, 8, 16, 32, or 64 entries |

## 4.9.4 Dispatcher

The IxQMgr component provides a dispatcher to enable clients to register notification callbacks to be called, when a queue is in a specified state. A queue's state is defined by the queue status flags E, NE, NF, F, NOTE, NOTNE, NOTNF, and NOTF.

There is no assumption made about how the dispatcher is called. For example, ixQMgrDispatcherLoopRun() may be registered as an ISR for the AQM interrupts, or it may be called from a client polling mechanism, that would read the queues status at regular intervals and call the dispatcher when the queue status changes. In the first example, the dispatcher is called in the context of an interrupt.

The queues that are serviced by the dispatcher each time ixQMgrDispatcherLoopRun() is called, are determined by a range set by the client by calling ixQMgrDispatcherRangeSet(). The order in which queues are serviced depends on the priority specified by calling ixQMgrQDispatchPrioritySet().

**Figure 43. AHB Queue Manager Dispatcher Operation**

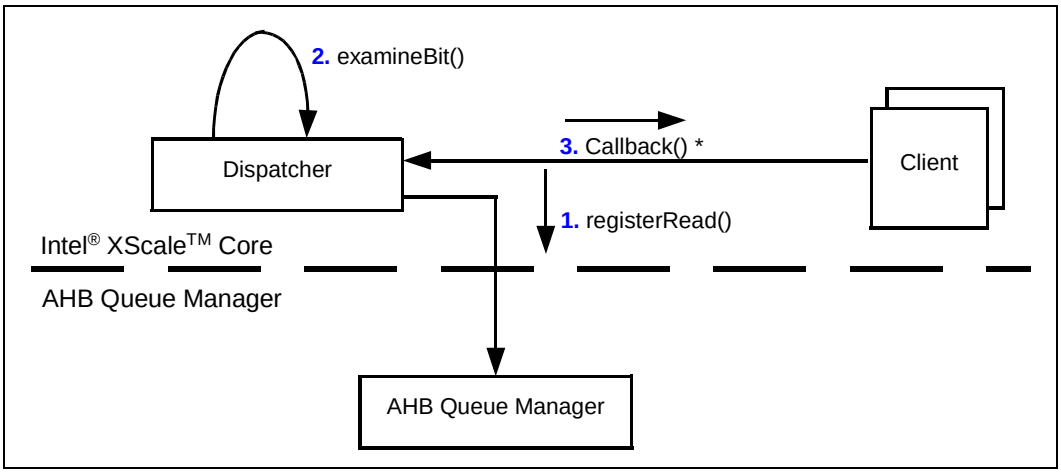



Figure 43 shows the operation of the dispatcher called in the context of an ISR. The interrupt register indicates which queues fired the interrupt. Each bit in this register is examined and, if set, the dispatcher will call each callback registered with this queue.

***Note:*** ***Application software does not need access the queues directly. The underlying Access Layer component software (for example, ixEthAcc, ixHssAcc etc.) take care of this. However, the application software does need to initialize the queue manager software using ixQmgrInit and set up the Dispatcher operation.***

### Figure 44. Example Code for Polled or Interrupt Driven Dispatcher Operation

```
/**
* @fn IX_STATUS ixEthAccCodeletDispatcherStart()
* @param BOOL useInterrupt - start in interrupt or polled mode
* This function starts the Queue manager dispatch timer.
* @return IX_SUCCESS - Dispatch timer successfully started
* @return IX_FAIL - Error starting dispatch timer
*/
PRIVATE IX_STATUS ixEthAccCodeletDispatcherStart(BOOL useInterrupt)
{
    unsigned int dispatchtid;
if(useInterrupt)/* Interrupt mode */
    {
    /* Hook the QM QLOW dispatcher to the interrupt controller. */
    if (ixOsServIntBind(IXP425_INT_LVL_QM1,
                (void *)(ixQMgrDispatcherLoopRun),
                (void *)IX_QMGR_QUELOW_GROUP) != IX_SUCCESS)
    {
        printf("Failed to bind to QM1 interrupt\n");
        return (IX_FAIL);
    }
    }
    else/* Polled mode */
    {
    if (ix_ossl_thread_create((ix_ossl_thread_entry_point_t)
                 ixEthAccCodeletDispatcherPoll,
                 NULL,
                 &dispatchtid) != IX_OSSL_ERROR_SUCCESS)
    {
        printf("Error spawning dispatch task\n");
        return (IX_FAIL);
    }
    ix_ossl_thread_set_priority(dispatchtid, IX_ETHACC_CODELET_QMR_PRIORITY);
    }
return (IX_SUCCESS);
}

/**
 * @fn void ixEthAccCodeletDispatcherPoll
* This function polls the Queue manager loop.
* @return void
*/

PRIVATE void ixEthAccCodeletDispatcherPoll (void* arg, void** ptrRetObj)
{
    while (1)
    {
    ix_ossl_sleep(1);
    ixQMgrDispatcherLoopRun (IX_QMGR_QUELOW_GROUP);
    }
}
```

The example code in Figure 44 is taken from the ixEthAcc codelet.

### 4.9.5 Threading

The IxQMgr does not perform any locking on accesses to the IxQMgr registers and queues. If multiple threads access the IxQMgr, the following IxQMgr functions need to be protected by locking.

- ixQMgrQWrite()
- ixQMgQRead()
- ixQMgrQNotificationtEnable()
- ixQMgrQNotificationDisable()
- ixQMgrQStatusGet()
- ixQMgrQWatermarkSet()
- ixQMgrDispatcherLoopRun()

All IxQMgr functions can be called from any thread, with the exception of ixQMgrInit() which should be called exactly once — before any other call.

### 4.9.6 Dependencies

The IxQMgr component is dependant on the ixOsServices component. IxQMgr uses ixOsServices in ixQMgrDispatcherInterruptConnect() to register AQM ISRs

### 4.9.7 Error Handling

IxQMgr returns an error code, defined in the IxQMgr, to the client. Error codes are returned for the following error cases:

- Invalid parameter value (NULL pointer etc.)
- Overflow occurred on queue write
- Underflow occurred on queue read
- Warning due to change of queue status
- Invalid queue identifier
- Invalid entry size
- Invalid callback identifier

# 4.10 ixUsb

## 4.10.1 Overview

The IXP42X Product Line processor's USB components comply with the 1.1 version of the USB standard.

***Note:*** The VxWorks* USB stack will not be shipped with the default Intel® IXDP425 Network Processor Development Platform BSP available from the Wind River* web site. If this specific USB functionality is required on VxWorks, a USB developer's kit must be purchased directly from Wind River.

## 4.10.2 API Interfaces

**Table 29. API interfaces Available for Access Layer**

| API | Description |
|---|---|
| ixUSBDriverInit | Initialize driver and USB Device Controller. |
| ixUSBDeviceEnable | Enable or disable the device. |
| ixUSBEndpointStall | Enable or disable endpoint stall. |
| ixUSBEndpointClear | Free all Rx/Tx buffers associated with an endpoint. |
| ixUSBSignalResume | Trigger signal resuming on the bus. |
| ixUSBFrameCounterGet | Retrieve the 11-bit frame counter. |
| ixUSBReceiveCallbackRegister | Register a data receive callback. |
| ixUSBSetupCallbackRegister | Register a setup receive callback. |
| ixUSBBufferSubmit | Submit a buffer for transmit. |
| ixUSBBufferCancel | Cancel a buffer previously submitted for transmitting. |
| ixUSBEventCallbackRegister | Register an event callback. |
| ixUSBIsEndpointStalled | Retrieve an endpoint's stall status. |
| ixUSBStatisticsShow | Display device state and statistics. |
| USBErrorStringGet | Convert an error code into a human-readable string error message. |
| ixUSBEndpointInfoShow | Display endpoint information table. |
| free_IX_USB_MBLK | Free an IX_USB_MBLK buffer. |

### 4.10.3 USB Operation

The IXP42X Product Line processor's Universal Serial Bus Device Controller (UDC) supports 16 endpoints and can operate half-duplex at a baud rate of 12 Mbps (slave only, not a host or hub controller).

The serial information transmitted by the UDC contains layers of communication protocols, the most basic of which are fields. UDC fields include:

- Sync
- Endpoint
- Data
- Packet identifier
- Frame number
- CRC
- Address

Fields are used to produce packets. Depending on the function of a packet, a different combination and number of fields are used. Packet types include:

- Token
- Handshake
- Data
- Special

Packets are then assembled into groups to produce frames. These frames or transactions fall into four groups:

- Bulk
- Interrupt
- Control
- Isochronous

Endpoint 0, by default, is used only to communicate control transactions to configure the UDC after it is reset or hooked up (physically connected to an active USB host or hub). Endpoint 0's responsibilities include:

- Connection
- Endpoint configuration
- Disconnect
- Address assignment
- Bus enumeration

The USB protocol uses differential signaling between the two pins for half-duplex data transmission. A 1.5-KΩ pull-up resistor is required to be connected to the USB cable's D+ signal to pull the UDC+ pin high when polarity for data transmission is needed. Using differential signaling allows multiple states to be transmitted on the serial bus. These states are combined to transmit data, as well as various bus conditions, including: idle, resume, start of packet, end of packet, disconnect, connect, and reset.

USB transmissions are scheduled in 1 ms frames. A frame starts with a SOF (Start-Of-Frame) packet and contains USB packets. All USB transmissions are regarded from the host's point of view: IN means towards the host and OUT means towards the device.

## 4.10.4 Packet Formats

USB supports four packet types:

- Token
- Handshake
- Data
- Special

A token packet is placed at the beginning of a frame and is used to identify OUT, IN, SOF, and SETUP transactions. OUT and IN frames are used to transfer data, SOF packets are used to time isochronous transactions, and SETUP packets are used for control transfers to configure endpoints. An IN, OUT and SETUP token packet consists of a sync, a PID, an address, an endpoint, and a CRC5 field.

For OUT and SETUP transactions, the address and endpoint fields are used to select which UDC endpoint is to receive the data, and for an IN transaction, which endpoint must transmit data. A PRE (Preamble) PID precedes a low-speed (1.5 Mbps) USB transmission. The UDC supports full-speed (12 Mbps) USB transfers only. PRE packets signifying low-speed devices are ignored as well as the low-speed data transfer that follows.

**Table 30. IN, OUT, and SETUP Token Packet Format**

| 8 Bits | 8 Bits | 7 Bits | 4 Bits | 5 Bits |
|---|---|---|---|---|
| Sync | PID | Address | Endpoint | CRC5 |

A Start Of Frame (SOF) is a special type of token packet that is issued by the host at a nominal interval of once every 1 ms +/- 0.0005 ms. SOF packets consist of a sync, a PID, a frame number (which is incremented after each frame is transmitted), and a CRC5 field, as shown in Table 31. The presence of SOF packets every 1ms prevents the UDC from going into suspend mode.

**Table 31. SOF Token Packet Format**

| 8 Bits | 8 Bits | 11 Bits | 5 Bits |
|---|---|---|---|
| Sync | PID | Frame Number | CRC5 |

Data packets follow token packets, and are used to transmit data between the host and UDC. There are two types of data packets as specified by the PID: DATA0 and DATA1. These two types are used to provide a mechanism to guarantee data sequence synchronization between the transmitter and receiver across multiple transactions.

During the handshake phase, both communicate and agree which data token type to transmit first. For each subsequent packet transmitted, the data packet type is toggled (DATA0, DATA1, DATA0, and so on). A data packet consists of a sync, a PID, from 0 to 1,023 bytes of data, and a CRC16 field, as shown in Table 32. Note that the UDC supports a maximum of 8 bytes of data for an Interrupt IN data payload, a maximum of 64 bytes of data for a Bulk data payload, and a maximum of 256 bytes of data for an Isochronous data payload.

**Table 32. Data Packet Format**

| 8 Bits | 8 Bits | 0–1,023 Bytes | 16 Bits |
|---|---|---|---|
| Sync | PID | Data | CRC16 |

Handshake packets consist of only a sync and a PID. Handshake packets do not contain a CRC because the PID contains its own check field. They are used to report data transaction status, including whether data was successfully received, flow control, and stall conditions. Only transactions that support flow control can return handshakes.

The three types of handshake packets are: ACK, NAK, and STALL.

- ACK indicates that a data packet was received without bit stuffing, CRC, or PID check errors.
- NAK indicates that the UDC was unable to accept data from the host or it has no data to transmit.
- STALL indicates that the UDC is unable to transmit or receive data, and requires host intervention to clear the stall condition.

Bit stuffing, CRC, and PID errors are signaled by the receiving unit by omitting a handshake packet. Table 33 shows the format of a handshake packet.

**Table 33. Handshake Packet Format**

| 8 Bits | 8 Bits |
|---|---|
| Sync | PID |

## 4.10.5 Transaction Formats

Packets are assembled into groups to form transactions. Four different transaction formats are used in the USB protocol. Each is specific to a particular endpoint type: bulk, control, interrupt, and isochronous. Endpoint 0, by default, is a control endpoint and receives only control transactions.

The host controller initiates all USB transactions and transmission takes place between the host and UDC one direction at a time (half-duplex).

Bulk transactions guarantee error-free transmission of data between the host and UDC by using packet-error detection and retry. The host schedules bulk packets when there is available time on the bus. The three packet types used to construct bulk transactions are: token, data, and handshake.

The eight possible types of bulk transactions based on data direction, error, and stall conditions are shown in Table 34. (Packets sent by the UDC to the host are highlighted in boldface type. Packets sent by the host to the UDC are not boldfaced.)

**Table 34. Bulk Transaction Formats (Sheet 1 of 2)**

| Action | Token Packet | Data Packet | Handshake Packet |
|---|---|---|---|
| Host successfully received data from UDC | In | **DATA0/DATA1** | ACK |
| UDC temporarily unable to transmit data | In | None | **NAK** |
| UDC endpoint needs host intervention | In | None | **STALL** |
| Host detected PID, CRC, or bit-stuff error | In | **DATA0/DATA1** | None |
| UDC successfully received data from host | Out | DATA0/DATA1 | **ACK** |
| UDC temporarily unable to receive data | Out | DATA0/DATA1 | **NAK** |

### Table 34. Bulk Transaction Formats (Sheet 2 of 2)

| Action | Token Packet | Data Packet | Handshake Packet |
|---|---|---|---|
| UDC endpoint needs host intervention | Out | DATA0/DATA1 | **STALL** |
| | | | |
| UDC detected PID, CRC, or bit stuff error | Out | DATA0/DATA1 | None |

**NOTE:** Packets from UDC to host are **boldface**.

Isochronous transactions guarantee constant rate, error-tolerant transmission of data between the host and UDC. The host schedules isochronous packets during every frame on the USB, typically 1ms, 2 ms or 4 ms. USB protocol allows for isochronous transfers to take up to 90% of the USB bandwidth. Unlike bulk transactions, if corrupted data is received, the UDC will continue to process the corrupted data that corresponds to the current start of frame indicator. Isochronous transactions do not support a handshake phase or retry capability. The two packet types used to construct isochronous transactions are: token and data. The two possible types of isochronous transactions, based on data direction, are shown in Table 35.

### Table 35. Isochronous Transaction Formats

| Action | Token Packet | Data Packet |
|---|---|---|
| Host successfully received data from UDC | In | **DATA0/DATA1** |
| | | |
| UDC successfully received data from host | Out | DATA0/DATA1 |

**NOTE:** Packets from UDC to host are **boldface**.

Control transactions are used by the host to configure endpoints and query their status. Like bulk transactions, control transactions begin with a setup packet, followed by an optional data packet, then a handshake packet. Note that control transactions, by default, use DATA0 type transfers. Table 36 shows the four possible types of control transactions.

### Table 36. Control Transaction Formats, Set-Up Stage

| Action | Token Packet | Data Packet | Handshake Packet |
|---|---|---|---|
| UDC successfully received control from host | Setup | DATA0 | **ACK** |
| | | | |
| UDC temporarily unable to receive data | Setup | DATA0 | **NAK** |
| | | | |
| UDC endpoint needs host intervention | Setup | DATA0 | **STALL** |
| | | | |
| UDC detected PID, CRC, or bit stuff error | Setup | DATA0 | None |

**NOTE:** Packets from UDC to host are **boldface**.

Control transfers are assembled by the host by sending a control transaction to tell the UDC what type of control transfer is taking place (control read or control write), followed by two or more bulk data transactions. The first stage of the control transfer is the setup and the device must either respond with an ACK or if the data is corrupted, it sends no handshake.

The control transaction, by default, uses a DATA0 transfer, and each subsequent bulk data transaction toggles between DATA1 and DATA0 transfers. For a control write to an endpoint, OUT transactions are used. For control reads, IN transactions are used.

The transfer direction of the last bulk data transaction is reversed. It is used to report status and functions as a handshake. The last bulk data transaction always uses a DATA1 transfer by default (even if the previous bulk transaction used DATA1). For a control write, the last transaction is an IN from the UDC to the host, and for a control read, the last transaction is an OUT from the host to the UDC.

### Table 37. Control Transaction Formats

| Control Write | Setup | DATA (BULK OUT) | STATUS (BULK IN) |
|---|---|---|---|
| Control read | Setup | **DATA (BULK IN)*** | STATUS (BULK OUT) |

**NOTE:** Transactions from UDC to host are **boldface**.

Interrupt transactions are used by the host to query the status of the device. Like bulk transactions, interrupt transactions begin with a setup packet, followed by an optional data packet, then a handshake packet. Table 38 shows the eight possible types of interrupt transactions.

### Table 38. Interrupt Transaction Formats

| Action | Token Packet | Data Packet | Handshake Packet |
|---|---|---|---|
| Host successfully received data from UDC | In | **DATA0/DATA1** | **ACK** |
| UDC temporarily unable to transmit data | In | None | **NAK** |
| UDC endpoint needs host intervention | In | None | **STALL** |
| Host detected PID, CRC, or bit stuff error | In | **DATA0/DATA1** | None |
| UDC successfully received data from host | Out | DATA0/DATA1 | **ACK** |
| UDC temporarily unable to receive data | Out | DATA0/DATA1 | **NAK** |
| UDC endpoint needs host intervention | Out | DATA0/DATA1 | **STALL** |
| UDC detected PID, CRC, or bit stuff error | Out | DATA0/DATA1 | None |

**NOTE:** Packets from UDC to host are **boldface**.

## 4.10.6 UDC Device Requests

The UDC's control, status, and data registers are used only to control and monitor the transmit and receive FIFOs for endpoints 1 - 15. All other UDC configuration and status reporting is controlled by the host, via the USB, using device requests that are sent as control transactions to endpoint 0. Each data packet of a setup stage to endpoint 0 is 8 bytes long and specifies:

- Data transfer direction:
  - — Host to device
  - — Device to host

- Data transfer type
  - Standard
  - Class
  - Vendor
- Data recipient
  - Device
  - Interface
  - Endpoint
  - Other
- Number of bytes to transfer
- Index or offset
- Value: Used to pass a variable-sized data parameter
- Device request

Table 39 shows a summary of all device requests. Refer to the *Universal Serial Bus Specification Revision 1.1* for a full description of host-device requests.

### Table 39. Host-Device Request Summary

| Request | Name |
|---|---|
| SET_FEATURE | Enables a specific feature, such as device remote wake-up and endpoint stalls. |
| CLEAR_FEATURE | Clears or disables a specific feature. |
| SET_CONFIGURATION | Configures the UDC for operation. Used following a reset of the controller or after a reset has been signalled via the USB. |
| GET_CONFIGURATION | Returns the current UDC configuration to the host. |
| SET_DESCRIPTOR | Sets existing descriptors or adds new descriptors.<br>Existing descriptors include: †<br>• Device • Configuration • String<br>• Interface • Endpoint |
| GET_DESCRIPTOR | Returns the specified descriptor, if it exists. |
| SET_INTERFACE | Selects an alternate setting for the UDC's interface. |
| GET_INTERFACE | Returns the selected alternate setting for the specified interface. |
| GET_STATUS | Returns the UDC's status including:<br>• Remote wake-up • Self-powered • Data direction<br>• Endpoint number • Stall status |
| SET_ADDRESS | Sets the UDC's 7-bit address value for all future device accesses. |
| SYNCH_FRAME | Sets an endpoint's synchronization frame. |

† Interface and endpoint descriptors cannot be retrieved or set individually. They exist only embedded within configuration descriptors.

The UDC decodes most commands with no intervention required by the user. Via control endpoint 0, the user must decode and respond to the GET_DESCRIPTOR command. All setup commands are passed through to the user so that software can be configured properly.

## 4.10.7 Configuration

In response to the GET_DESCRIPTOR command, the user sends back a description of the UDC configuration. *The UDC can physically support more data-channel bandwidth than the USB will allow.* When responding to the host, the user must be careful to specify a legal USB configuration.

For example, if the user specifies a configuration of six isochronous endpoints of 256 bytes each, the host will not be able to schedule the proper bandwidth and will not take the UDC out of Configuration 0. The user must determine which endpoints to not tell the host about, so that they will not get used.

Another option, especially attractive for isochronous endpoints, is to describe a configuration of less than 256 bytes maximum packet to the host. The direction of the endpoints is fixed and the UDC will physically support only the following maximum packet sizes:

- Interrupt endpoints — 8 bytes
- Bulk endpoints — 64 bytes
- Isochronous endpoints — 256 bytes

In order to increase flexibility, the UDC supports a total of four configurations. While each of these configurations is identical within the UDC, the software can be used to make three distinct configurations. Configuration 0 is a default configuration of endpoint 0 only.

For a detailed description of the configuration descriptor, see the USB 1.1 Specification.

## 4.10.8 Endpoint Stall Feature

Throughout this document the word “stall” is used, as in the USB specification, with two meaning. One usage signifies a special handshake packet, referred to as STALL, and the other signifies a functional stall.

A device uses the STALL handshake in one of two distinct occasions.

The first case — known as “functional stall” — is when the *Halt* feature, associated the endpoint, is set. A special case of the functional stall is the “commanded stall.” Commanded stall occurs when the host explicitly sets the endpoint’s *Halt* feature using the SET_FEATURE command.

Once a function’s endpoint is halted, the function must continue returning STALL packets until the condition causing the halt has been cleared through host intervention (using SET_FEATURE). This can happen both for IN and OUT endpoints. In the case of IN endpoints, the endpoint sends a STALL handshake immediately after receiving an IN token. For OUT endpoints the STALL handshake is sent as soon as the data packet after the OUT token is received.

**Figure 45. STALL on IN Transactions**

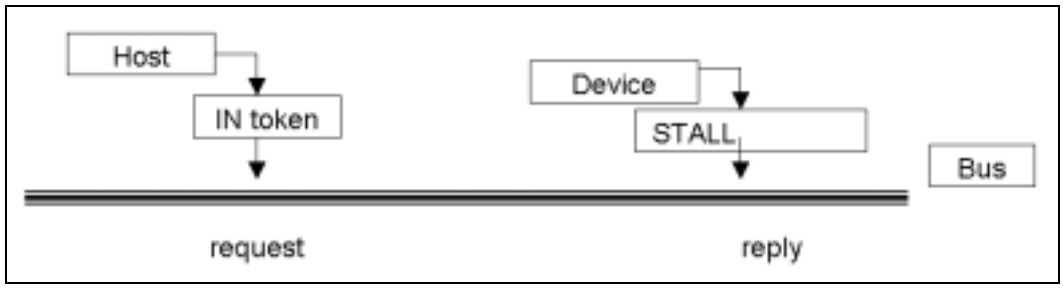
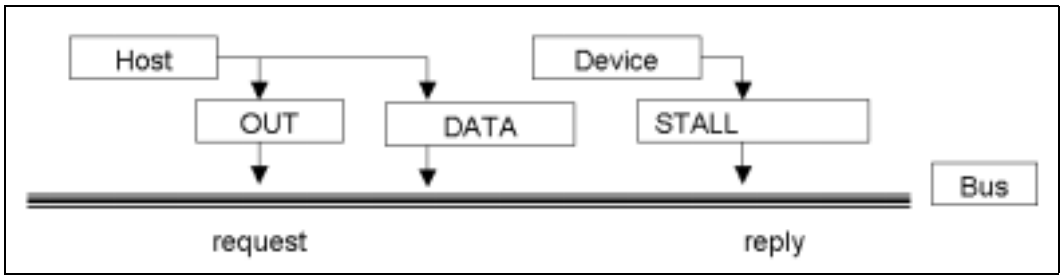


**Figure 46. STALL on OUT Transactions**




The second case of a STALL handshake is known as a “protocol stall” and is unique to control pipes. Protocol stall differs from functional stall in meaning and duration.

A protocol STALL is returned during the Data or Status stage of a control transfer, and the STALL condition terminates at the beginning of the next control transfer (Setup). Protocol stalls are usually sent to notify the host that a particular USB command is malformed or not implemented.

## 4.10.9 Frame Synchronization

The SYNCH_FRAME request is used by isochronous endpoints that use implicit-pattern synchronization. The isochronous endpoints may need to track frame numbers in order to maintain synchronization.

Isochronous-endpoint transactions may vary in size, according to a specific repeating pattern. The host and endpoint must agree on which frame begins the repeating pattern. The host uses this request to specify the exact frame on which the repeating pattern begins.

The data stage of the SYNCH_FRAME request contains the frame number in which the pattern begins. Having received the frame number, the device can start monitoring each frame number sent during the SOF. This is recorded in the frame counter and made available through specific driver and device manager functions (see ixUSBFrameCounterGet and ixUSBDMFrameCounterGet).

## 4.10.10 SNMP

Status statistics management in the USB component will follow the framework described in the SNMP functional specification. This section lists the specific items that a client of the USB component can query, set, or be informed, when a change happens.

## 4.10.11 IxUSBManagerInfo

enum ixUSBManagerInfo {…} will contain the items listed in Table 40. The table rows above the heavy line show the entities defined in the Definitions of Managed Objects for the Universal Serial Bus (USB) Interface.

**Table 40. SNMP Entities (Sheet 1 of 2)**

| Entity Name | Meaning | Type |
|---|---|---|
| usbNumber_TRO | Number of ports, regardless of their current state in the USB general port table | UINT32 |
| usbPortIndex_TRO | Unique identifier of the USB port hardware.<br>By convention and if possible, hardware port numbers map directly to external connectors. | UINT32 |
| usbPortType_TRO | Type of the USB port | • Host(1)<br>• Device(2)<br>• Hub(3) |
| usbPortRate_TRO | USB port rate that could be low-speed(1) for 1.5 Mbps, full-speed(2) for 12 Mbps, or high-speed(3) for USB 2.0 | • Low-speed (1)<br>• Full-speed(2)<br>• High-speed(3) |
| usbDeviceIndex_TRO | Index is identical to usbPortIndex for the correspondent USB port | UINT32 |
| usbDevicePower_TRO | Way USB device port is powered | • Unknown(1)<br>• Self-powered(2)<br>• Bus-powered(3) |
| usbDeviceVendorID_TRO | USB device port vendor HEX-formatted string as it is provided to the USB host by the USB device | OCTET STRING |
| usbDeviceProductID_TRO | Product ID HEX-formatted string as it is provided to the USB host by the USB device | OCTET STRING |
| usbDeviceNumberConfigurations_TRO | Total number of configurations the USB port supports. Device port should support at least one configuration | UINT32 |
| usbDeviceActiveClass_TRO | Object returns USB Device Class type of the active configuration. | • Other(1)<br>• cdc(2) |
| usbDeviceStatus_TRO | Current status of the USB device state machine. | • unattached(1)<br>• attached(2)<br>• powered(3)<br>• default(4)<br>• addressed(5)<br>• configured(6)<br>• suspended(7) |
| usbDeviceEnumCounter_TRO | Total number reconnections (enumerations) since device is operational. | UINT32 |
| usbDeviceRemoteWakeup_TRO | If set to true(1), the device supports Remote Wakeup function. If set to false(2), the device doesn't support it. | BOOL |
| usbDeviceRemoteWakeupOn_TRO | If set to true(1), the remote wakeup function is activated by the host. If set to false(2), remote wakeup function is not active. | BOOL |
| usbDeviceFrameCounter_TRO | Device 11-bit frame counter | UINT32 |
| usbDeviceActiveConfiguration_TRO | Currently selected configuration index | UINT32 |

**Table 40. SNMP Entities (Sheet 2 of 2)**

| Entity Name | Meaning | Type |
|---|---|---|
| usbDeviceActiveInterface_TRO | Currently selected interface index | UINT32 |
| usbDeviceAddress_TRO | Device address | UINT32 |
| usbDeviceEndpointNumber_TRO | Number of endpoints in the device | UINT32 |
| usbDeviceEndpointInfo_TRO | Array of endpoint descriptive information | usbEndpointInfo |
| usbDeviceEndpointStats_TRO | Array of endpoint statistics | usbEndpointStats |

**NOTE:** Types described by enumerations are mapped to SNMP enums.

usbEndpointInfo is defined by the entities listed in Table 41.

**Table 41. SNMP Definition of usbEndpointInfo**

| Entity Name | Meaning | Type |
|---|---|---|
| usbEndpointType | Endpoint type | control(1), bulk(2), interrupt(3), isochronous(4) |
| usbEndpointDirection | Endpoint data transfer direction | in(1), out(2), in_out(3) |
| usbEndpointMaxPacketSize | Maximum packet size supported by endpoint | UINT32 |
| usbEndpoinStall | Endpoint stall status (true(1) if stalled, false(2) if not stalled) | BOOL |
| usbEndpointFlags | Endpoint flags | UINT32 |

**NOTE:** Types described by enumerations are mapped to SNMP enums.

usbEndpointStats is defined by the entities given in Table 42.

**Table 42. SNMP Definition of usbEndpointStats**

| Entity Name | Meaning | Type |
|---|---|---|
| usbEndpointRxCounter | Received packets counter | UINT32 |
| usbEndpointTxCounter | Transmitted packets counter | UINT32 |
| usbEndpointDroppedTxCounter | Packets supposed to be transmitted but were dropped | UINT32 |
| usbEndpointDroppesdRxCounter | Received and dropped packets counter | UINT32 |
| usbEndpointRxByteCounter | Received bytes counter | UINT32 |
| usbEndpointTxByteCounter | Transmitted bytes counter | UINT32 |

### 4.10.11.1 IxUSBManagerTraps

There are no SNMP traps defined for this component.

### 4.10.11.2 SNMP Interface Functions

These functions follow the prototypes laid out in the SNMP functional specification.

The data type ixUSBManagerInfo is defined in SNMP functional specification.

## 4.10.12 Error Handling

The USB API calls return the IX_FAIL error code after detecting errors. It is the responsibility of the user to implement appropriate error handling.

Detailed error codes are used to report both USB Driver and USB Device Manager errors. They are provided in the *lastError* field of the *USBDevice* structure that must be passed by the user in every API call. When the API calls are successful the *lastError* field is assigned the IX_SUCCESS value.

According to current coding standards as of August 2001, IX_SUCCESS value is enumerated to 0.

**Table 43. Detailed Error Codes**

```
#ifndef IX_USB_ERROR_BASE
#define IX_USB_ERROR_BASE 4096
#endif /* IX_USB_ERROR_BASE */

/* error due to unknown reasons  */

#define IX_USB_ERROR(IX_USB_ERROR_BASE + 0)

/* invalid USBDevice structure passed as parameter or no device present */
#define IX_USB_INVALID_DEVICE (IX_USB_ERROR_BASE + 1)

/* no permission for attempted operation */
#define IX_USB_NO_PERMISSION(IX_USB_ERROR_BASE + 2)

/* redundant operation */
#define IX_USB_REDUNDANT(IX_USB_ERROR_BASE + 3)

/* send queue full */
#define IX_USB_SEND_QUEUE_FULL(IX_USB_ERROR_BASE + 4)

/* invalid endpoint */
#define IX_USB_NO_ENDPOINT(IX_USB_ERROR_BASE + 5)

/* no IN capability on endpoint */
#define IX_USB_NO_IN_CAPABILITY(IX_USB_ERROR_BASE + 6)

/* no OUT capability on endpoint */
#define IX_USB_NO_OUT_CAPABILITY(IX_USB_ERROR_BASE + 7)

/* transfer type incompatible with endpoint */
#define IX_USB_NO_TRANSFER_CAPABILITY(IX_USB_ERROR_BASE + 8)

/* endpoint stalled */
#define IX_USB_ENDPOINT_STALLED(IX_USB_ERROR_BASE + 9)

/* invalid parameter(s) */
#define IX_USB_INVALID_PARMS(IX_USB_ERROR_BASE + 10)
```

**NOTE:** “Error due to unknown reasons” — This code is also used when there is only one possible error reason and the error was already signaled by the IX_FAIL return code.

### 4.10.12.1 USB Data Flow

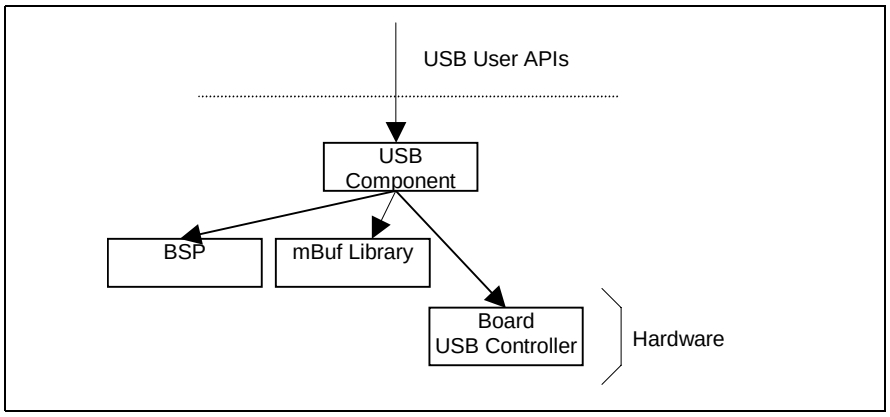
The USB device driver is a self-contained component with no interactions with other data components. Figure 47 shows the dependencies for this USD component.

### 4.10.12.2 Dependencies and Data Flows

The USB component has no explicit dependencies to other components. The USB device is a memory mapped device on the AMBA peripheral bus. It will not interact directly with the NPEs. Any data path between USB and other components must be performed via the Intel XScale core.

The USB device driver is a self-contained component with no interactions with other data components. Figure 47 shows the dependencies for this USD component.

**Figure 47. USB Dependencies**




## 4.11 ixUart

The IXP42X product line processors' UARTs have been modeled on the industry standard 16550 UART. There are, however, some differences between them which prevents the unmodified use of 16550-based UART drivers. They will support baud rates between 9,600 bps and 912.6 Kbps.

The higher data rates allow the possibility of using the UART as a connection to a data path module, such as Bluetooth*. While the UART is instantiated twice on the IXP42X product line processors, the same low-level routines will be used by both. The default configuration for IXP42X product line processors will be:

- UART0 — Debug Port (console)
- UART1 — Fast UART (e.g., Bluetooth)

Any combination of debug or high-speed UART, however, could be used.

A generic reference implementation is provided that can be used as an example for other implementations/operating systems.

These routines are meant to be stand-alone, such that they do not require an operating system to execute. If a new operating system is later added to those supported, these routines can be easily modified to link in to that platform, without the need for extensive rework.

The first release of the driver will provide generic support for polled and loop back mode only. (Interrupt driven I/O will *not* be supported.)

## 4.11.1 Interface Description

The API covers the following functions:

- Device initialization
- UART char output
- UART char input
- UART IOCTL
- Baud rate set/get
- Parity
- Number of stop bits
- Character length 5, 6, 7, 8
- Enable/disable hardware flow control for Clear to Send (CTS) and Request to Send (RTS) signals

## 4.11.2 UART Data Flow

The UART device driver will not form part of the data path in IXP400 software v1.2.2. It will enable OS-console operations.

### 4.11.2.1 FIFO Versus Polled Mode

The UART supports both FIFO and polled mode operation. Polled mode is the simpler of the two to implement, but is also the most processor-intensive since it relies on the Intel XScale core to check for data.

The device's Receive Buffer Register (RBR) must be polled at frequent intervals to ascertain if data is available. This must be done frequently to avoid the possibility of buffer overrun. Similarly, it checks the Transmit Buffer Register (TBR) for when it can send another character.

The FIFO on the IXP42X product line processors UART is 64 bytes deep in both directions. The transmit FIFO is 8 bits wide and the receive FIFO is 11 bits wide. The transmit FIFO is wider to accommodate the potentially largest data word (i.e., including optional stop bits and parity 8+2+1=11).

Interrupts can occur in one of two ways. One is when the FIFO has reached its programmed trigger level (set by the FIFO Control Register [FCR]). The other is when a character timeout has occurred (also set in the FCR). The driver will implement both modes of operation.

The default setup for the UART is:

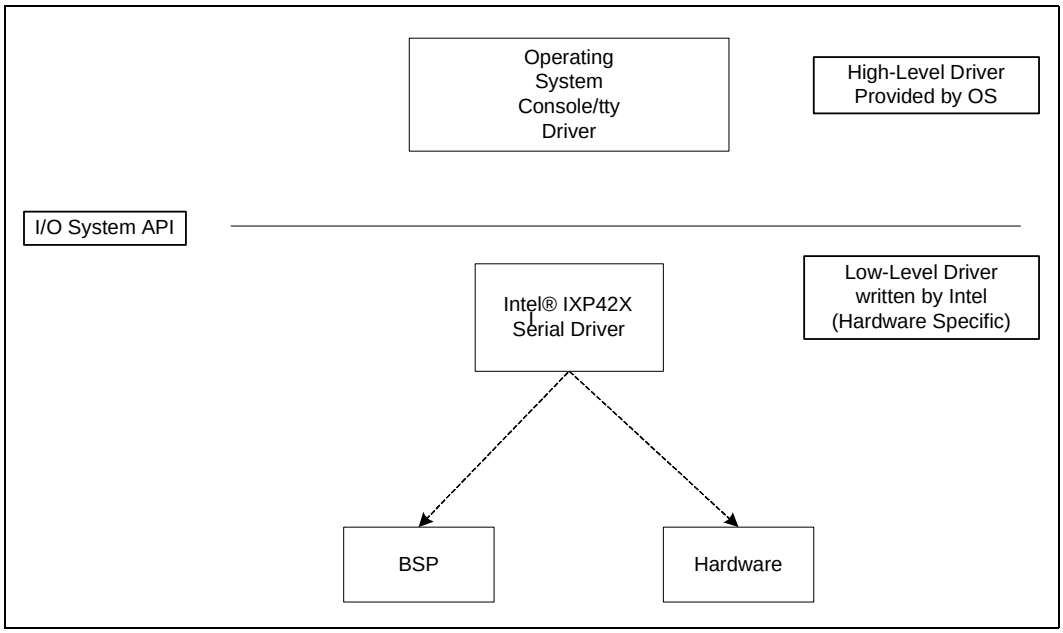
- 9,600 bps baud rate
- 8-bit data word
- One stop bit

- No parity
- No flow control
- Interrupt mode (Polled for generic interface)

These settings are part of the BSP specific constants for the IXP42X product line processors UART. They may be changed at run time.

## 4.11.3 Dependencies

**Figure 48. UART Component Dependencies**



# 4.12 ixCryptoAcc

## 4.12.1 Overview

The Security Hardware Accelerator access component (ixCryptoAcc) provides support for authentication and encryption/decryption services needed in IPSec stacks. IPSec clients can offload the task of encryption/decryption by using the DES or AES coprocessor, depending on the cryptographic algorithm used. The IPSec clients also can offload the task of authentication by using the hashing coprocessor, instead of manipulating the software.

The cryptographic acceleration is provided by the Security Hardware Accelerator coprocessors in NPE B of the Intel® IXP42X Product Line Processors.

The IxCryptoAcc component provides control and access to the features supported by the NPE:

- Operating modes:
  - — Encryption only
  - — Decryption only
  - — Authentication calculation only
  - — Authentication check only
  - — Encryption followed by authentication calculation
  - — Authentication check followed by decryption
- Cryptographic algorithms:
  - — DES (64-bit block cipher size, 64-bit key)
  - — Triple DES (64-bit block cipher size; three keys, 64-bit each)
  - — AES (128-bit block cipher size; key sizes: 128-, 192-, 256-bit)
- Mode of operation for encryption and decryption:
  - — ECB
  - — CBC
  - — CTR (for AES algorithm *only*)
  - — Because the IETF is not currently using the CFB and OFB chaining modes in any of the IPSec implementation, CFB and OFB modes will not be supported by the access component.
- Authentication algorithms:
  - — HMAC-SHA1 (512-bit data block size, 160-bit key size)
  - — HMAC-MD5 (512-bit data block size, 128-bit key size)
- Reports operation failure to client

The ixCryptoAcc component also supports a maximum of 10,000 security associations (tunnel) simultaneously. (A Security Association [SA] is a simplex “connection” that affords security services to the traffic carried by it.)

**Figure 49. Generalized IPSec Stack Architecture**

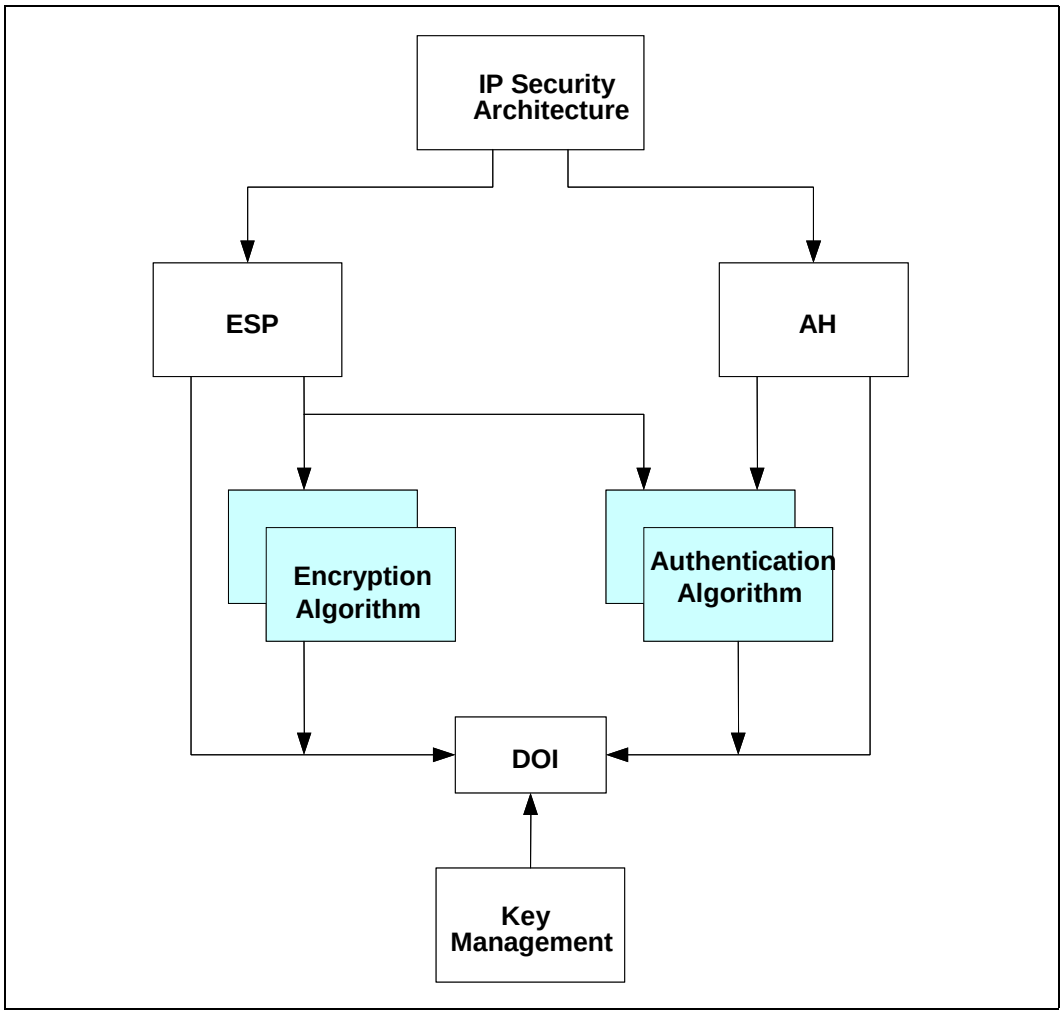



Figure 49 shows the relationship of encryption and authentication algorithm within the IPSec protocol.

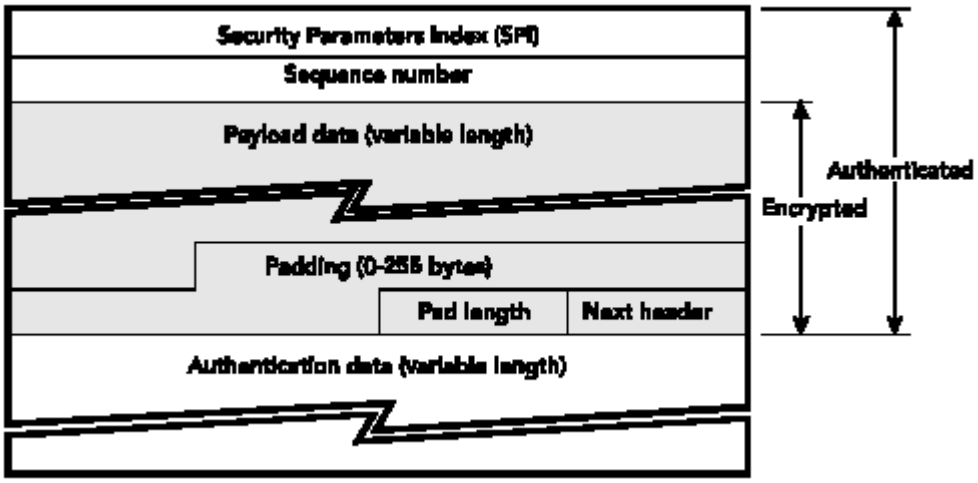
### 4.12.1.1 Authentication Mode

IPSec standards have defined new packet formats. The authentication header (AH) provides data integrity and the encapsulating security payload (ESP) provides confidentiality and data integrity. Both AH and ESP provide data integrity, through the SHA1 and MD5 algorithms.

The IxCryptoAcc component supports two different modes of authentication. The ICV is calculated through SHA1 or MD5 and inserted into the AH packet and ESP packet.

In ESP authentication mode, the ICV is appended at the end of the packet, which is after ESP trailer if encryption required.

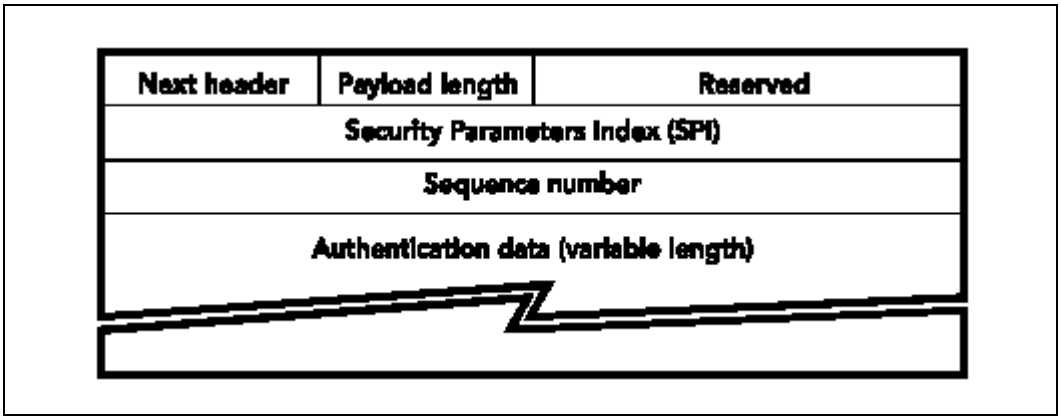
**Figure 50. ESP Packet Structure**




In AH mode, the ICV value is part of the authentication header. AH is embedded in the data to be protected. This results in AH being included for ICV calculation which means the authentication data field (ICV value) must be cleared before executing the ICV calculation. The same applies to the ICV verification — the authentication data needing to be cleared before the ICV value calculated and compared with the original ICV value in the packet. If the ICV values don't match, authentication is failed.

NPE determines where to insert the ICV value, based on the ICV offset specified in interface.

**Figure 51. Authentication Header**



#### 4.12.1.2 Security Association Database Management

The Security Hardware Accelerator client will allocate the mbuf to be passed into the Security Hardware Accelerator. The Hardware Accelerator in turn will allocate memory for the CCD, where all the information, required for cryptographic services, are stored.

The component assumes that the allocated mbufs are sufficient in length and no checking has been put in place for the mbufs length in the production codes. There is, however, mbuf checking in the DEBUG mode of the production codes. This is done to ensure the codes are optimized in the data path.

When appending the ICV at the end of the payload, it is assumed that the mbuf is length is sufficient and will not cause memory segmentation. The ICV offset should be within the length of the mbuf.

#### 4.12.1.3 Buffer Management

The mbuf buffer format is expected to be used between the IxCryptoAcc access component and the client. All buffers used between the IxCryptoAcc access component and clients are allocated and freed by the clients.

The client passes a buffer to IxCryptoAcc, when it requests hardware-accelerator services, and the IxCryptoAcc component returns the buffer to the client when the requested job is done. The encrypted / decrypted payload is written into the source buffer, or destination buffer depending on the transfer mode in-place, before returning the buffer to the client.

#### 4.12.1.4 Error Handling

IxCryptoAcc returns an error type to the client and the client is expected to handle the error. Internal errors will be reported using an IxCryptoAcc-specific, error-handling mechanism listed in IxCryptoAccStatus.

#### 4.12.1.5 Endianness

All the data structures passed into the component are defined in host's byte order. The mode support by the component is big endian.

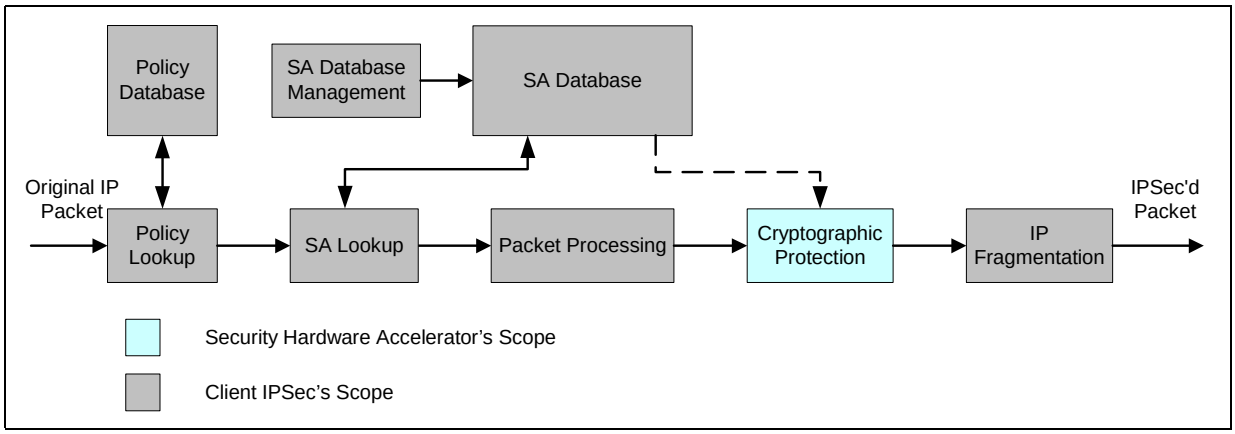
#### 4.12.1.6 Import and Export Regulations of Cryptographic Technology

Some of the cryptographic technologies provided by this software (such as 3DES and AES) may be subjected to both export controls from the United States and import controls worldwide. Where local regulations prohibit, some described modes of operation may be disabled.

#### 4.12.1.7 NPE-based Security Hardware Accelerator and IxCryptoAcc Scope

When deploying security-related applications, the generalized architecture in Figure 52 is used. The figure shows the scope and the roles played by the NPE and the ixCryptoAcc component in an IPSec application.

**Figure 52. Security Coprocessor and IPSec Stack Scope**




The IPSec protocol stack provides security for the transported packets by encrypting and authenticating the IP payload. Before an IP packet is sent out to the public network, it is processed by the IPSec application (known as "client," in this scenario) to encapsulate the IP packet into the ESP or AH packet format.

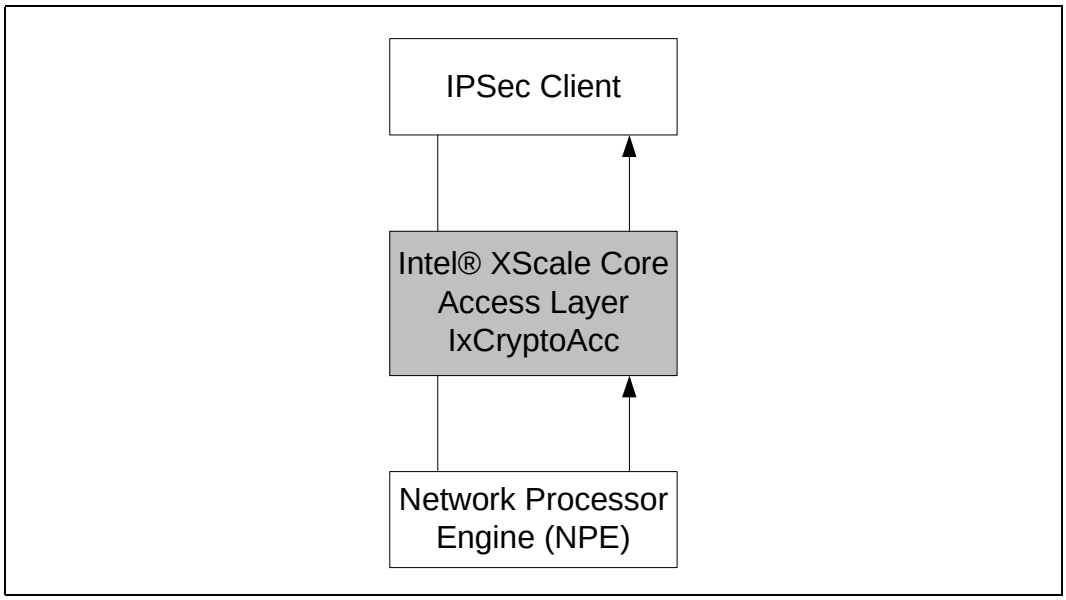
The NPE does not have access to the Security Association (SA) database. The information in the SA database that is required for the cryptographic protection will be passed in, via the client API, to the NPE (Cryptographic Protection Block).

The client looks up the policy and SA database to determine the mode of transporting packets, the IPSec protocol (ESP or AH), etc. The client determines use of the transport or tunnel mode from the policy. The mode is transparent to ixCyptoAcc component.

The client processes the IP packet into ESP or AH packet format, the IP packet is padded accordingly (if ESP is chosen), and the IP header mutable fields are handled (if AH). Then, the NPE executes cryptographic protection algorithms (encryption and/or authentication), based on the SA information. This is done regardless of the mode.

The client sends out the protected IP packet after the cryptographic protection is applied. If the IP packet is too large in size, the client fragments the packet before sending.

**Figure 53. IxCrypto Component Interface Layering**




The main functionality of the ixCryptoAcc component, is to provide access to the Network Processor Engine (NPE) for the cryptographic services. The information — required for cryptographic processes — is passed to the ixCryptoAcc in calls made by the IPSec client in the API. The ixCrypoAcc component stores the information in a Cryptographic Context Database (CCD).

The NPE accesses the CCD and uses the information to perform the cryptographic services.

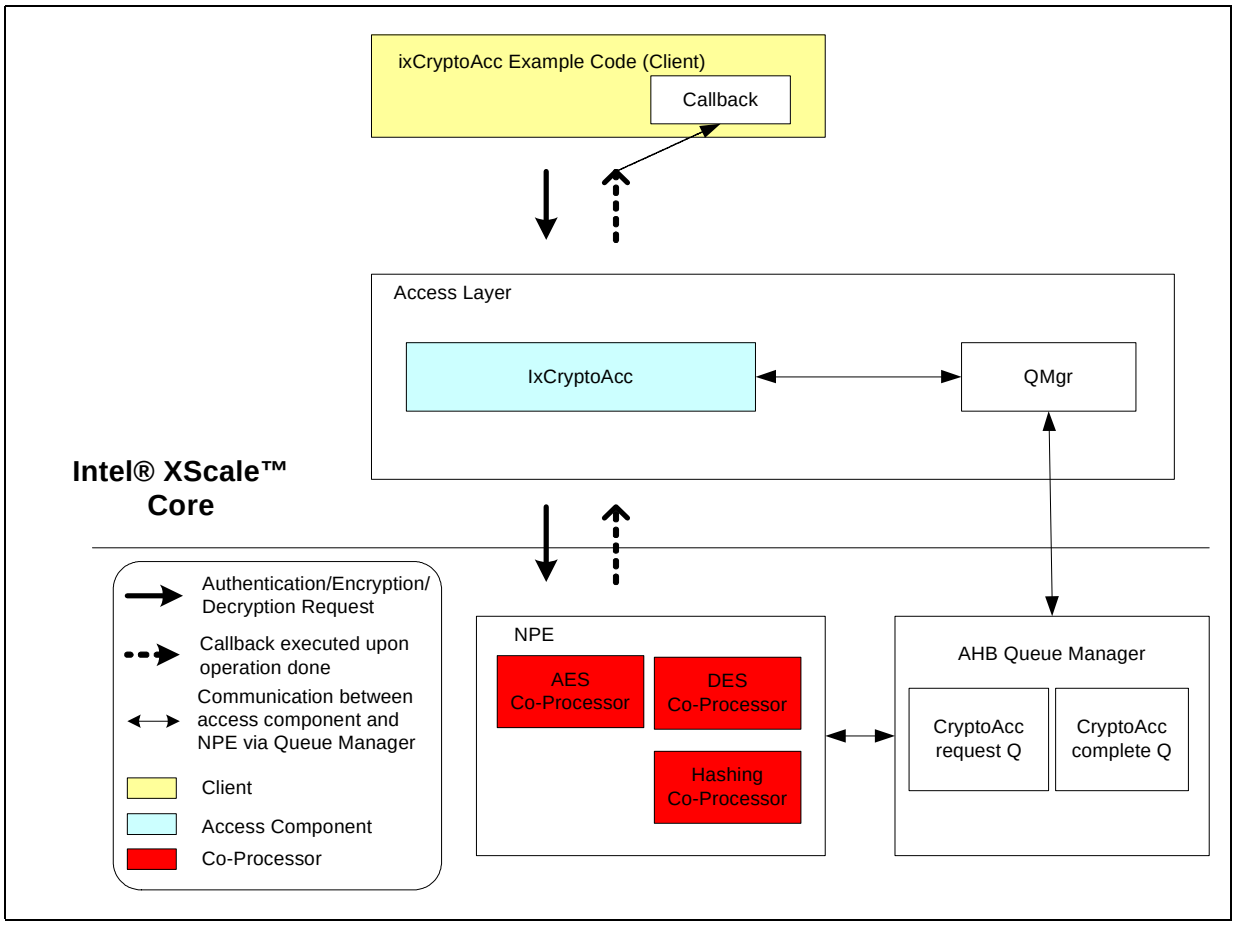
### 4.12.1.8 IxCryptoAcc Component Architecture

Packets for encryption/decryption and authentication are prepared by the client and passed to the IxCryptoAcc component via the component API. IxCryptoAcc invokes the NPE to gather the data from SDRAM, with appropriate key information, and the NPE performs encryption/decryption and authentication using both the DES/AES and hashing coprocessors.

The resulting data is stored back into the SDRAM.

The architecture is shown in Figure 54.

**Figure 54. Detailed IxCryptoAcc Component Architecture**



## 4.12.2 Functional Interface

### 4.12.2.1 Overview

The API provided by the IxCryptoAcc component includes the following functionality. The functionality provides four basic services:

- Initialization
- Registration (creation) of a security context
- Unregistration (freeing) of a security context
- Performance of security services, as defined by the context

The complete services include:

- Initialization — Initializes the IxCryptoAcc access component.
- Registration of the security context — Allocates a hardware accelerator context, creating an entry in the CCD.

  This associates a Security Association configuration to a unique SA ID. The arguments are:

  — Operation: {Encrypt, Decrypt, Authenticate, Encrypt&Authenticate, Authenticate&Decrypt}

  — Cryptographic algorithm: {DES, 3DES, AES}

  — Cipher mode: {ECB, CBC, CTR}

  — Cipher key

  — Cipher key length

  — Cipher block length

  — Security Parameter Index from SAD

  — Authentication algorithm: {SHA1, MD5}

  — Hash key

  — Hash key length

  — Digest length

  — Initialization vector length

  — mBuf pointers

  — Two Callback functions

  ***Note:*** The initialization vector is needed in CBC and CTR mode of encryption/decryption operations. Initial chaining variables used in authentication algorithms and reverse keys for AES are derived in the access component using the NPE. Therefore, those parameters are not supplied in the API.

- Unregistration of the security context — Removes the hardware accelerator context from the CCD.
- Performance of security context services — Initiates the hardware accelerator services to begin processing of the packet data. This interface requires the arguments:
  - — Crypto Context ID
  - — mbuf pointer
  - — Offset and length of data for forward/reverse cipher operation
  - — Initialization vector
  - — Offset and length of data for forward/reverse authentication operation
  - — ICV offset

#### 4.12.2.2 API

A summary of the Application Programming Interface (API) is provided in Table 44. Details of the full API are provided in Appendix A, "Application Programming Interfaces."

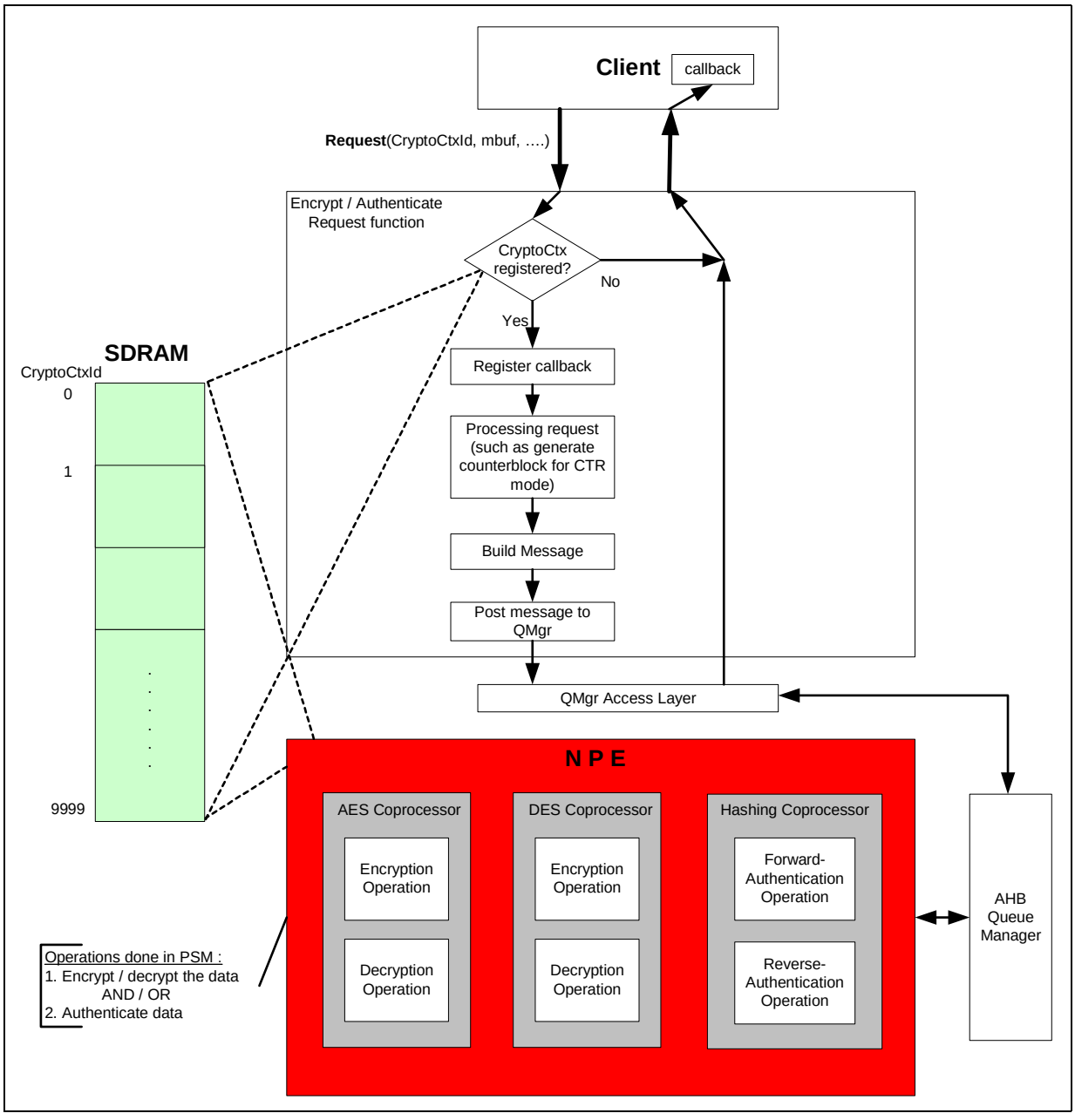
**Table 44. IxCryptoAcc Component API Summary**

| Name | Purpose | Parameters | Return |
|---|---|---|---|
| IxCryptoAccInit | Initializes the security-access software. | (None) | Status |
| ixCryptoAccCtxRegister | Sets up a security context. | Encryption and Authentication details, such as keys, modes, etc. | ContextId (Callback made once context is setup) |
| ixCryptoAccCtxUnregister | Deletes a security context. | ContectId | Status |
| ixCryptoAccAuthCryptPerform | Requests an encryption and/or authentication operation. | ContextId, Buffer to use, offsets to data within buffer, any initialization vectors needed | Callback with status and buffer made once operation complete |

### 4.12.2.3 Basic API Flow

The flow of the calls through the API are detailed in Figure 55.

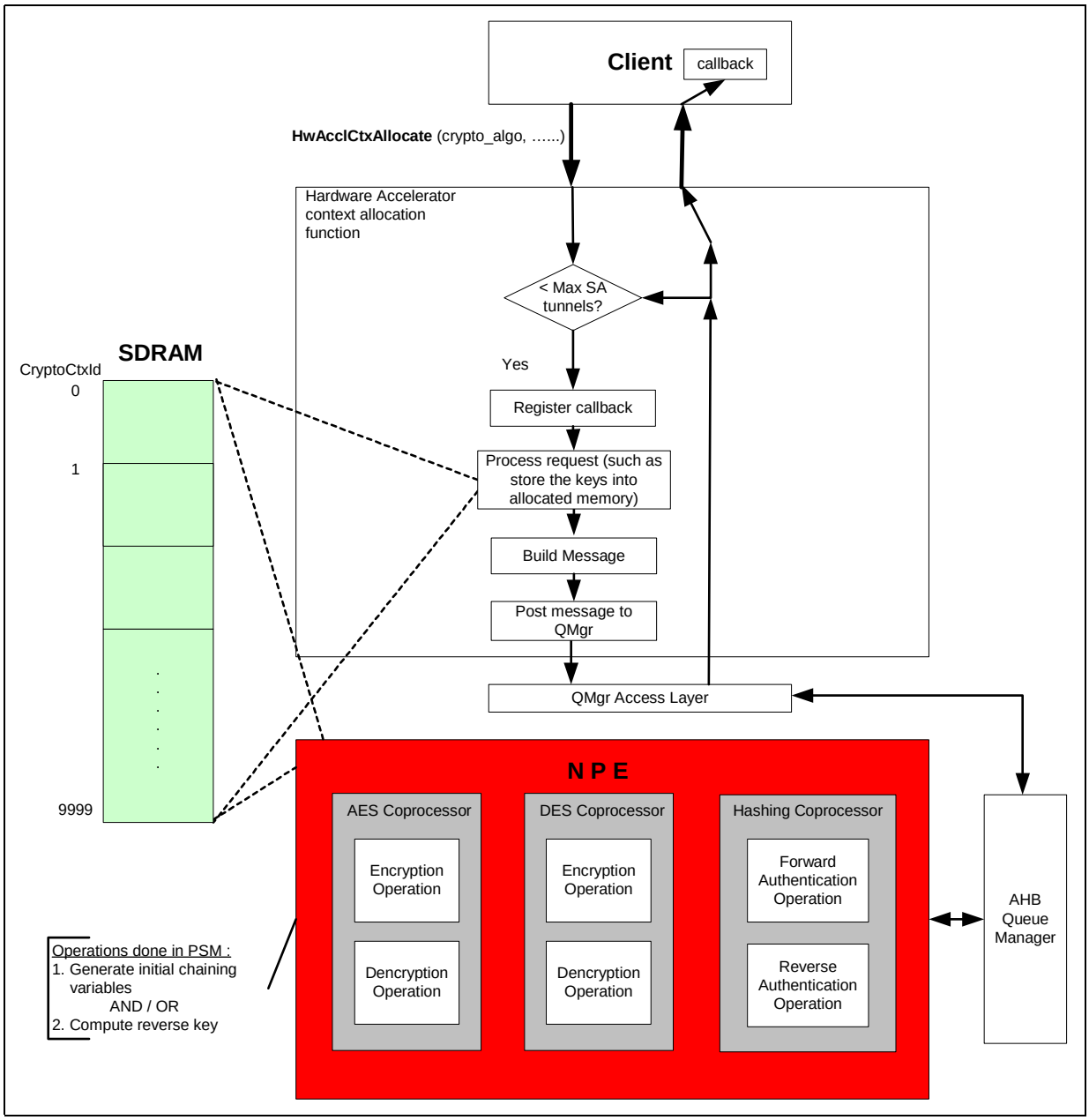
**Figure 55. Basic IxCryptoAcc API Call Process Flow**



### 4.12.2.4 CCD Update Flow

Figure 56 shows the flow through the ixCryptoAcc API when adding to the CCD.

**Figure 56. IxCryptoAcc API Call Process Flow for CCD (SA) Updates**



#### 4.12.2.5 Assumptions, Restrictions, and Dependencies

The CryptoAcc functionality has a number of key assumptions and restrictions:

- The API assumes that all requests originate from the same Intel XScale core task (No Re-entrancy).

  The critical section is not protected, thus only one task shall access the critical section at a time.

- Client handles IP mutable fields.

  If a field in the IP header is modified during transit, the value of the field is set to 0 for purposes of the ICV computation. These fields are considered as mutable fields (RFC 2402).

- Client pads the IP datagram to be a multiple of the cipher block size, using ESP trailer for encryption (RFC 2406, explicit padding).

- NPE pads the IP datagram to be a multiple of the block size, specified by the authentication algorithm (RFC 2402, implicit padding).

- Client provides an initialization vector to the access component for the DES or AES algorithm, in CBC mode and CTR mode.

- NPE produces the initial values/chaining variables used in authentication algorithms through the key provided.

- NPE computes the reverse keys or derives the keys from the key provided for AES algorithm.

- NPE inserts the ICV for a forward-authentication operation and verifies the ICV for a reverse-authentication operation.

  Status is returned in the CryptoAcc complete Q entry.

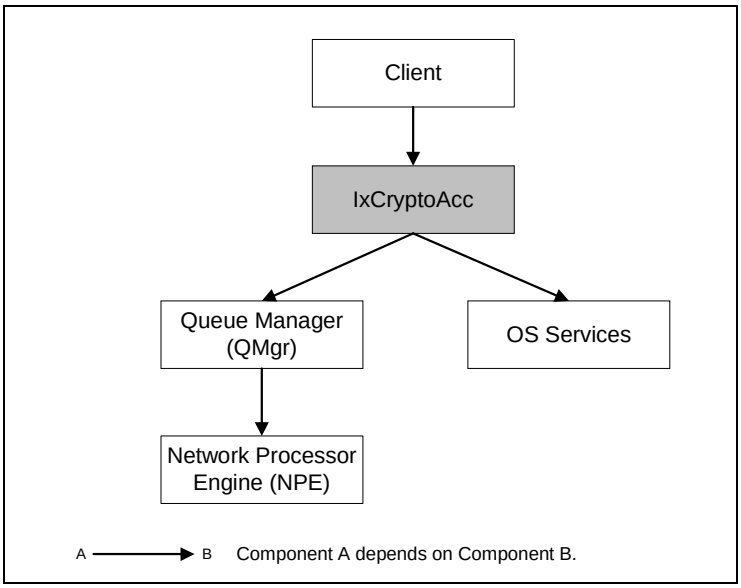
- Two queues have been reserved for IxCryptoAcc component.

- Client collects the statistic for authentication failure.

### 4.12.2.6 Dependencies

Figure 57 shows the functional dependencies of IxCryptoAcc component.

**Figure 57. IxCryptoAcc Component Dependencies**




The dependency diagram can be summarized as follows:

- Client component will call IxCryptoAcc for cryptographic services. NPE will perform the encryption, decryption, and authentication process via QMgr.
- ixCryptoAcc depends on the QMgr component to configure and use the hardware queues to access the NPE.
- OS Services component is used for error-handling and reporting.
- When the Queue Manager (QMgr) is full, the Hardware Accelerator will return IX_CRYPTO_ACC_QUEUE_FULL to the client. The client will have to re-send the data to be encrypted or decrypted or authenticated, after a random interval. This status will be invoked by ixCryptoAccAuthCryptPerform API.

### 4.12.2.7 Issues

The access component does not support re-entrancy. In the interests of performance, however, it would be good to support re-entrancy according to the IPSec stack implementation by Intoto*.

In the Intoto stack, there is one possible usage that needs re-entrancy: when the IPSec and IKE tasks call the access component at the same time. This issue will be addressed at later stage, as further investigation is needed, concerning the of supporting of re-entrancy for the API.

## 4.12.3 Use Models

### 4.12.3.1 IxCryptoAcc Data Flow Overview

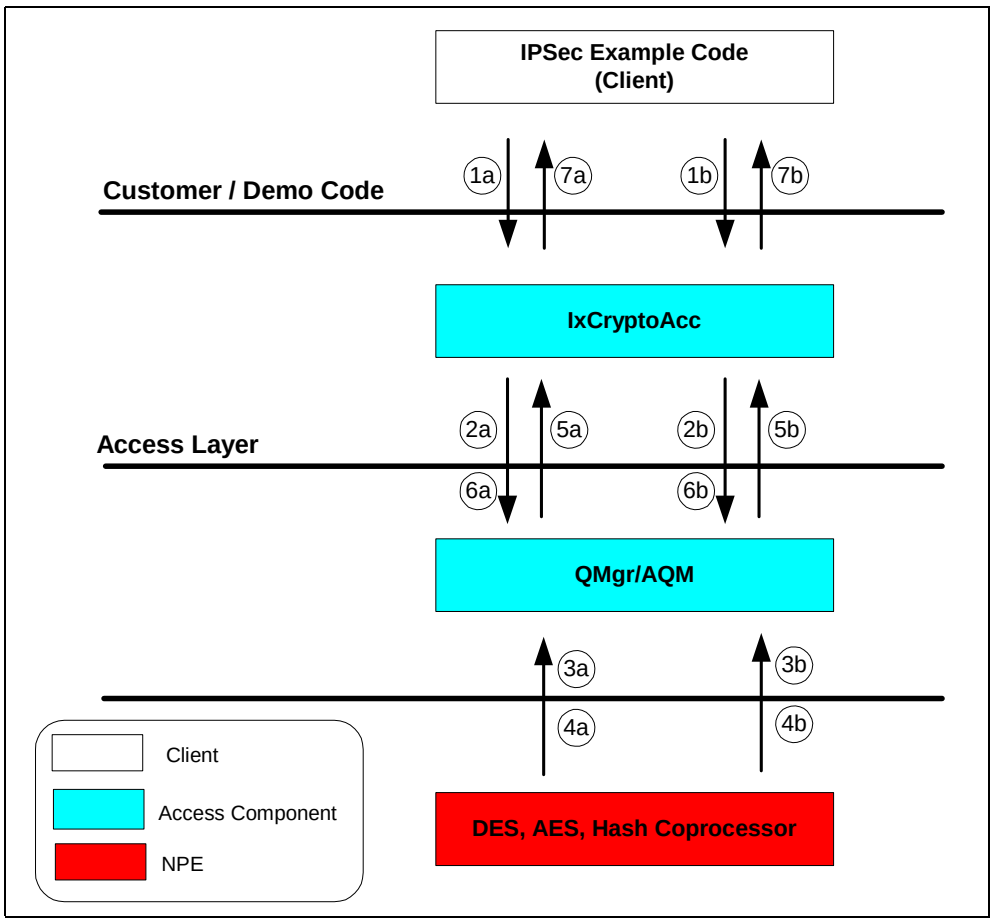
#### 4.12.3.1.1 Description

Client requests to encrypt/decrypt/authenticate the IP datagram using security hardware accelerator services provided by IxCryptoAcc component through DES, AES coprocessor, and hashing coprocessor.

#### 4.12.3.1.2 Preconditions

- QMgr and IxCryptoAcc have been initialized.
- IxCryptoAcc has hooked into the QMgr and configured the threshold at which it wants notification via QMgr for CryptoAcc complete Queue.

**Figure 58. IxCrytoAcc Execution Model**



#### 4.12.3.1.3 Sequence Details

**Hardware Accelerator Context Configuration Execution Model**

1a. The client registers Security Association Configuration information with IxCryptoAcc component.

Security Association configuration information is provided to IxCryptoAcc.

2a. IxCryptoAcc determines the crypto algorithms and writes the descriptor to CryptoAccQ via QMgr.

3a. NPE awakes on condition, reads the descriptor from AQM, and begins reverse keys computation operation for AES (if AES is chosen) and initial chaining variables generation operation for authentication algorithm.

4a. Once all the operations complete, the NPE places descriptor to the CryptoAcc complete Q and returns to pending state.

5a. QMgr receives an interrupt from AQM through polling mechanism. The QMgr dispatches notification to IxCryptoAcc.

6a. IxCryptoAcc reads the descriptor from AQM via QMgr.

7a. IxCryptoAcc returns an ID (CryptoCtxId) and registration status to client via callback function.

**Hardware Accelerator Service Request Execution Model**

1b. The client initiates a request to authenticate, encrypt, or decrypt the data in mbuf using services provided by hardware accelerator (DES, AES, hash coprocessor).

2b. IxCryptoAcc determines the mode of operation and writes the descriptor (CryptoCtxId, mode, data offset, length of data, mbuf pointer) to CryptoAcc Q via QMgr.

3b. NPE awakes on condition, reads the descriptor from AQM, and begins encryption/decryption and/or authentication operations.

4b. Once all the encryption/decryption/authentication operations complete, the NPE places descriptor (status, mbuf pointer) to the CryptoAcc complete Q and returns to pending state.

5b. QMgr receives an interrupt from AQM through polling mechanism. The QMgr dispatches notification to IxCryptoAcc using the registered call-back.

6b. IxCryptoAcc reads the descriptor from AQM via QMgr.

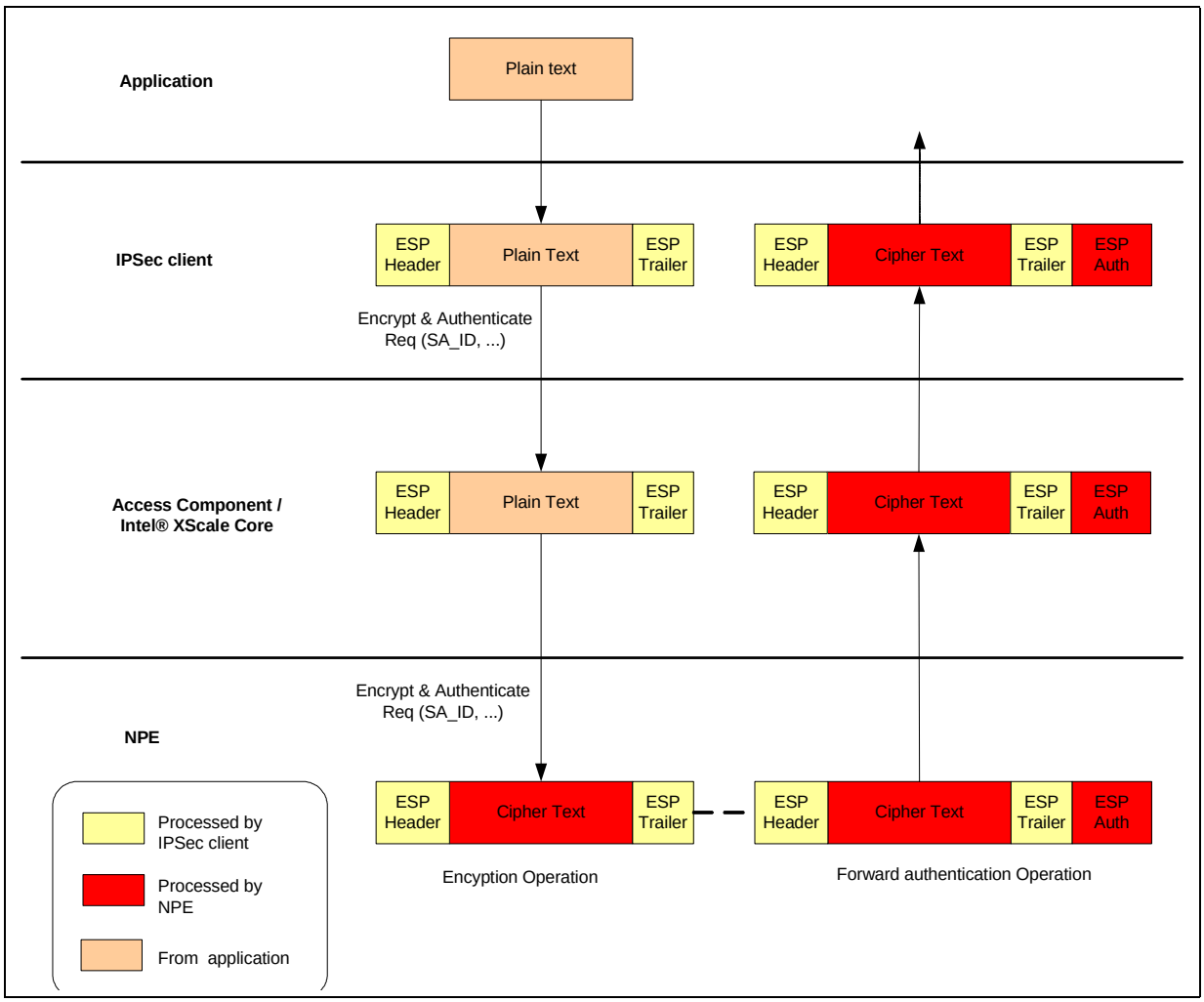
7b. IxCryptoAcc return status to client via callback function.

### 4.12.3.2 Reference ESP Dataflow

Figure 59 shows the example data flow for IP Security environment. Transport mode ESP is used in this example. The IP header is not indicated in the figure.

The IP header is located in front of the ESP header while plain text is the IP payload.

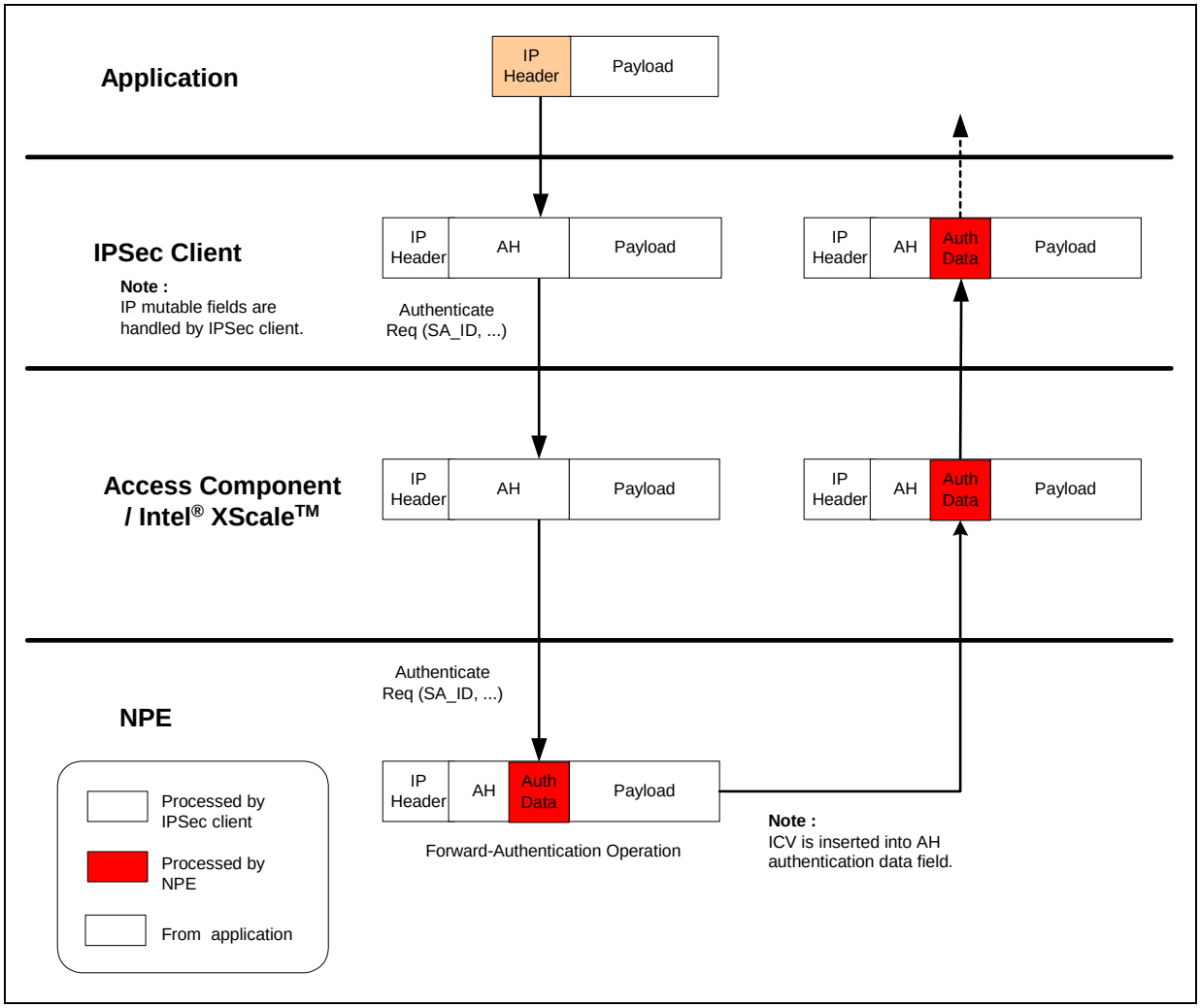
**Figure 59. ESP Data Flow**



### 4.12.3.3 Reference AH Dataflow

Figure 60 shows the example data flow for IP Security environment. Transport mode AH is used in this example. IPSec client handles IP header mutable fields.

**Figure 60. AH Data Flow**



## 4.12.4 Memory Requirements

### 4.12.4.1 Estimated Code Memory

The amount of code memory required by IxCryptoAcc was estimated by using the Wideband Delphi estimation technique and is summarized in Table 45. The size was determined by the following equation:

Size (bytes) = LOC * ANI * BPI

LOC = Non-commented lines of C code.

ANI = Average number ARM assembler instructions per line of C code. Assume a value of six.

BPI = Bytes per instruction. Assume value of four.

**Table 45. Code Size Estimate**

| Code Description | KLOCs | Size (KBytes) |
|---|---|---|
| IxCryptoAcc | 1.75 | 42 |
| **Total** | | 42 |

### 4.12.4.2 Data Memory

This section provides the details on the estimated amount of data memory required by IxCryptoAcc for it to operate under peak call-traffic load. The data memory is reserved statically and is identified as the Cryptographic Context Database (CCD).

The Security Hardware Accelerator Intel XScale core Access component allocates its own memory for the SA Database. The client, however, needs to allocate mbuf for the data and chaining variables to the Security Hardware Accelerator.

Table 46 summarized the size of each entry in CCD required, per tunnel, by the Security Hardware Accelerator component. The Security Hardware Accelerator supports 10,000 of the 140-Byte SA tunnels. Hence, 10,000 x 140 Bytes = 1,400 Kbytes of memory required for the CCD.

The breakdown of the size of each tunnel, in bytes, is shown in Table 46.

**Table 46. SA Tunnel (Cryptographic Context) Memory Requirements**

| Component | Cfg | Chaining Variable | Keys | Callback Ptr | Valid Bit | SPI | Ctr | NPE Ptr | IV Len | Block Len & Digest Len | NPE Op Mode | Rsvd | Sub-total (Byte) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Intel XScale® Core Access Layer** | 0 | 0 | 0 | 8 | 4 | 4 | 8 | 4 | 4 | 4 | 4 | 0 | 40 |
| **NPE** | 8 | 40 | 32 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 16 | 80 |
| **Total** | 8 | 40 | 32 | 8 | 4 | 4 | 8 | 4 | 4 | 4 | 4 | 8 | **140** |

## 4.12.5 Performance Estimation

In computing the performance estimates for the Security Hardware Accelerator firmware, the following assumptions were made:

- All of the NPE is available only for IPSec-related processing. If Ethernet processing is required on the NPE, the IPSec performance will be de-rated:
  - — If filtering/learning is enabled — By roughly 0.8% per 1 Mbit of bidirectional, Ethernet throughput
  - — If filtering/learning is disabled — By roughly 0.5% per 1 Mbit of bidirectional, Ethernet throughput
- AHB contention is at reasonable levels.
- Mbuf chaining is minimal.
- DES-CBC encryption algorithm is used.

  The performance of AES will always be better than that of DES and the ECB and CTR cipher modes will always have performance at least equal to that of CBC.
- HMAC-MD5 authentication algorithm is used.

  The performance of HMAC-SHA-1 will always be better than that of HMAC-MD5.
- Forward processing is requested.

  Reverse processing will have similar performance.

**Table 47. Security Hardware Accelerator Throughput Estimates (Mbps)**

| Processing | | Datagram Size (Bytes) | | |
|---|---|---|---|---|
| | | 64 | 256 | 1,024 |
| 3DES only | Typical[1] | 78.5 | 81.3 | 82.0 |
| | **Best** | **104.6** | **108.3** | **109.3** |
| MD5/SHA-1 only | Typical[1] | 106.6 | 171.7 | 202.6 |
| | **Best** | **142.2** | **228.9** | **270.1** |
| 3DES+MD5/SHA-1 | Typical[1] | 72.9 | 77.0 | 78.1 |
| | **Best** | **97.1** | **102.6** | **104.1** |

**NOTES:**
1. Typical case performance is de-rated from the best case (by 25%) to account for typical, less-than-optimal context interleaving and typical levels of resource (AHB bandwidth and AHB coprocessor utilization) contention.
2. The performance figures for 3DES and MD5/SHA-1 are highly coupled. Any type of processing (3DES, MD5/SHA-1, or 3DES+MD5/SHA-1) counts against the maximum throughput for any other type of processing.

## 4.12.6 Overview of NPE Support for Security

### 4.12.6.1 NPE Firmware Versioning

The security support is fully enabled in the Intel® IXP42X Product Line Processors and is supported only in NPE C. There are variations in the firmware support.

Table 48 shows how the Ethernet and Cryptographic NPE firmware version IDs are mapped into a single, unified NPE C firmware build version ID. While the Ethernet and Cryptographic use a bit-mapped approach for defining functionality, a simple sequential numbering scheme is used for the functionality byte of the NPE C version ID string.

**Table 48. Combined NPE C Version Mappings**

| Ethernet Firmware Version ID | CryptoGraphic Firmware Version ID | Combined NPE C Version ID | Description |
|---|---|---|---|
| 17.00.00 | None | 02.00.01.00 | • Full Ethernet build minus Fast Path<br>• No cryptographic support |
| 1F.00.00 | None | 02.01.01.00 | • Full Ethernet build, including FastPath<br>• No cryptographic support |
| None | 00.00.00 | 02.02.01.00 | • No Ethernet support<br>• Full cryptographic build minus AES support. |
| None | 01.00.00 | 02.03.01.00 | • No Ethernet support<br>• Full cryptographic build, including AES support |
| 17.00.00 | 00.00.00 | 02.04.01.00 | • Full Ethernet build minus FastPath<br>• Full cryptographic build minus AES support |
| 17.00.00 | 01.00.00 | 02.05.01.00 | • Full Ethernet build minus FastPath<br>• Full cryptographic build, including AES support |
| 1F.00.00 | 00.00.00 | 02.06.01.00 | • Full Ethernet build, including FastPath<br>• Full cryptographic build minus AES support |
| 1F.00.00 | 01.00.00 | 02.07.01.00 | • Full Ethernet build, including FastPath<br>• Full cryptographic build, including AES support |

### 4.12.6.2 Supported Encryption and Authentication Algorithms

The NPE supports three different ciphering algorithms

- Data Encryption Standard (DES)
- Triple DES
- Advanced Encryption Standard (AES)

Table 49 summarizes the supported cipher algorithms and the key sizes. The actual key size in DES and 3DES is less because every byte has one parity bit. The parity bit is not used in the encryption process.

**Table 49. Supported Encryption Algorithms**

| Cipher Algorithm | Key Sizes (Bits) | Parity Bit (Bits) | Actual Key Size (Bits) | Plaintext / Ciphertext Block Size (Bits) |
|---|---|---|---|---|
| DES | 64 | 8 | 56 | 64 |
| 3DES | 192 | 24 | 168 | 64 |
| AES | 128<br>192<br>256 | NA | 128<br>192<br>256 | 128 |

The order expected by the Security Hardware Accelerator is in the network byte order (big endian). It is the responsibility of the client to ensure order. For 3DES, the order the keys are passed in should be Key 1, Key 2, and Key 3.

Table 50 summarizes the authentication algorithms supported by the Security Hardware Accelerator. Basically, the Hardware Accelerator supports two authentication algorithms recommended by RFC 2402 and RFC 2406. The authentication algorithm supported is HMAC-SHA1 (RFC 2404) and HMAC-MD5 (RFC 2403).

**Table 50. Supported Authentication Algorithms**

| Authentication Algorithm Supported | Data Block Size (Bits) | Key Size (Bits) |
|---|---|---|
| HMAC-SHA1 | 512 | 160 |
| HMAC-MD5 | 512 | 128 |

### 4.12.6.3 Cipher Modes

There are three cipher modes supported by the NPE:

- Electronic code book (ECB)
- Cipher block chaining (CBC)
- Counter mode (CTR)

### 4.12.6.4 Electronic Code Book (ECB)

The ECB mode for encryption and decryption is supported for DES, Triple DES and AES. ECB is a direct application of the DES algorithm to encrypt and decrypt data.

When using the DES in ECB mode and any particular key, each input is mapped onto a unique output in encryption and this output is mapped back onto the input in decryption. The DES is an iterative, block, product cipher system (i.e., encryption algorithm). A product cipher system mixes transposition and substitution operations in an alternating manner.

### 4.12.6.5 Cipher Block Chaining (CBC)

The CBC mode for encryption and decryption is supported for DES, Triple DES and AES. It requires initialization vector (IV) of size 64-bit for DES and 128-bit for AES initialization vector (IV).

### 4.12.6.6 Counter Mode (CTR)

The counter mode (CTR) is only applicable for AES. The counter block consists of the SPI (the 32-bit value used to distinguish among different SAs terminating at the same destination and using the same IPSec protocol), IV, and a counter that is incremented per input block of plain text. The same AES key is used for the entire encryption process.

### 4.12.6.7 Detailed Cryptographic NPE Data Flow

Communication between the Security Hardware Accelerator (NPE C) and the Intel XScale core processor is facilitated by two mechanisms:

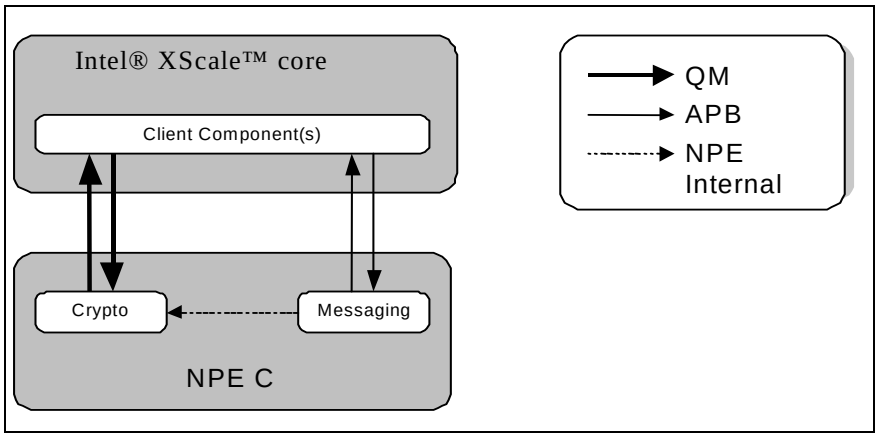
- Queue Manager — External to the NPE
- APB— The NPE accesses it via its internal Debug and Execution Control block

Figure 61 shows a high-level representation of how the NPE-based cryptographic firmware component utilizes these two interfaces to communicate with both the Intel XScale core client software and the firmware.

The primary means of communication between the cryptographic firmware and the ixCryptoAcc component is the Queue Manager. It is used to communicate the existence and location of data awaiting cryptographic service, including cryptographic information (such as keys and modes of encryption/authentication operation), in external SDRAM.

The APB interface is not used directly by the NPE firmware. It is required only to initialize and halt the NPE.

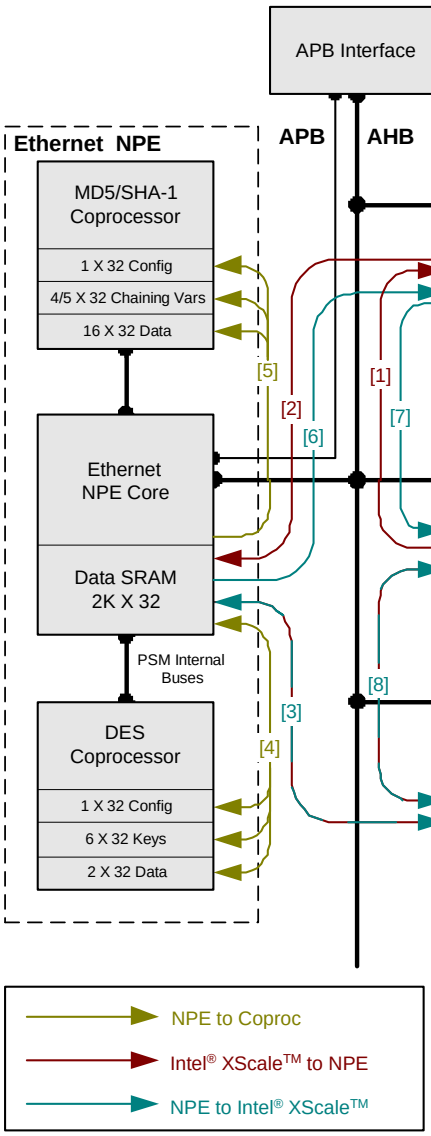
**Figure 61. NPE IxCryptoAcc Component Interface**




The NPE firmware does not participate in the negotiation of service support. Instead, the Intel XScale core client software is responsible for determining and managing the services offered by the NPE firmware, based on the version number of the NPE firmware that has been downloaded.

Figure 62 shows the detailed interaction between the Intel XScale core and the NPE.

## Figure 62. Detailed NPE-to-Intel XScale® Core Cryptographic Data Flow




1. Prior to initiating a cryptographic request, the Intel XScale core creates a valid cryptographic descriptor in SDRAM, ensuring, in particular, that it references a valid cryptographic parameters structure. [8]
2. The Intel XScale core enqueues a pointer to the cryptographic descriptor onto the CryptoReq Queue. [1]
3. The NPE becomes aware of the new request by monitoring the empty flag for CryptoReq and reads the cryptographic descriptor pointer. [2]
4. The NPE reads the cryptographic descriptor from SDRAM. [3]
5. The NPE reads the referenced cryptographic parameters structure from SDRAM. [3]
6. The NPE uses the information — contained within the cryptographic descriptor and cryptographic parameters structure — to configure and initialize the DES (AES) and /or MD5/SHA-1 coprocessor. [4,5]
7. The NPE reads data from the source mbuf into 64-byte internal blocks, following mbuf chains as necessary. Only that portion of the data actually required for cryptographic processing is read. [3]

8a. If DES/AES processing is required, the NPE runs the data through the DES/AES coprocessor, one block at a time, performing any additional operations as required by the block cipher mode. Encrypted data is stored back to PSM data memory. [4]

8b. If MD5/SHA-1 processing is required, the NPE runs the data through the MD5/SHA-1 coprocessor, 64-bytes at a time.

**Note:** If the requested operation is Auth-Ver-ICV/ Decrypt, steps 8A and 8B are reversed in order. [7]

9. If HMAC (or combined crypt/HMAC) processing is requested — after all of the authentication data has been run through the MD5.SHA-1 coprocessor the first time — the MD5/SHA-1 coprocessor is re-initialized with the secondary chaining variables and the secondary hashing operation is performed. The resulting ICV is stored to PSM data memory. [3]
10. If encryption processing is requested, the NPE writes the encrypted data to the destination mbuf, following mbuf chains as necessary (only for “in-place”-type operations). [3]

11a. If an HMAC-Gen-ICV operation is requested, the ICV is written to SDRAM at the ICV address. [3]

11b. If an HMAC-Ver-ICV operation is requested, the PSM reads the ICV — from the ICV address in SDRAM — and compares it to the computed ICV. The result of this comparison is returned in the CryptoDone Queue entry. (See below.) [3]

12. After all cryptographic processing — associated with a particular request — is completed, a pointer to the cryptographic descriptor is enqueued to the CryptoDone Queue. The queue entry also may contain a bit to indicate the success of an HMAC verification operation. [6]
13. The Intel XScale core is alerted that the task is complete when the CryptoDone Queue empty flag goes low. The Intel XScale core reads the CryptoDone Queue entry and continues to process the returned data and status information. [7]

### 4.12.7 Security Hardware Accelerator Demonstration

The example code supplied with the IxCryptoAcc component provides an example of the usage of encryption/decryption and authentication functionality. It can also act as a stub to proof the functionality of IxCryptoAcc component and the co-processors.

***Note:*** This example is not an IPSec application.

## 4.13 ixDMAAcc

### 4.13.1 Overview

The IxDmaAcc provides DMA capability to offload large data transfers between peripherals in the IXP42X Product Line processor memory map from the Intel XScale core. The IxDmaAcc is designed to improve the Intel XScale core system performance by allowing NPE to directly handle large transfers. The Direct Memory Access component (ixDmaAcc) provides the capability to do DMA transfer between peripherals that are attached to AHB busses (North AHB and South AHB busses). It also includes the APB bus, expansion bus and PCI bus.

The ixDmaAcc component the client to access the NPEs' DMA services. The DMA service is selectable to reside in one of the three NPEs during build time. The choice of which NPE runs the DMA feature is controlled by the use of three mutually exclusive build-version macros ("#defines"). The current approach to selecting which NPE the DMA service will reside on is by using a compile switch in the Makefile to define which build version macro will be used. At any NPE build for the DMA, only one of the macros will be left unmasked.

The ixDmaAcc component uses the services of ixQMgr and ixOSServices.

### 4.13.2 Features

The IxDmaAcc component provides these features:

- A DMA Access-layer API.
- Clients' parameters validation.
- Queues DMA requests (FIFO) to the Queue Manager.

### 4.13.3 Assumptions

The DMA service is predicated on the following assumptions:

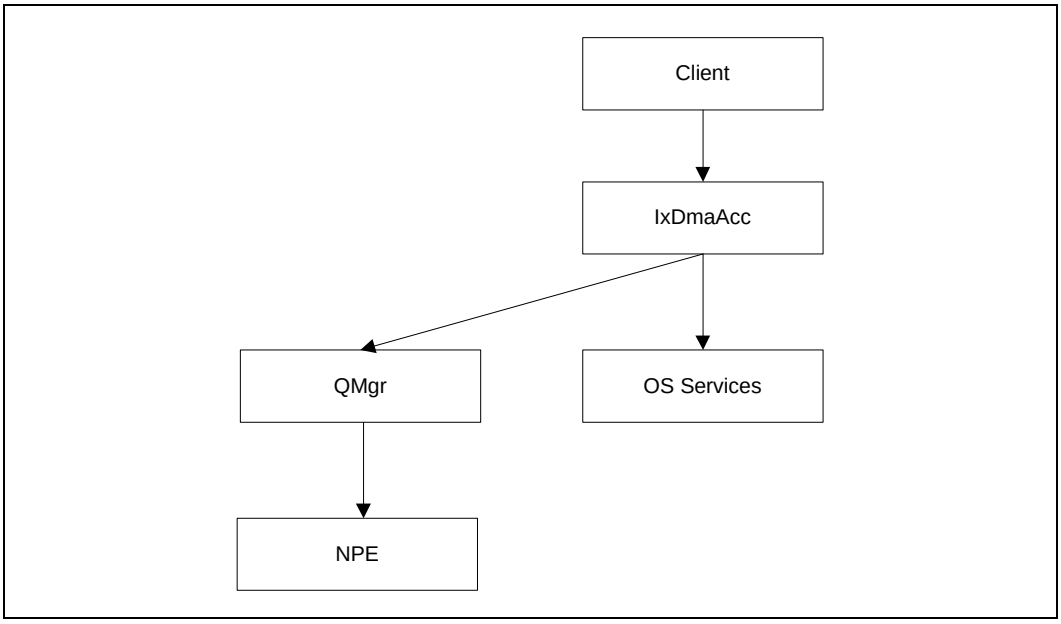
- IxDmaAcc has no knowledge about the IXP42X Product Line processor memory map.The client needs to verify the validity of the source address and destination address of the DMA transfer.
- IxDmaAcc has no knowledge on the devices that involve in the DMA transfer. The client is responsible for ensuring the devices are initialized and configured correctly before request for DMA transfer.

### 4.13.4 Dependencies

Figure 63 shows the functional dependencies of IxDmaAcc component. IxDmaAcc depends on:

- Client component using IxDmaAcc for DMA transfer access.
- ixQMgr component to configure and use the Queue Manager hardware queues.
- ixOS Services component for error handling.
- NPE to perform DMA transfer.

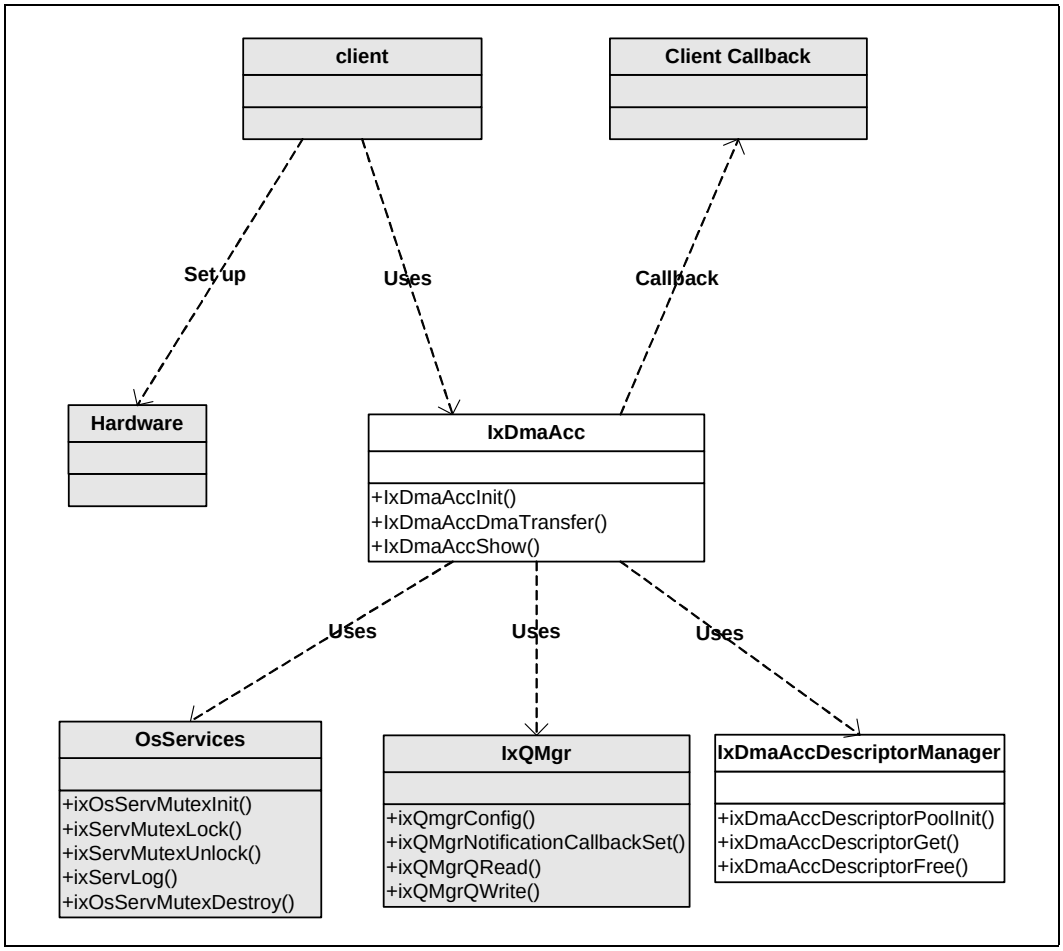
**Figure 63. ixDmaAcc Dependencies**



## 4.13.5 DMA Access Layer API

One of the primary roles of the IxDmaAcc is to provide DMA services to different clients. These DMA services are offered through a set of functions that initialize, transfer, and display the data that needs direct memory access.

**Figure 64. IxDmaAcc Component Overview**




***Note:*** IxDmaAcc components are in white.

Figure 64 shows the dependency between IxDmaAcc component and other external components (in grey). IxDmaAcc depends on:

- Client component using IxDmaAcc for DMA transfer access.
- IxQMgr component for configuring and using the hardware queues to queue the DMA request and to get the ‘DMA done’ request status.
- IxOsServices component for mutual exclusion, error handling, and message log.

The ixDmaAcc component consists of three APIs:

- **PUBLIC IX_STATUS ixDmaAccInit (IxNpeDlNpeId npeId)**

  This function initializes the DMA Access component internals.

- **PUBLIC IxDmaReturnStatus ixDmaAccDmaTransfer (IxDmaAccDmaCompleteCallback callback, UINT32 SourceAddr, UINT32 DestinationAddr, UINT16 TransferLength, IxDmaTransferMode TransferMode, IxDmaAddressingMode AddressingMode, IxDmaTransferWidth TransferWidth)**

  This function performs DMA transfer between devices within the IXP42X memory map.

- **PUBLIC IX_STATUS ixDmaAccShow (void)**

  This function displays internal component information relating to the DMA service (for example, the number of the DMA requests currently pending in the queue).

#### 4.13.5.1 IxDmaAccDescriptorManager

This component provides a private API that is used internally by the ixDmaAcc component. It provides a wrapper around the descriptor-pool-access to simplify management of the pool. This API allocates, initializes, gets, and frees the descriptor entry pool.

The descriptor memory pool is implemented using a circular buffer of descriptor data structures. These data structures hold references to the descriptor memory. The buffer is allocated during initialization. The buffer holds the maximum number of active DMA request the IxDmaAcc supports (16).

This data structure can be accessed by ixDmaAccDescriptorGet function to get an entry from the pool and ixDmaAccDescriptorFree to return the entry back to the pool.

These internal functions include:

- ixDmaAccDescriptorPoolInit(void) — Allocates and initializes the descriptor pool.
- ixDmaAccDescriptorPoolFree(void) — Frees the allocated the descriptor entry pool.
- ixDmaAccDescriptorGet(IxDmaDescriptorPoolEntry *pDescriptor) — Returns pointer to descriptor entry.
- ixDmaAccDescriptorFree(void) — Frees the descriptor entry.

### 4.13.6 Parameters Description

The client needs to specify the source address, destination address, transfer mode, transfer width, addressing mode, and transfer length for each DMA transfers request. The following subsections describe the parameter details.

#### 4.13.6.1 Source Address

Source address is a valid IXP42X Product Line processor memory map address that points to the first word of the data to be read. The client is responsible to check the validity of the source address because the access layer and NPE do not have information on the IXP42X Product Line processor memory map.

### 4.13.6.2 Destination Address

Destination address is a valid IXP42X Product Line processor memory map address that points to the first word of the data to be written. The client is responsible to check the validity of the destination address because the access layer and NPE do not have information on the IXP42X Product Line processor memory map.

### 4.13.6.3 Transfer Mode

Transfer mode describes the type of DMA transfers. There are four types of transfer modes supported:

- **Copy Only** — Moves the data from source to destination.
- **Copy and Clear Source** — Moves the data from source to destination and clears source to zero after the transfer is completed.
- **Copy and Bytes Swapping (endian)** — Moves the data from source to destination. The data written to the destination is bytes swapped. The bytes are swapped within word boundary (for example, 01234567 -> 32107654 where the number indicates the byte location in memory).
- **Copy and Bytes Reverse** — Moves the data from source to destination. The data written to the destination is bytes reversed. The bytes are swapped across word boundary (for example, 01234567 -> 76543210 where the number indicates the byte location in memory).

### 4.13.6.4 Transfer Width

Transfer width describes how the data will be transferred across the AHB buses. There are four transfer widths supported:

- **Burst** — Data may be accessed in a multiple of word per read or write transactions (normally used to access 32-bit devices).
- **8-bit** — Data must be accessed using an individual 8-bit SINGLE transaction (normally used to access 8-bit devices).
- **16-bit** — Data must be accessed using an individual 16-bit SINGLE transaction (normally used to access 16-bit devices).
- **32-bit** — Data must be accessed using an individual 32-bit SINGLE transaction (normally used to access 32-bit devices).

### 4.13.6.5 Addressing Modes

Addressing mode describes the types of source and destination addresses to be accessed. Two addressing modes are supported:

- **Incremental Address** — Address increments after each access, and is normally used to address a contiguous block of memory (i.e. SDRAM).
- **Fixed Address** — Address remains the same for all access, and is normally used to operate on FIFO-like devices (i.e. UART).

### 4.13.6.6 Transfer Length

This is the size of the data to be transferred from the source address to the destination address. Transfer length restrictions are:

- Transfer length of 8-bit devices can be in multiple of byte, half-word, or word.
- Transfer length of 16-bit devices can be in multiple of half-word or word.
- Transfer length of 32-bit devices is in multiple of word.

### 4.13.6.7 Supported Modes

This section summarizes the transfer modes supported by the IxDmaAcc. Some of the supported modes have restrictions. See Section 4.13.9 for details on restrictions.

**Table 51. DMA Modes Supported for Addressing Mode of Incremental Source Address and Incremental Destination Address**

| Increment Source Address | Increment Destination Address | Transfer Mode | | | |
|---|---|---|---|---|---|
| Transfer Width Source | Transfer Width Destination | Copy Only | Copy and Clear | Copy and Bytes Swapping | Copy and Bytes Reverse |
| 8-bit | 8-bit | SUPPORTED | SUPPORTED | SUPPORTED | SUPPORTED |
| 8-bit | 16-bit | SUPPORTED | SUPPORTED | SUPPORTED | SUPPORTED |
| 8-bit | 32-bit | SUPPORTED | SUPPORTED | SUPPORTED | SUPPORTED |
| 8-bit | Burst | SUPPORTED | SUPPORTED | SUPPORTED | SUPPORTED |
| 16-bit | 8-bit | SUPPORTED | SUPPORTED | SUPPORTED | SUPPORTED |
| 16-bit | 16-bit | SUPPORTED | SUPPORTED | SUPPORTED | SUPPORTED |
| 16-bit | 32-bit | SUPPORTED | SUPPORTED | SUPPORTED | SUPPORTED |
| 16-bit | Burst | SUPPORTED | SUPPORTED | SUPPORTED | SUPPORTED |
| 32-bit | 8-bit | SUPPORTED | SUPPORTED | SUPPORTED | SUPPORTED |
| 32-bit | 16-bit | SUPPORTED | SUPPORTED | SUPPORTED | SUPPORTED |
| 32-bit | 32-bit | SUPPORTED | SUPPORTED | SUPPORTED | SUPPORTED |
| 32-bit | Burst | SUPPORTED | SUPPORTED | SUPPORTED | SUPPORTED |
| Burst | 8-bit | SUPPORTED | SUPPORTED | SUPPORTED | SUPPORTED |
| Burst | 16-bit | SUPPORTED | SUPPORTED | SUPPORTED | SUPPORTED |
| Burst | 32-bit | SUPPORTED | SUPPORTED | SUPPORTED | SUPPORTED |
| Burst | Burst | SUPPORTED | SUPPORTED | SUPPORTED | SUPPORTED |

**Table 52. DMA Modes supported for Addressing Mode of Incremental Source Address and Fixed Destination Address.**

| Increment Source Address | Increment Destination Address | Transfer Mode | | | |
|---|---|---|---|---|---|
| Transfer Width Source | Transfer Width Destination | Copy Only | Copy and Clear | Copy and Bytes Swapping | Copy and Bytes Reverse |
| 8-bit | 8-bit | SUPPORTED | SUPPORTED | SUPPORTED | SUPPORTED |
| 8-bit | 16-bit | SUPPORTED | SUPPORTED | SUPPORTED | SUPPORTED |
| 8-bit | 32-bit | SUPPORTED | SUPPORTED | SUPPORTED | SUPPORTED |
| 8-bit | Burst | NOT SUPPORTED | NOT SUPPORTED | NOT SUPPORTED | NOT SUPPORTED |
| 16-bit | 8-bit | SUPPORTED | SUPPORTED | SUPPORTED | SUPPORTED |
| 16-bit | 16-bit | SUPPORTED | SUPPORTED | SUPPORTED | SUPPORTED |
| 16-bit | 32-bit | SUPPORTED | SUPPORTED | SUPPORTED | SUPPORTED |
| 16-bit | Burst | NOT SUPPORTED | NOT SUPPORTED | NOT SUPPORTED | NOT SUPPORTED |
| 32-bit | 8-bit | SUPPORTED | SUPPORTED | SUPPORTED | SUPPORTED |
| 32-bit | 16-bit | SUPPORTED | SUPPORTED | SUPPORTED | SUPPORTED |
| 32-bit | 32-bit | SUPPORTED | SUPPORTED | SUPPORTED | SUPPORTED |
| 32-bit | Burst | NOT SUPPORTED | NOT SUPPORTED | NOT SUPPORTED | NOT SUPPORTED |
| Burst | 8-bit | SUPPORTED | SUPPORTED | SUPPORTED | SUPPORTED |
| Burst | 16-bit | SUPPORTED | SUPPORTED | SUPPORTED | SUPPORTED |
| Burst | 32-bit | SUPPORTED | SUPPORTED | SUPPORTED | SUPPORTED |
| Burst | Burst | NOT SUPPORTED | NOT SUPPORTED | NOT SUPPORTED | NOT SUPPORTED |

**Table 53. DMA Modes supported for Addressing Mode of Fixed Source Address and Incremental Destination Address.**

| Increment Source Address | Increment Destination Address | Transfer Mode | | | |
|---|---|---|---|---|---|
| Transfer Width Source | Transfer Width Destination | Copy Only | Copy and Clear | Copy and Bytes Swapping | Copy and Bytes Reverse |
| 8-bit | 8-bit | SUPPORTED | SUPPORTED | SUPPORTED | SUPPORTED |
| 8-bit | 16-bit | SUPPORTED | SUPPORTED | SUPPORTED | SUPPORTED |
| 8-bit | 32-bit | SUPPORTED | SUPPORTED | SUPPORTED | SUPPORTED |
| 8-bit | Burst | SUPPORTED | SUPPORTED | SUPPORTED | SUPPORTED |
| 16-bit | 8-bit | SUPPORTED | SUPPORTED | SUPPORTED | SUPPORTED |
| 16-bit | 16-bit | SUPPORTED | SUPPORTED | SUPPORTED | SUPPORTED |
| 16-bit | 32-bit | SUPPORTED | SUPPORTED | SUPPORTED | SUPPORTED |
| 16-bit | Burst | SUPPORTED | SUPPORTED | SUPPORTED | SUPPORTED |
| 32-bit | 8-bit | SUPPORTED | SUPPORTED | SUPPORTED | SUPPORTED |
| 32-bit | 16-bit | SUPPORTED | SUPPORTED | SUPPORTED | SUPPORTED |
| 32-bit | 32-bit | SUPPORTED | SUPPORTED | SUPPORTED | SUPPORTED |
| 32-bit | Burst | SUPPORTED | SUPPORTED | SUPPORTED | SUPPORTED |
| Burst | 8-bit | NOT SUPPORTED | NOT SUPPORTED | NOT SUPPORTED | NOT SUPPORTED |
| Burst | 16-bit | NOT SUPPORTED | NOT SUPPORTED | NOT SUPPORTED | NOT SUPPORTED |
| Burst | 32-bit | NOT SUPPORTED | NOT SUPPORTED | NOT SUPPORTED | NOT SUPPORTED |
| Burst | Burst | NOT SUPPORTED | NOT SUPPORTED | NOT SUPPORTED | NOT SUPPORTED |

## 4.13.7 Data Flow

The purpose of the DMA access layer is to transfer DMA configuration information from its clients to the NPEs. It is a control component where the actual DMA data flow is transparent to the IxDmaAcc component.

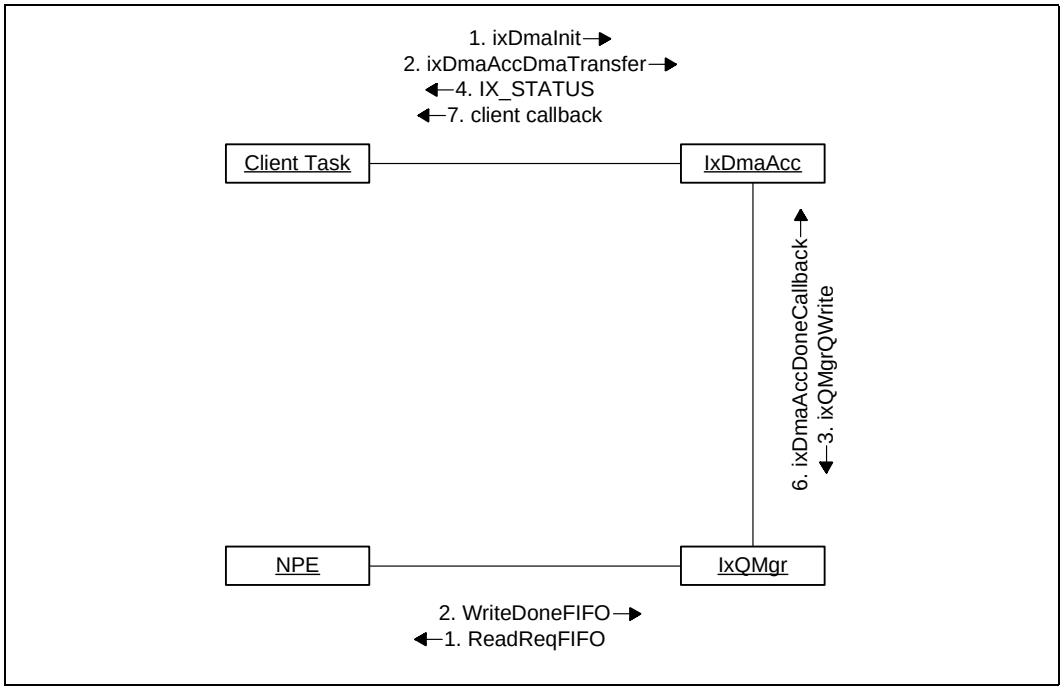
## 4.13.8 Control Flow

For a DMA transaction to start, the client must initialize the DMA access layer, write to the queue manager, and receive a status of the transaction.

The IxDmaAcc component simultaneously supports multiple services. Consequently, a new request may be submitted before the confirmation of a previous DMA request is received from the NPE. The DMA Access layer API, however, assumes that all requests originate from the same Intel XScale core task. The DMA request is queued in the AQM's request queue and waits to be serviced by the DMA NPE. Upon completion of the DMA transfer, the NPE writes a message to the AQM-done queue. The AQM dispatcher then calls the ixDmaAcc callback and the access layer calls the client callback.

Figure 65 shows the overall flow of the DMA transfer operation between the client, the access layer, and the NPE.

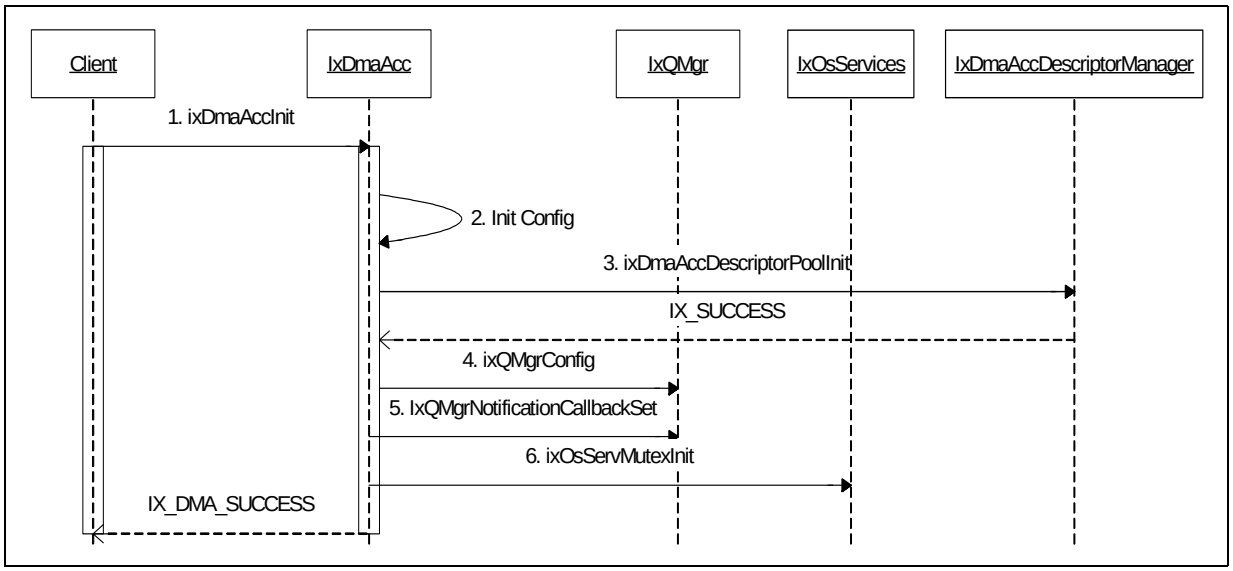
**Figure 65. IxDmaAcc Control Flow**



### 4.13.8.1 DMA Initialization

Figure 66 and the following steps describe the DMA access-layer initialization:

**Figure 66. IxDMAcc Initialization**




1. Client calls ixDmaAccInit to initialize the IxDmaAcc component with an NPE ID as a parameter. The NPE ID indicates which NPE is been used to provide the DMA functionality.
2. ixDmaAccInit checks if ixQmgr and the OSSL components have been initialized.
3. ixDmaAccInit calls ixDmaAccDescriptorPoolInit to allocate and initialize an array of descriptor data structures to store the DMA request and client's callback function.

   (See the ixDmaAccDescriptorManager description.)
4. ixDmaAccInit calls ixQmgrConfig to configure the DMA request queue and the DMA done queue.

   The queue ID depends on which NPE the DMA component will be loaded. The selection of which NPE to run is made during run time by the client code.

   The client also need to initialize AQM (the Queue Manager).
5. ixDmaAccInit calls ixQMgrNotificationCallbackSet to register the callback function for the DMA-done queue.
6. ixDmaAccInit calls ixOsSevices to initialize mutex.

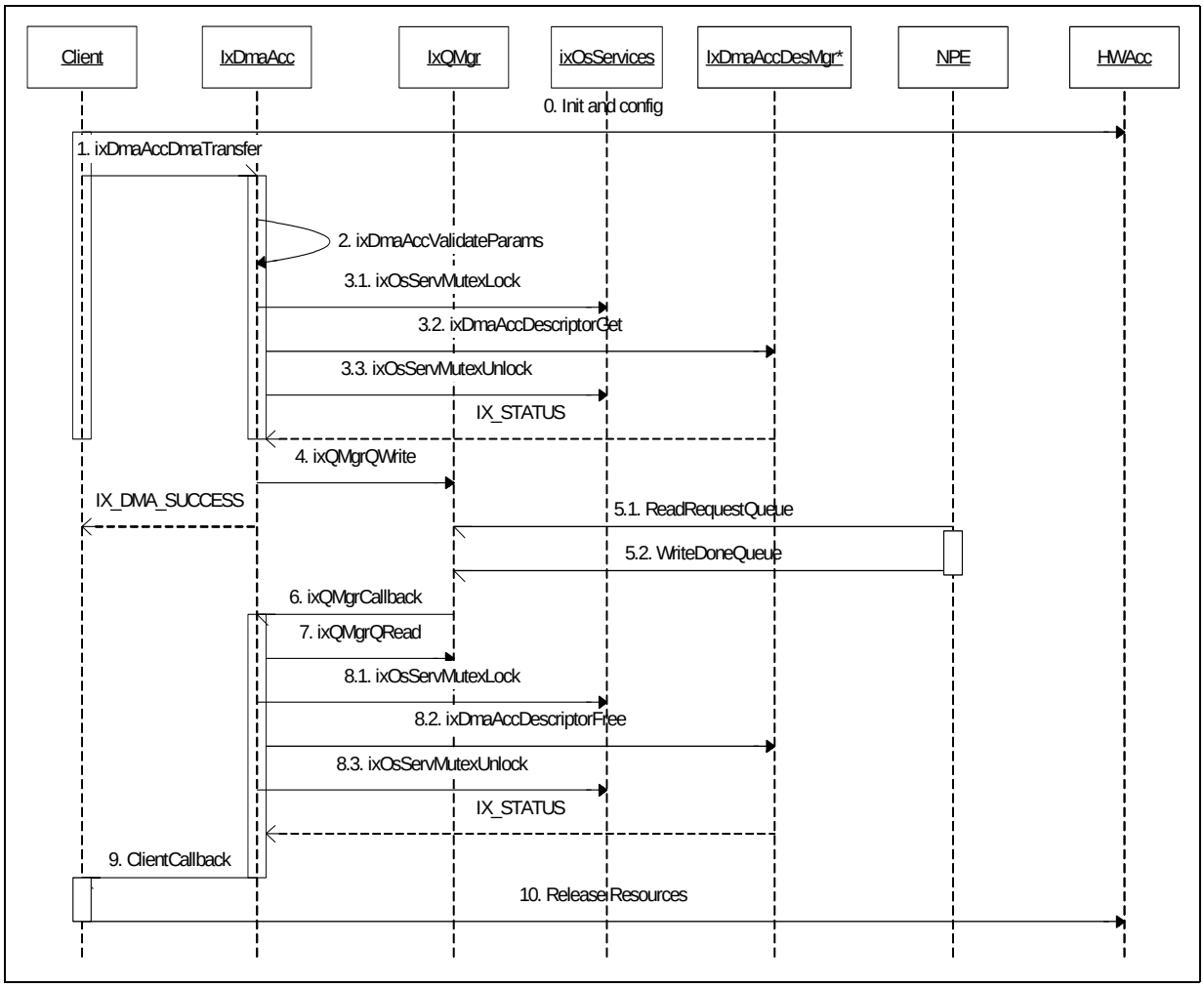
   The mutex ID will be used to access queue descriptor entry pool.

   ixDmaAccInit returns IX_DMA_SUCCESS upon completion of the DMA initialization.

### 4.13.8.2 DMA Configuration and Data Transfer

Figure 67 describes the configuration and DMA data transfer between a client and an NPE.

**Figure 67. DMA Transfer Operation**




0. Client needs to initialize and configure the hardware for the DMA transfer to ensure that the devices are set up properly and ready for DMA transfer.
1. Client requests the DMA transfer by calling ixDmaAccDmaTransfer function.
2. Internally, ixDmaAccDmaTransfer function calls ixDmaAccValidateParams function to validate the client's input parameters.
3. If the client input parameters are valid, the ixDmaAccDmaTransfer function gets a descriptor entry from the descriptor manager.

   The descriptor pool needs to be guarded by mutual exclusion because there are two contexts that access the pool descriptor buffer. The ixDmaAcc component will get the pool entry and the AQM will free the entry pool (via callback).
4. The ixDmaAccDmaTransfer function composes the descriptor — based on the client's parameters — and calls ixQMgrQWrite to queue the descriptor to AQM.

5. ixDmaAccDmaTransfer returns and gets ready to process the new DMA transfer request.
6. The NPE reads the queue manager and does the DMA transfers. Upon completion of the DMA transfer, the NPE writes to AQM's done queue. The AQM dispatcher calls the IxDmaAcc's registered callback function.
7. IxDmaAccCallback calls ixQMgrQRead to read the result and that result is stored in the third descriptor. If the third word of the descriptor is zero, an AHB error is asserted by a peripheral having been accessed.
8. The descriptor pool needs to be guarded by mutual exclusion because there are two contexts that access the pool descriptor buffer (see step 3).
9. IxDmaAccCallback frees the descriptor.

   The descriptor pool needs to be guarded by mutual exclusion (see step 3).
10. IxDmaAccCallback calls client registered callback.
11. Client releases the resources allocated in step 0.

### 4.13.9 Restrictions of the DMA Transfer

The client is responsible for ensuring that the following restrictions are followed when issuing a DMA request:

- The Intel XScale core is operating in the big-endian mode.
- The host devices are operating in big-endian mode. This means that the valid bytes for 8-bit and 16-bit transfer width are in the most significant bytes (MSB). For example, for the 16-bit transfer, the data is 0xAABBXXXX, where X is don't care value.
  - There is a slight difference in the access to the APB memory map region, specifically for UART accessed. A read from an APB target is a 32-bit read from a word-aligned address. In the case of the UART Rx and Tx FIFOs, only the least significant byte (bits 7:0) of each word read/written contains valid data not in the MSB. Therefore, instead of using 0xC8000000 for UART1 and 0xC8001000 for UART2, any DMA request involving the UARTs must instead specify an address of 0xC8000003 for UART1 and 0xC8001003 for UART2 (in both cases the transfer width should be set to 8 bits). APB discards 1:0 bit address when decode the AHB addresses; therefore, valid data is read in MSB.
- Fixed address does not support burst mode. Fixed address associates with a single transaction. This means that the fixed address will either have a transfer width of 8-bit, 16-bit, or 32-bit single transaction. Fixed address (either fixed source address or fixed destination address) does not support burst transaction because burst transaction will always increment the address throughout the transaction. In addition, the AHB co-processor does not have an instruction set to do burst transfer on fixed address mode.
- Fixed source address with copy and clear transfer mode, the source is clear only once after the transfer is completed.
- In the fixed source address mode, the client application is responsible to ensure that the data is available for transfer. For example, using FIFO with entry size 32-bit as a fixed address mode with the transfer length of 8 bytes, the client must ensure that the data is available before the DMA transfer is performed.
- Due to the asymmetric nature of the expansion bus, the incrementing source address and a "burst" transfer width will not support the "copy & clear" mode for expansion bus sources. The reason that this mode is not supported is that expansion bus targets can be read in burst mode, but they cannot be written in burst mode.

- If DMA transfer mode of “Byte-Swapped” or “Byte Reverse” is selected and if the Source DMA Addressing mode is “Incremental,” the DMA Source address must be “word-aligned” and the DMA transfer length would be a multiple of words. The reason is that endianness swapping will always be done on the word boundary.
- Burst mode is not supported for DMA targets at AHB South Bus. This is due to hardware restriction. Therefore, all DMA transactions originated or designated the south AHB bus peripherals is carried out in SINGLE transaction mode.
- The DMA access component is fully tested on SDRAM and Flash devices only. Even though the IxDmaAcc is designed to provide capability to offload large data transfers between peripherals in the IXP42X Product Line processor memory map.
- These DMA restrictions apply when a Flash is a destination device:
  - Burst mode is not supported and only supports SINGLE mode.
  - Incremental source to fixed destination DMA addressing mode is not supported.
  - DMA transfer width for the destination must match the Flash device data bus width.
  - Byte-reverse DMA mode with fixed source to incremental destination is not supported with the Flash write buffer mode.
- These DMA restrictions apply when a Flash is a source device:
  - Copy and clear DMA mode is not supported
  - DMA transfer width for the source must match the Flash device data bus width.

### 4.13.10 Error Handling

IxDmaAcc returns an error type to the user when the client is expected to handle the error. Internal errors will be reported using standard IXP42X Product Line processor error-reporting techniques, such as the OS services error-reporting mechanism.

# Operating System Abstraction Layers 5

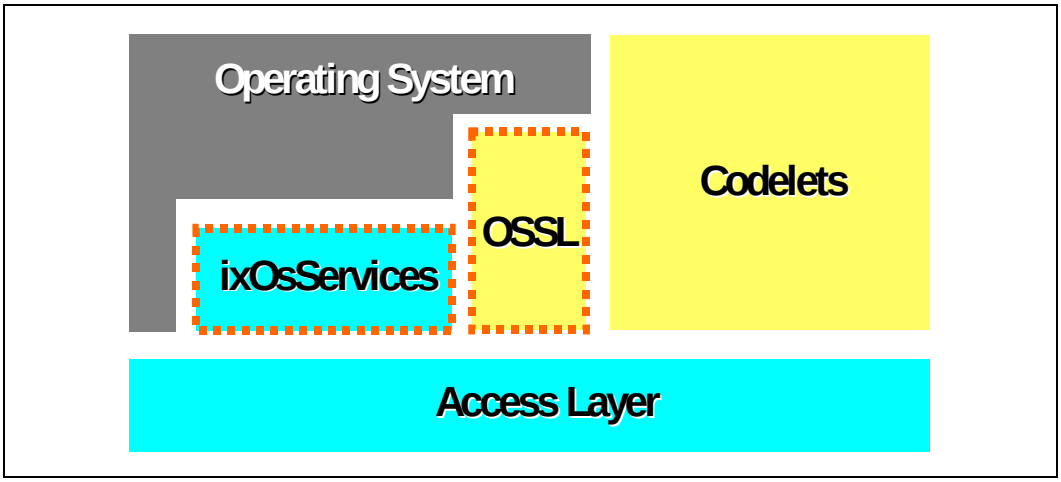
## 5.1 Overview

There are two operating-system abstraction layers provided as part of the IXP400 software v1.2.2 architecture.

The ixOsServices Layer provides a very thin set of abstracted operating-system (OS) services. IXP400 software access-layer components abstract their OS dependencies to this module. The set of functions, defined by the ixOsServices, represent the minimum effort required to port the IXP400 software Access Layer components to different Operating Systems

The Operating System Service Layer defines a more fully featured portable interface for Operating System Services.

**Figure 68. Intel® IXP400 Software Operating System Services Layers**




***Note:*** The IXP400 software codelets also use these layers, but are not limited to the range of services provided. This enables the codelets to rely directly on the OS for more advanced OS features.

## 5.2 ixOsServices

The OS-services layer provides a minimum set of abstracted OS services. The IXP400 software access-layer components abstract their OS dependency to this module.

The `ixoservices.h` file contains the API to the functions that are somewhat OS-dependant and would require porting to a particular OS. A primary focus of the IXP400 software access-layer component development was to make the APIs as OS-independent as possible.

### 5.2.1 Mutual Exclusion Service

The mutual exclusion service provides interrupt binding and locking mechanisms, mutex locks, and fast mutexes.

The following services are provided:

- **Interrupt locking mechanisms —** Depending on the implementation, the OS may allow for fine-grained locking of interrupts of a specific level or a system-wide disable/enable API.

  Use of locking mechanisms that have a system-wide affect must be carefully scrutinized. These mechanisms must be used for a minimum duration and no function calls allowed between lock/unlock events.

- **Mutex lock —** This relies on underlying operating-system services. A mutex will ensure that only one task/process has access to a piece of code, data, or resource.

  A mutex lock can suspend a process, if there is task contention that lock and will automatically resume the process when the contention disappears. Mutex locks form part of the operating system and are expensive in terms of CPU cycles.

- **"Fast" semaphore —** This will use the Intel® StrongARM* instruction set to provide single access to a piece of code, data, or resource.

  The underlying operating system is not aware of this lock and no task suspension or priority access to the lock is given. If there is contention for the lock, the onus is on the caller to decide what to do — that is, retry to obtain the lock or abort.

  The fast semaphore lock is very simple and very efficient, if lock contention is very infrequent, this type of lock should be used.

### 5.2.2 Trace Service

The trace service is a simple debugging mechanism, with compile-time debug trace level that depends on existent OS-logging features such as kprintf in Linux* and logMsg in VxWorks*.

### 5.2.3 Timer Service

The timer services include:

- Non-preemptive, timed delays, busy loop with microsecond granularity
- Timed delays, OS-dependant yielding with millisecond granularity
- Time-stamp measurements, using Intel XScale core clock granularity

### 5.2.4 ixOsServices Functions

The ixOsServices layer implements 17 functions. Basic ixOsServices provided are shown in Table 54.

**Table 54. Intel® IXP400 Software v1.2.2 ixOsServices Function Descriptions**

| ixOsServices | | |
|---|---|---|
| | **Function** | **Description** |
| **Mutual Exclusion** | ixOsServIntBind | Binds a routine to a hardware interrupt. |
| | ixOsServIntUnBind | Unbinds a routine from a hardware interrupt. |
| | ixOsServIntLock | Disables IRQs. |
| | ixOsServIntUnlock | Enables IRQs. |
| | ixOsServIntLevelSet | Sets the interrupt mask. |
| | ixOsServMutexInit | Initializes a mutual-exclusion device. |
| | ixOsServMutexLock | Locks a mutual-exclusion device. |
| | ixOsServMutexUnlock | Unlocks a mutual-exclusion device. |
| | ixOsServMutexDestroy | Destroys a mutual-exclusion object. |
| | ixOsServFastMutexInit | Initializes a fast mutex. |
| | ixOsServFastMutexTryLock | Attempts to lock the fast-mutex object. |
| | ixOsServFastMutexUnlock | Unlocks the fast-mutex object. |
| **Logging** | ixOsServLog | Logs the specified message, using the logging task. |
| | ixOsServLogLevelSet | Sets the logging level. |
| **Timer** | ixOsServSleep | Blocks the calling task, using a timed busy loop. |
| | ixOsServTaskSleep | Blocks the calling task, using an OS-dependent, preemptive timed delay. |
| | ixOsServTimestampGet | Retrieves the system time stamp. |

## 5.3 OSSL

The Operating System Services Layer (OSSL) defines a portable interface for operating-system services. OSSL abstraction layer is a set of functions that provide OS-independent APIs and data types.

These primitives are implemented as different sets of APIs. These APIs invoke corresponding OS functions. The OSSL provides an abstraction layer for the following types of OS services:

- Thread management
- Semaphores
- Mutual exclusion
- Timers
- Memory management
- Message logging

### 5.3.1 Thread Management

It is often essential to organize applications into independent (though cooperating) tasks. Each of these tasks, while executing, is called a thread. Threads have immediate, shared access to most resources. Threads also maintain enough separate contexts to maintain individual control.

The ‘ix_ossl_thread_create(..)’ function call is used create an OSSL thread. A user-provided, thread-entry-point function is passed as an argument to the thread creation function.

### 5.3.2 Semaphores

Semaphores provide inter-thread synchronization mechanisms. They coordinate a thread’s execution with other threads.

The OSSL semaphores are binary and have “available” or “unavailable” states. A thread signals the occurrence of an event by setting the semaphore to an “available” state.

Another thread waits for the semaphore to become “available.” The waiting thread is blocked until the event occurs and the semaphore becomes “available” or for a specified time-out period, whichever occurs earlier.

Currently, counting semaphores are not supported.

### 5.3.3 Mutual Exclusion

Mutual exclusion (mutex) is used to guard a critical section. A critical section is a part of the program that needs exclusive access to shared resources such as a buffer or an I/O device.

A critical section should be protected by a mutex for the correct behavior of the code executing in the critical section. Each critical section should be protected by a mutex. Critical sections are identified during development and a mutex should be allocated using `ix_ossl_mutex_init`.

When a thread wants to access a critical section, it must first lock the mutex for that critical section. When the thread is finished with the critical section, it unlocks the mutex, allowing another thread to use the critical section.

### 5.3.4 Semaphores versus Mutexes

Semaphores are used for thread coordination and synchronization. For example, a receiving thread can set a semaphore “available” as soon it receives packets from the port. A processing thread will wait on this semaphore and — when a semaphore becomes “available” — it will process the packet buffer.

This kind of thread coordination is used to avoid busy wait in a program. A busy wait is a waste of CPU resources. A busy wait is avoided by having the thread wait on a semaphore, which puts the thread in a pending queue — a more efficient way of using system resources.

Mutexes are used to provide mutually exclusive access to critical sections of the program code. For example, a thread that wants to access a shared packet buffer should lock the mutex for the buffer before it starts using it. After the thread completes the processing, it unlocks the mutex.

Each mutex lock must be followed by a subsequent mutex unlock by the same thread. Otherwise, dead locks will occur and program will enter into an undesirable state.

A mutex is owned by the thread that locked it, a semaphore is not similarly owned. That means a semaphore can be unlocked by another thread, unlike a mutex. A mutex has to be unlocked by the same thread that locked it.

### 5.3.5 Timers

OSSL timers provide primitives to get system time and cause thread delays. System time is provided in nanosecond-level resolution.

### 5.3.6 Memory Management

Memory-management functions provide dynamic memory allocation, de-allocation, memory-copy, and memory-set operations. There is no garbage collection done in OSSL and modules in IXA SDK should deallocate the dynamic memory using `ix_ossl_free`.

### 5.3.7 Message Logging

Message logging is used to log and/or display the error messages. The message logging is initialized using the `ix_ossl_message_log_init` function. This function should be called before any call to `ix_ossl_message_log()`.

For each OS, the messages will be logged into an implementation-dependent stream. Further details will be provided on where the messages will be logged. `ix_ossl_message_log` is used to log error messages.

## 5.3.8 OSSL Functions

Each OSSL API function will use the corresponding OS-provided service. The ixOSSL layer implements 23 functions. The basic OSSL services provided are shown in Table 55.

**Table 55. Intel® IXP400 Software v1.2.2 ixOSSL Function Descriptions**

| ixOSSL | | |
|---|---|---|
| | **Function** | **Description** |
| **Thread** | ix_ossl_thread_create | Creates a cancellable thread. |
| | ix_ossl_thread_get_id | Returns the ID of a calling thread. |
| | ix_ossl_thread_exit | Causes the calling thread to exit. |
| | ix_ossl_thread_kill | Kills the running thread. |
| | ix_ossl_thread_set_priority | Sets the priority of a thread. |
| **Semaphores** | ix_ossl_sem_init | Initializes a new semaphore. |
| | ix_ossl_sem_fini | Frees and deletes semaphore. |
| | ix_ossl_sem_take | Takes control, if semaphore full. |
| | ix_ossl_sem_give | Unblocks next available thread in pend queue. |
| | ix_ossl_sem_flush | Unblocks all pending threads. |
| **Mutex** | ix_ossl_mutex_init | Initializes a new mutex. |
| | ix_ossl_mutex_fini | Frees and deletes a mutex. |
| | ix_ossl_mutex_lock | Locks a mutex. |
| | ix_ossl_mutex_unlock | Unlocks a mutex. |
| **Timer** | ix_ossl_sleep | Causes the calling thread to sleep for a specified time in milliseconds. |
| | ix_ossl_sleep_tick | Causes the calling thread to sleep for a specified time in OS ticks. |
| | ix_ossl_time_get | Returns the current value of a timer. |
| **Memory Mgmt** | ix_ossl_malloc | Allocates a memory block. |
| | ix_ossl_free | Frees a memory block. |
| | ix_ossl_memcpy | Copies memory bytes between buffers. |
| | ix_ossl_memset | Sets buffers to a specified character. |
| **Log** | ix_ossl_message_log_init | Initializes the error message logging. |
| | ix_ossl_message_log | Logs a specified message. |

## 5.4 Mapping Software Components to Operating System Services

**Table 56. Access Layer Component OS Service Dependencies (Sheet 1 of 2)**

| | | Access Layer Components | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | ixNpeMh | ixQmgr | ixAtmdAcc | ixHssAcc | ixEthAcc | ixEthDB | ixFPathAcc | ixCryptoAcc | ixDmaAcc | ixUsbAcc | ixUartAcc |
| ixOsServices | ixOsServIntBind | X | | | | | | | | | X | |
| | ixOsServIntUnBind | | | | | | | | | | | |
| | ixOsServIntLock | | X | | X | X | X | | X | X | X | |
| | ixOsServIntUnLock | | X | | X | X | X | | X | X | X | |
| | ixOsServIntLevelSet | | | | | | | | | | | |
| | ixOsServMutexInit | X | X | X | X | X | X | X | | X | | |
| | ixOsServMutexLock | X | X | X | X | X | X | X | | | | |
| | ixOsServMutexUnlock | X | X | X | X | X | X | X | | | | |
| | ixOsServMutexDestroy | | | | | | | | | | | |
| | ixOsServFastMutexInit | | X | | | | X | | | | | |
| | ixOsServFastMutexTryLock | | X | | | | X | | | | | |
| | ixOsServFastMutexUnlock | | X | | | | X | | | | | |
| | ixOsServLog | X | X | X | X | X | X | X | X | X | | |
| | ixOsServLogLevelSet | | | | X | | | X | | | | |
| | ixOsServSleep | X | | | | | | X | | | X | |
| | ixOsServTaskSleep | | | | | X | X | X | X | | | |
| | ixOsServTimestampGet | | | | | | | | | | | |

### Table 56. Access Layer Component OS Service Dependencies (Sheet 2 of 2)

| | | Access Layer Components | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | ixNpeMh | ixQmgr | ixAtmdAcc | ixHssAcc | ixEthAcc | ixEthDB | ixFPathAcc | ixCryptoAcc | ixDmaAcc | ixUsbAcc | ixUartAcc |
| OSSL | ix_ossl_thread_create | | | | | | X | | | | | |
| | ix_ossl_thread_get_id | | | | | | | | | | | |
| | ix_ossl_thread_exit | | | | | | | | | | | |
| | ix_ossl_thread_kill | | | | | | | | | | | |
| | ix_ossl_thread_set_priority | | | | | | | | | | | |
| | ix_ossl_sem_init | | | | | | X | | | | | |
| | ix_ossl_sem_fini | | | | | | | | | | | |
| | ix_ossl_sem_take | | | | | | X | | | | | |
| | ix_ossl_sem_give | | | | | | X | | | | | |
| | ix_ossl_sem_flush | | | | | | | | | | | |
| | ix_ossl_mutex_init | | | | | | | | | | | |
| | ix_ossl_mutex_fini | | | | | | | | | | | |
| | ix_ossl_mutex_lock | | | | | | | | | | | |
| | ix_ossl_mutex_unlock | | | | | | | | | | | |
| | ix_ossl_sleep | | | | | | | | | | | |
| | ix_ossl_sleep_tick | | | | | | | | | | | |
| | ix_ossl_time_get | | | | | | | | | | | |
| | ix_ossl_malloc | | | | | | | | | | | |
| | ix_ossl_free | | | | | | | | | | | |
| | ix_ossl_memcpy | | | | | | | | | | | |
| | ix_ossl_memset | | | | | | | | | | | |
| | ix_ossl_message_log_init | | | | | | | | | | | |
| | ix_ossl_message_log | | | | | | | | | | | |

***Note:*** The ixEthDB is the only Access Layer component that requires OSSL services.

# System Configuration 6

This chapter describes the Intel® IXP42X Product Line Processors PCI-configuration APIs, and the ADSL driver implementation on the IXDP425 Development Platform.

## 6.1 PCI Configuration

The IXP42X Product Line processor has a PCI core — 32-bit at 33 MHz or 66 MHz. The PCI core can be either a host (hosting up to four PCI devices) or non-host (plugged into a PCI slot). There is a strapping option available via the Expansion Bus at reset to select either a host or non-host.

This section documents the PCI APIs that are available as part of the Wind River Systems* Board Support Package for the Intel® IXDP425 Network Processor Development Platform.

**Figure 69. PCI Controller Block Diagram**

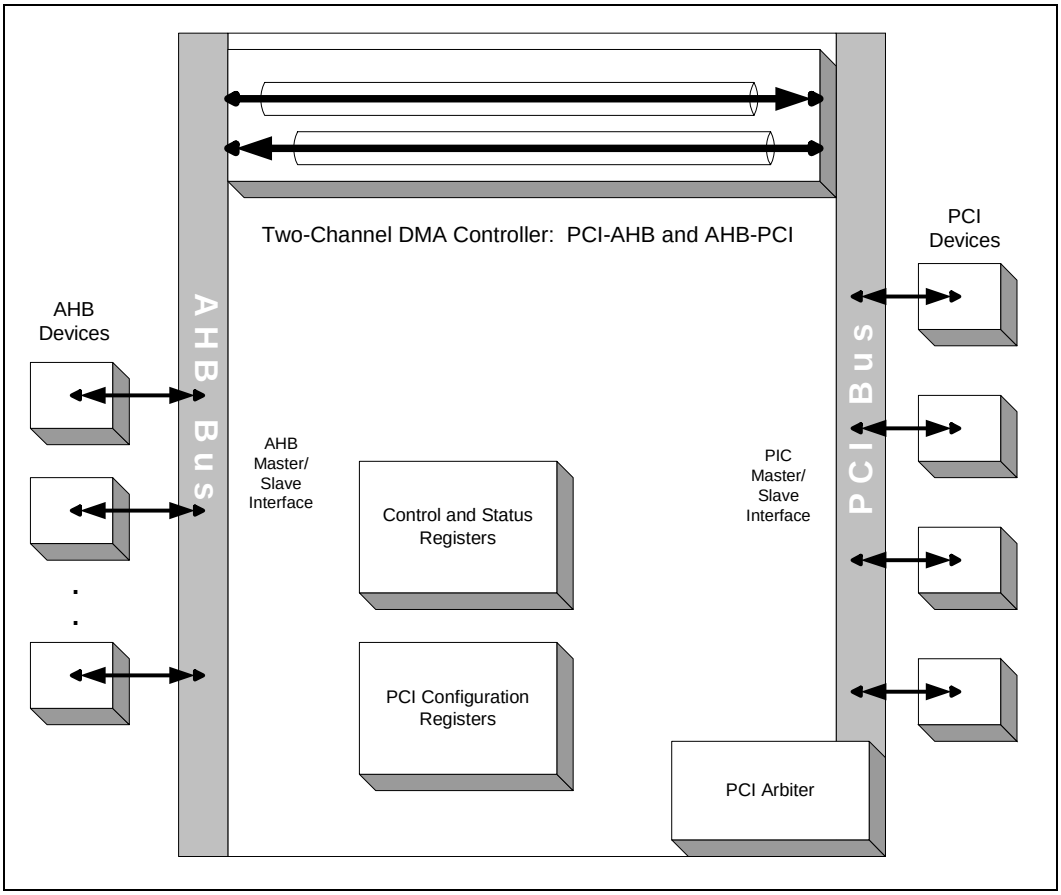


Table 57 lists the PCI-related APIs that are available.

### Table 57. API Summary

| | |
|---|---|
| **Initialization API**<br>SysPciInit | **PCI Special-Cycle-Generation API**<br>pciSpecialCycle |
| **Address-Mapping APIs**<br>SysPciMappingAdd<br>SysPciToPhys<br>SysPhysToPci | **PCI DMA-Access APIs**<br>pciDmaTransfer<br>pciDmaMbufTransfer<br>pciDmaShow |
| **Miscellaneous PCI-Support APIs**<br>SysPciAssignAddress<br>pciDevConfig | **PCI Interrupts APIs**<br>pciIntConnect<br>pciIntDisconnect<br>pciInt |
| **PCI Resources-Identification APIs**<br>pciInfoGet<br>pciInfoSearch<br>pciDeviceGet<br>pciFindDevice<br>pciFindClass<br>pciAddrShow<br>pciDeviceShow<br>pciHeaderShow<br>pciFindDeviceShow<br>pciFindClassShow | **PCI Data-Access APIs**<br>**PCI I/O-Access APIs**<br>pciIOOutByte<br>pciIOOutWord<br>pciIOOutLong<br>pciIOInByte<br>pciIOInWord<br>pciIOInLong<br>**PCI Memory-Access APIs**<br>pciMemOutByte<br>pciMemOutWord<br>pciMemOutLong<br>pciMemInByte<br>pciMemInWord<br>pciMemInLong |
| **PCI Configuration-Access APIs**<br>pciConfigInByte<br>pciConfigInWord<br>pciConfigInLong<br>pciConfigOutByte<br>pciConfigOutWord<br>pciConfigOutLong | **PCI Error-Support APIs**<br>pciErrorCallbackRegister |

Table 58 lists the major PCI data types available.

### Table 58. PCI Interface — Important Data Types

```
/*BAR – Base Address Register*/
enum PciBarId{CSR_BAR=0, IO_BAR, SD_BAR, NO_BAR};

typedef struct {
    UINT32 size;
    UINT32 address;
} PciBar;

/*PciDevice structures are actually mapped 1 to 1 with PCI functions, not devices.
Up to 8 functions may be supported by a single PCI device, though usually each
device only has 1 function*/
typedef struct {
    int bus;/*the bus segment on which the device is located*/
    int device;/*the device number assigned to this device*/
    int func;/*the function number assigned to this device*/
    int irq;      /*IRQ number used by this device*/
    BOOL error;/*flag indicating error on the device*/
    UINT16 vendor_id;/*Number identifying device vendor(Intel=x8086)*/
    UINT16 device_id;/*Number identifying device*/
    PciBar bar[7];/*Array of base address register descriptors*/
} PciDevice;
```

Table 59 shows the data that is declared global to the module (but not global to the entire system).

### Table 59. PCI Interface — Data Global to Module

```
static PciDevice devices[MAX_DEVICES]; /*array of device descriptors*/
static int nDevices;     /*Number of devices on the bus*/

typedef enum {
    PCI_OK,
    PCI_ERR_PSE,  /*System Error*/
    PCI_ERR_PFE,  /*Fatal Error*/
    PCI_ERR_PPE,  /*Parity Error*/
    PCI_ERR_AHBE  /*AHB Error*/
}PCI_STATUS;
typedef void (*PCICALLBACKFUNC)(PCI_STATUS,int);
```

## 6.1.1 Initialization APIs

### 6.1.1.1 sysPciInit

#### Description

This function initializes the IXP42X Product Line processor's PCI controller hardware and the PCI driver software.

The function detects whether the PCI controller is configured as a host or non-host, by reading strapping-pin options. When the PCI controller is configured as a host, this API will generate reset and clock signals for the PCI bus, according to the PCI-2.2 specification.

This function configures the PCI reset and PCI clock GPIOs as outputs driving 0. Wait 1 ms to satisfy the minimum reset-assertion time of the PCI specification. Configure the PCI clock GPIO for the proper PCI bus frequency. Enable the PCI clock GPIO to drive the PCI clock.

Wait 100 µs to satisfy the "minimum reset-assertion time from clock stable" requirement of the PCI specification. Set the PCI reset GPIO output to drive a 1. This releases the PCI bus.

By default, GPIO Pins 4 through 7 are configured as interrupts for PCI interrupt signals A through D. This may be overridden by modifying the following lines (in the BSP header file):

```
#define IXP425_PCI_INTA_PIN IXP425_GPIO_PIN_4
#define IXP425_PCI_INTB_PIN IXP425_GPIO_PIN_5
#define IXP425_PCI_INTC_PIN IXP425_GPIO_PIN_6
#define IXP425_PCI_INTD_PIN IXP425_GPIO_PIN_7
```

The pciInt routine (described below) is attached to these interrupt vectors.

When the IXP4XX Product Line processor's PCI controller is non-host, the host on the bus is assumed to provide PCI reset and PCI clock. Relevant GPIO pins are tri-stated.

#### Prototype

```
STATUS sysPciInit()
```

#### Parameters

None.

#### Impact on Global Data

None.

#### Pre-Conditions

This function must only be called once per system boot. Subsequent invocations will return an error.

**Return Codes**

- OK
- ERROR

## 6.1.2 Address-Mapping APIs

### 6.1.2.1 Overview

**Figure 70. PCI Memory Address Space to AHB Address Space**

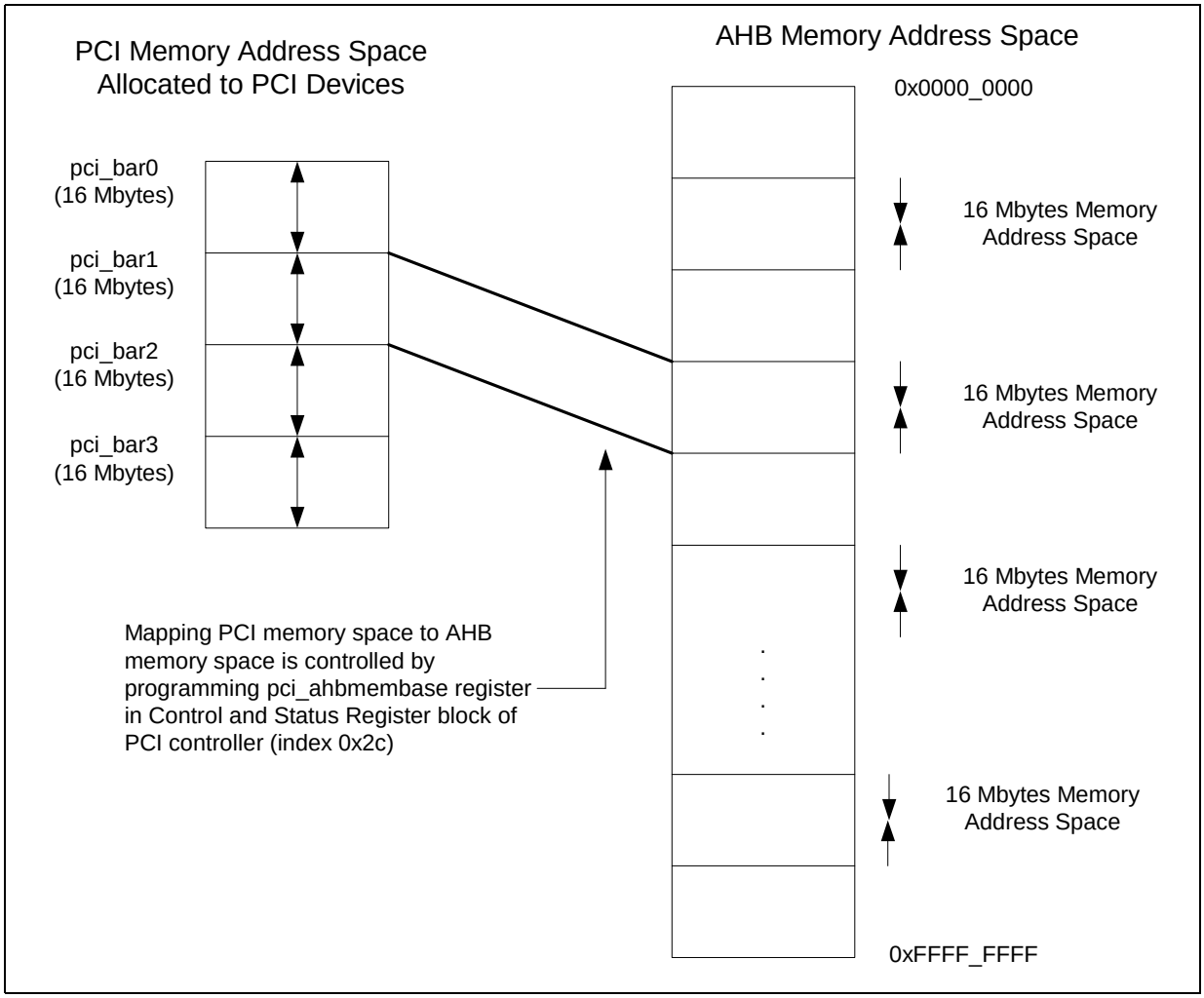


**Figure 71. PCI Memory Address Space to AHB Memory Address Space**

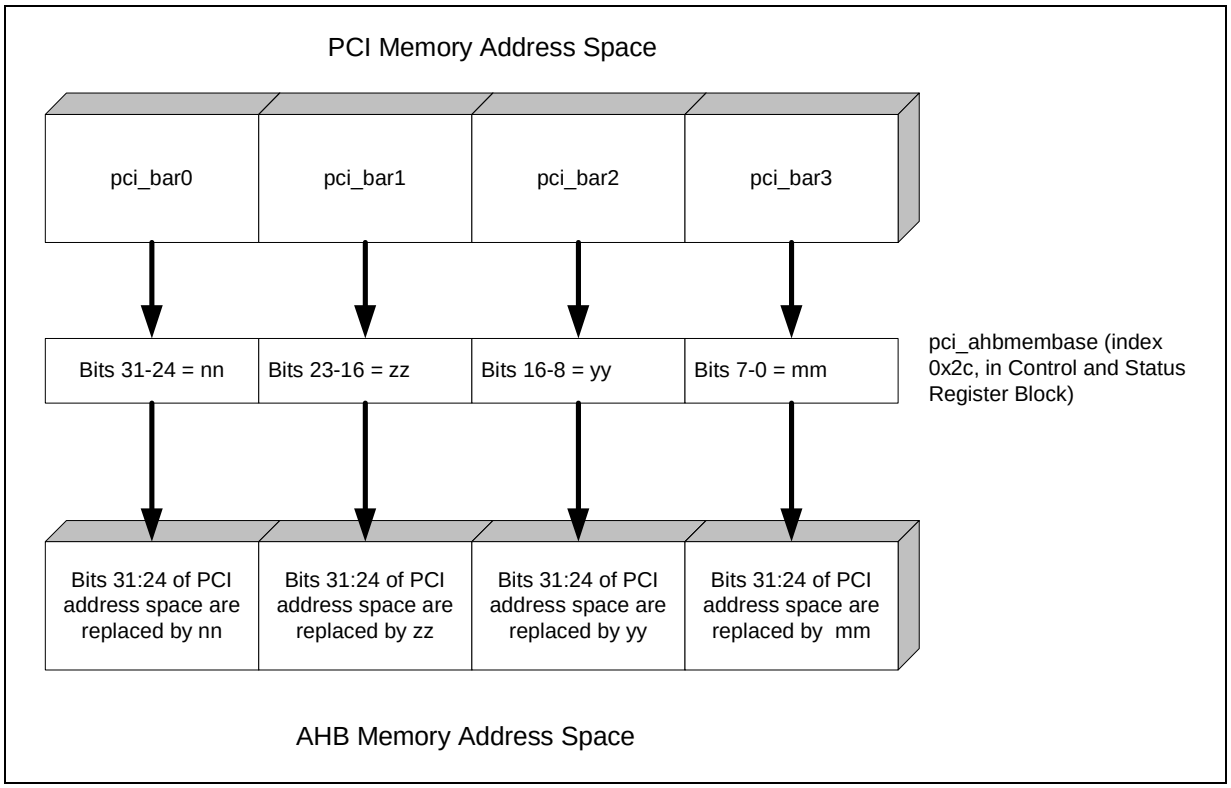


**Figure 72. PCI I/O Address Space to AHB Address Space**

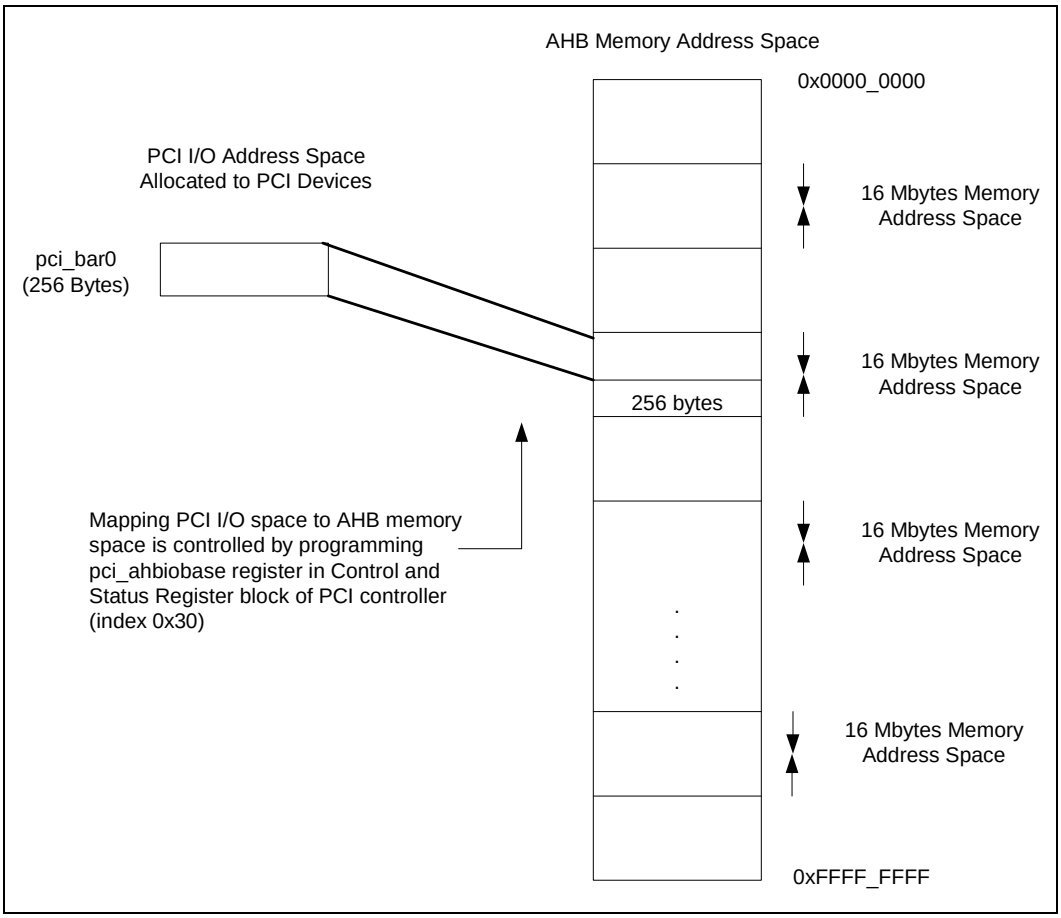


**Figure 73. PCI I/O Address Space to AHB Memory Address Space**

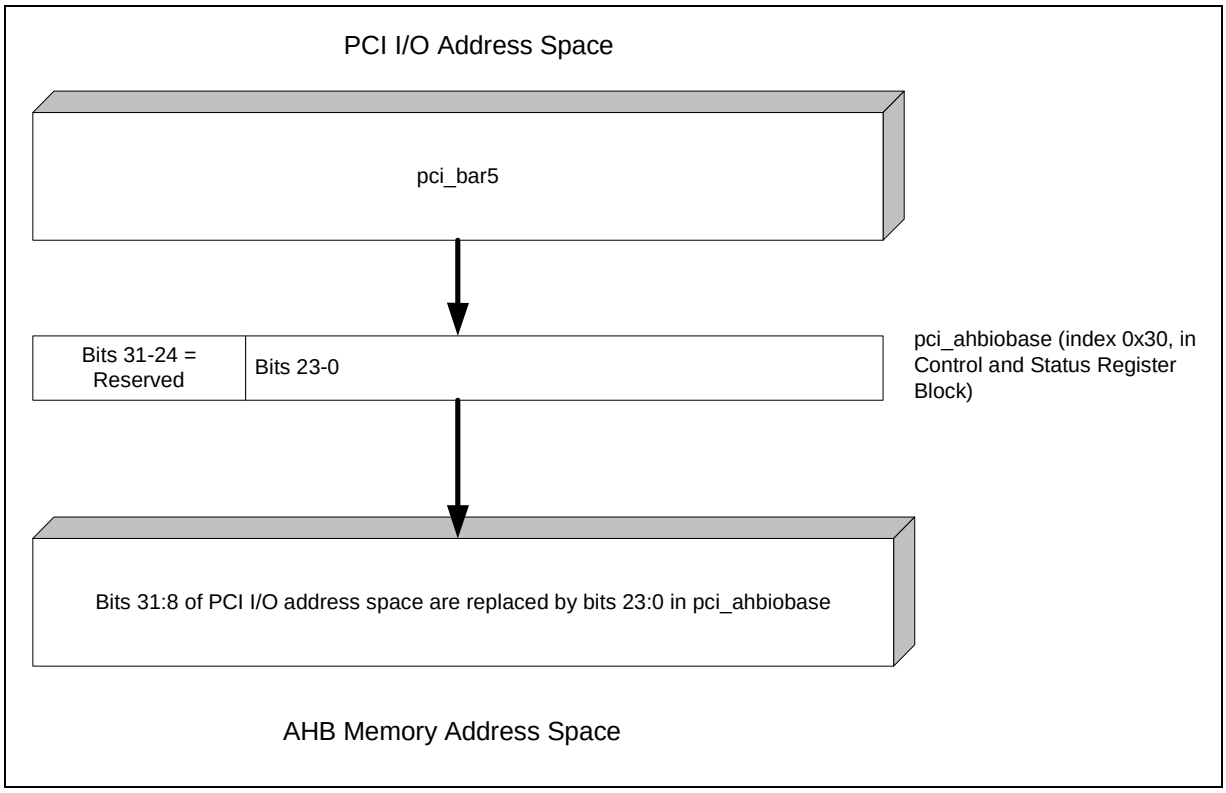


**Figure 74. AHB Memory Address Space to PCI Memory Address Space**

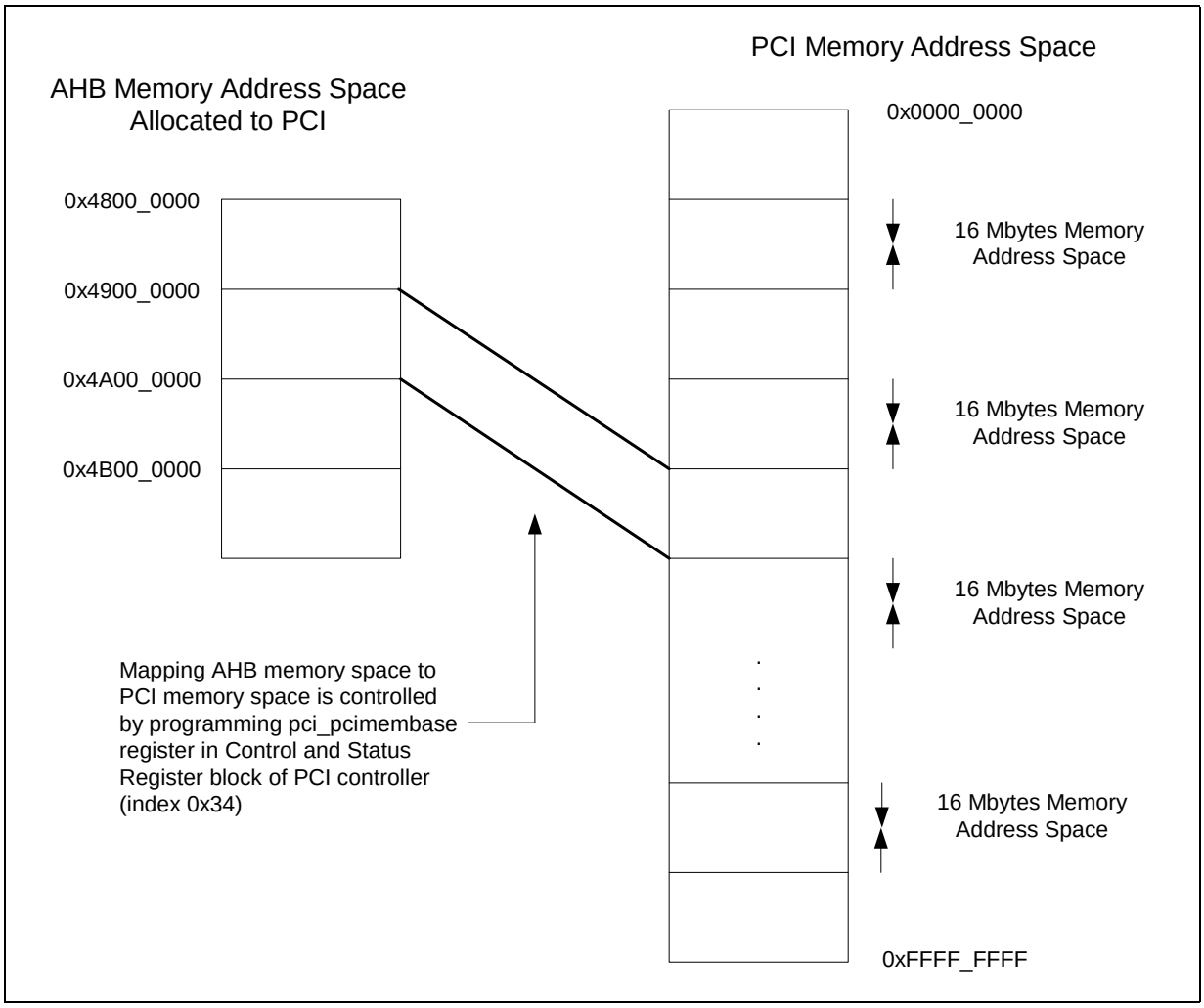
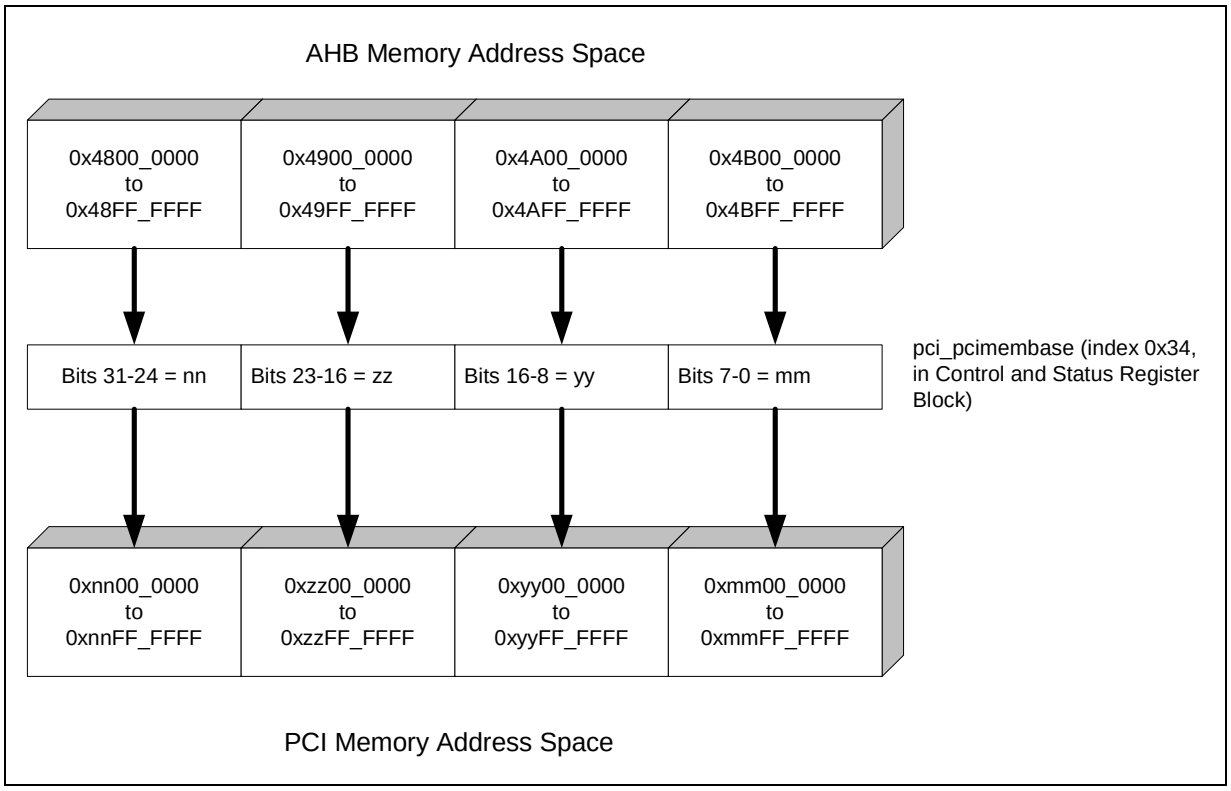


**Figure 75. AHB Memory Address Space to PCI Memory Address Space**




## 6.1.3 Mapping APIs

### 6.1.3.1 SysPciMappingAdd

#### Prototype

```
STATUS sysPciMappingAdd(UINT32 sdramAddress, UINT32 size)
```

#### Parameters

- sdramAddress — Address on the AHB bus that the PCI device needs to access through this mapping. The SDRAM address must be in the range 0x 0000_0000 to 0x 3FFF_FFFF.

  This address must have been allocated from the HEAP area, using the C-language, memory-allocation routines.

- size — Range of addresses (starting at sdramAddress) that the PCI device needs to access. The maximum size is 16 Mbytes.

#### Description

This function configures the next available slot (from a total of four) in the controller's AHBMEMBASE register, in order to permit a device on the PCI bus to access the specified range of SDRAM addresses.

#### Return Codes

- OK
- ERROR = If all available slots have been used.

### 6.1.3.2 sysPciToPhys

#### Prototype

```
void * sysPciToPhys (void * PciAddr)
```

#### Parameters

PciAddr specifies the PCI address that is to be converted.

#### Description

This function converts a PCI address to an Intel XScale core physical address. This only works for the PCI-memory space. This function typically is called by a code, running on the Intel XScale core, to access a particular PCI-memory address space.

#### Return Codes

The physical address is returned.

### 6.1.3.3 sysPhysToPci

#### Prototype

```
void * sysPhysToPci (void * PhysAddr)
```

#### Parameters

PhysAddr specifies the physical address that is to be converted to PCI memory address.

#### Description

This function converts a physical address to a PCI address. This only works for the PCI-memory space. A PCI process that needs to access the AHB memory address space typically calls this function.

#### Return Codes

The PCI address is returned.

## 6.1.4 Miscellaneous PCI-Support APIs

### 6.1.4.1 sysPciAssignAddress

#### Prototype

```
void sysPciAssignAddress(void)
```

#### Parameters

None.

#### Description

This function scans the PCI bus to see what other devices are there. It sorts the BARs that it finds and assigns them addresses and adds page entries to cover the range of PCI space used. It enables PCI transactions, as defined by INITIAL_PCI_CMD, which defaults to enabling I/O transactions.

As a special case, the function assigns its own SDRAM space to PCI address zero. This function only has an effect if this processor is the PCI host.

#### Impact on Global Data

The devices array is updated with discovered data.

#### Pre-Conditions

The PCI bus must have been initialized successfully.

#### Return Codes

None.

### 6.1.4.2 pciDevConfig

#### Prototype

```
STATUS pciDevConfig(int pciBusNo,intpciDevNo, int pciFuncNo, ULONG devIoBaseAdrs,
ULONG devMemBaseAdrs, ULONG command)
```

#### Parameters

- pciBusNo — Bus segment number on which the device, to be configured, resides.
- pciDevNo — Number of the device to be configured.
- pciFuncNo — Number of the function to be configured.
- devIoBaseAdrs — Base I/O address of the device.
- devMemBaseAdrs — Base memory address of the device.
- command — Command to be issued to the device.

#### Description

This routine configures a device that is on the PCI bus by writing to the configuration header of the selected device. It disables the device by clearing the command register in the configuration header, then sets the I/O and/or memory-space base address registers, the latency timer value, and the cache line size. Finally, it re-enables the device by loading the command register with the specified command.

***Note:*** This routine is designed for Type 0 PCI configuration headers only. It is *not* usable for configuring other things, such as a PCI-to-PCI bridge.

#### Return Codes

- OK
- Always

## 6.1.5 PCI Resources-Identification APIs

### 6.1.5.1 pciInfoGet

#### Prototype

```
PciDevice *pciInfoGet(UINT16 device)
```

#### Parameters

device — PCI device for returning information

#### Description

This function retrieves the PCI assignments of the specified device. It returns a pointer to a PciDevice structure containing the bus number, a device number, and a flag indicating whether there was an error for that device. It also returns the vendor/device ID and lists the BARs, indicating the size of the region and its address for that device.

If the specified device is not within range of the configured devices, a NULL pointer will be returned.

#### Return Codes

- Pointer to PCI device information
- NULL

### 6.1.5.2 pciInfoSearch

#### Prototype

```
PciDevice *pciInfoSearch (int bus, int device, int func, int count)
```

#### Parameters

- bus — Number identifying the bus segment on which the function is to be found.
- device — Device ID of the device containing the function.
- func — Function number.
- count — The n’th instance of functions matching the bus, device, and function specified. Will be returned, where n is the value specified in count.

#### Description

The array of PCI device descriptors is searched until the nth match is found.

#### Return Codes

- Pointer to PCI device information
- NULL

### 6.1.5.3 pciDeviceGet

#### Prototype

```
PciDevice *pciDeviceGet (int vendor_id, int device_id, int count)
```

#### Parameters

- vendor_id — Vendor ID to search for.
- device_id — Device ID to search for.
- count — The n’th matching device will be returned where n is specified in this parameter.

#### Description

This function returns a pointer to the n'th instance of a specified device. If the entry is not found, NULL is returned.

#### Return Codes

- Pointer to PCI device information
- NULL

### 6.1.5.4 pciFindDevice

#### Prototype

```
STATUS pciFindDevice(int vendorId, int deviceId, int index,
 int * pBusNo, int * pDeviceNo, int * pFuncNo);
```

#### Parameters

- vendorId — Vendor id to search for.
- deviceId — Device id to search for.
- index — The instance of device to return.
- pBusNo — Pointer to an integer into which the device's bus number will be placed.
- pDeviceNo — Pointer to an integer into which the matching device number will be placed.
- pFuncNo — Pointer to an integer into which the matching function number will be placed.

#### Description

This routine finds the nth device with the given device and vendor ID.

#### Return Codes

- OK
- ERROR = If the deviceId and vendorId didn't match.

### 6.1.5.5 pciFindClass

#### Prototype

```
STATUS pciFindClass(int classCode, int index, int * pBusNo, int * pDeviceNo, int *
FuncNo)
```

#### Parameters

- classCode — The 24-bit class code to search for.
- index — The instance of device to return.
- pBusNo — Pointer to an integer into which the device's bus number will be placed.
- pDeviceNo — Pointer to an integer into which the matching device number will be placed.
- pFuncNo — Pointer to an integer into which the matching function number will be placed.

#### Description

This routine finds the nth device with the given 24-bit PCI class code.

#### Return Codes

- OK
- ERROR = If the deviceId and vendorId didn't match.

### 6.1.5.6 pciAddrShow

#### Prototype

```
void pciAddrShow(void)
```

#### Parameters

None.

#### Description

This function prints the PCI address assignments that are automatically generated. The information appears in a table giving each device's bus number, device number, a flag (indicating whether there was an error for that device), and the vendor/device ID. The function then lists the BARs for that device, giving the size of the region and its start address.

If the low-order bit of the address is "1," the address corresponds to PCI I/O space. Otherwise, the address is in PCI memory space.

#### Return Codes

None.

### 6.1.5.7 pciDeviceShow

#### Prototype

```
STATUS pciDeviceShow (int busNo)
```

#### Parameters

busNo — Bus segment for which device information should be displayed.

#### Description

This routine prints information about all PCI devices on the specified bus.

#### Impact on Global Data

None.

#### Pre-Conditions

None.

#### Post-Conditions

None.

#### Return Codes

- OK
- ERROR = If the library is not initialized.

### 6.1.5.8 pciHeaderShow

#### Prototype

```
STATUS pciHeaderShow(int busNo, int deviceNo, int funcNo)
```

#### Parameters

- busNo — Bus segment for which device information should be displayed.
- deviceNo — Device for which information should be displayed.
- funcNo — Function for which information should be displayed.

#### Description

This routine prints information about the specified PCI function.

#### Return Codes

- OK
- ERROR = If the library is not initialized.

### 6.1.5.9 pciFindDeviceShow

#### Prototype

```
STATUS pciFindDeviceShow (int vendor_id, int device_id, int index)
```

#### Parameters

- vendor_id — Vendor ID to search for.
- device_id — Device ID to search for.
- index — Desired instance of the device.

#### Description

This routine prints information about the specified instance of a device matching the vendor and device IDs.

#### Return Codes

- OK
- ERROR = If the library is not initialized.

#### 6.1.5.10 pciFindClassShow

##### Prototype

```
STATUS pciFindClassShow (int classCode, int index)
```

##### Parameters

- classCode — 24-bit class code to search for.
- index — Instance of the device for which information should be displayed.

##### Description

This routine finds a device by its 24-bit PCI class code and prints its information.

##### Return Codes

- OK
- ERROR = If the library is not initialized.

### 6.1.6 PCI Configuration-Access APIs

#### 6.1.6.1 pciConfigInByte

##### Prototype

```
STATUS pciConfigInByte(int busNo, intdeviceNo, int funcNo, int address, UINT8 *
pData)
```

##### Parameters

- busNo — Bus segment on which the device resides.
- deviceNo — Number identifying device.
- funcNo — Number identifying the function.
- address — Configuration register address from which to read.
- pData — Location where data read is placed.

##### Description

This routine reads one byte from the PCI configuration space.

##### Pre-Conditions

PCI configuration base addresses have been configured through pciIomapLibInit.

##### Return Codes

- OK
- ERROR = If the library is not initialized.

### 6.1.6.2 pciConfigInWord

#### Prototype

STATUS pciConfigInWord(int busNo, int deviceNo, int funcNo, int address, UINT16 * pData);

#### Parameters

- busNo — Bus segment on which the device resides.
- deviceNo — Number identifying device.
- funcNo — Number identifying the function.
- address — Configuration register address from which to read.
- pData — Location where data read is placed.

#### Description

This routine reads one word from the PCI configuration space.

#### Pre-Conditions

PCI configuration base addresses have been configured through pciIomapLibInit.

#### Return Codes

- OK
- ERROR = If the library is not initialized.

### 6.1.6.3 pciConfigInLong

#### Prototype

```
STATUS pciConfigInLong(int busNo, intdeviceNo, int funcNo, int address, UINT32 *
pData)
```

#### Parameters

- busNo — Bus segment on which the device resides.
- deviceNo — Number identifying device.
- funcNo — Number identifying the function.
- address — Configuration register address from which to read.
- pData — Location where data read is placed.

#### Description

This routine reads one long-word (32 bits) from the PCI configuration space.

#### Pre-Conditions

PCI configuration base addresses have been configured through pciIomapLibInit.

#### Return Codes

- OK
- ERROR = If the library is not initialized.

### 6.1.6.4 pciConfigOutByte

#### Prototype

```
STATUS pciConfigOutByte(int busNo, int deviceNo, int funcNo, int address, UINT8
data)
```

#### Parameters

- busNo — Bus segment on which the device resides.
- deviceNo — Number identifying device.
- funcNo — Number identifying the function.
- address — Configuration register offset.
- data — Data to be written.

#### Description

This routine writes one byte to the PCI configuration space.

#### Pre-Conditions

PCI configuration base addresses have been configured through pciIomapLibInit.

#### Return Codes

- OK
- ERROR = If the library is not initialized.

### 6.1.6.5 pciConfigOutWord

#### Prototype

```
STATUS pciConfigOutWord(int busNo, int deviceNo, int funcNo, int address, UINT16
data)
```

#### Parameters

- busNo — Bus segment on which the device resides.
- deviceNo — Number identifying device.
- funcNo — Number identifying the function.
- address — Configuration register offset.
- data — Data to be written.

### Description

This routine writes one 16-bit word to the PCI configuration space.

### Pre-Conditions

PCI configuration base addresses have been configured through pciIomapLibInit.

### Return Codes

- OK
- ERROR = If the library is not initialized.

## 6.1.6.6 pciConfigOutLong

### Prototype

```
STATUS pciConfigOutLong(int busNo, intdeviceNo, int funcNo, int address, UINT32
data);
```

### Parameters

- busNo — Bus segment on which the device resides.
- deviceNo — Number identifying device.
- funcNo — Number identifying the function.
- address — Configuration register.
- data — Data to be written.

### Description

This routine writes one 32-bit-long word to the PCI configuration space.

### Pre-Conditions

PCI configuration base addresses have been configured through pciIomapLibInit.

### Return Codes

- OK
- ERROR = If the library is not initialized.

## 6.1.7 PCI Special-Cycle-Generation API

### 6.1.7.1 pciSpecialCycle

#### Prototype

```
STATUS pciSpecialCycle(int busNo, int message);
```

#### Parameters

busNo — Bus segment on which the special cycle is to be generated message. During the special cycle, the data is on AD[31:0].

#### Description

This routine generates a special cycle with a message.

#### Return Codes

- OK
- ERROR = If the library is not initialized.

## 6.1.8 PCI DMA-Access APIs

### 6.1.8.1 pciDmaTransfer

#### Prototype

```
STATUS pciDmaTransfer(void *buffer, void *pciAddr, int direction, int length
PCICALLBACKFUNC* callback, int parameter)
```

#### Parameters

- buffer — 32-bit-word-aligned, logical address of data buffer.
- pciAddr — 32-bit-word-aligned, logical PCI address to read/write.
- direction:
  - DMA_DIR_IN — A “0” bit indicates that the data is to be read from the device on the PCI bus and written to the AHB bus.
  - DMA_DIR_OUT — A “1”bit indicates that the data is to be read from AHB bus and written to the PCI bus.
- length — Number of words to transfer.
- callback — Function which will be invoked in ISR context, when the DMA transfer completes.
- parameter — Parameter that will be passed to the callback, when it is invoked.

  This will allow client code to determine which DMA transfer has completed, should there be multiple pending transfers.

### Description

The PCI controller core includes a two-channel DMA controller used to off-load transfers of large data blocks between target devices on the PCI and AHB buses. Each channel is unidirectional. One channel is used for AHB-to-PCI transfers and the other for PCI-to-AHB transfers.

Three registers are required to control each transfer: a source-address, destination-address, and word-count register. Two triplets are available for each channel that can be used in a ping-pong fashion to set up and perform back-to-back transfers across a given DMA channel.

DMA requests will be queued transparently by this function. Two queues are maintained, one for each channel. If the relevant queue is empty, the DMA transfer is invoked immediately. Otherwise, the request is placed at the end of the queue.

This function returns immediately having either initiated or queued the DMA request. When the DMA interrupt goes off at the end of a DMA transfer, the ISR will initiate the next transfer in the queue before invoking the callback function. On such successful completion of a transfer, the first parameter of the callback is set to PCI_OK.

If an error occurs in the processing of the DMA transfer, the client callback is invoked with the first parameter set to PCI_ERROR.

### Return Codes

- OK
- ERROR = If the parameters are invalid.

## 6.1.8.2 pciDmaMbufTransfer

### Prototype

```
void pciDmaMbufTransfer (mBlk* buffer, void* pciAddr, int direction,
PCICALLBACKFUNC* callback, int parameter)
```

### Parameters

- buffer — Pointer to mbuf chain.
- pciAddr — Logical address on the PCI bus to read/write.
- direction:
  - DMA_DIR_IN — A “0” bit indicates that data is to be read from the device on the PCI bus and written to the AHB bus.
  - DMA_DIR_OUT — A “1” bit indicates data is to be read from AHB bus and written to the PCI bus.
- callback — Function which will be invoked in ISR context, when the DMA transfer completes.
- parameter — Client-specified parameter passed to the callback, when it is invoked.

  This allows client code to determine which DMA transfer has completed should there be multiple pending transfers.

#### Description

This function performs an mbuf-oriented DMA transfer. Instead of data being transferred to or from a flat array in RAM, the data is transferred to or from an mbuf structure. This function is effectively a wrapper for pciDmaTransfer. It maintains a queue of mbuf-oriented transfers, separate to the queues maintained by pciDmaTransfer.

If a fragmented mBuf is passed to this function for transmission, multiple DMA transfers will be scheduled.

#### Return Codes

None.

### 6.1.8.3 pciDmaShow

#### Prototype

```
void pciDmaShow( )
```

#### Parameters

None.

#### Description

This function displays the current status of all DMA requests.

#### Return Codes

None.

## 6.1.9 PCI Interrupts APIs

### 6.1.9.1 pciIntConnect

#### Prototype

```
STATUS pciIntConnect(int vector, VOIDFUNCPTR routine, int parameter);
```

#### Parameters

- vector — Interrupt vector (number) to attach to.
- routine — Routine to be invoked for the specified vector.
- parameter — Parameter to be passed to the routine, on invocation.

#### Description

This routine connects an interrupt handler to the PCI interrupt line (A - D). A linked list is created, if multiple handlers are assigned to the single PCI interrupt. A separate list is maintained for each PCI interrupt (A-D) line.

The value passed, in the “vector” parameter, is treated as an integer and taken to be the IRQ number.

#### Impact on Global Data

#### Return Codes

- OK
- ERROR = If the interrupt handler cannot be built.

### 6.1.9.2 pciIntDisconnect

#### Prototype

```
STATUS pciIntDisconnect(int vector, VOIDFUNCPTR routine)
```

#### Parameters

- vector — Interrupt vector to detach.
- routine — Routine that was the interrupt service routine for the specified vector.

#### Description

This routine disconnects the interrupt handler from the PCI interrupt line.

#### Return Codes

- OK
- ERROR = If the library is not initialized.

### 6.1.9.3 pciInt

#### Prototype

```
VOID pciInt(int irq)
```

#### Parameters

irq — IRQ identifying which PCI interrupt (A - D) has occurred.

#### Description

This routine is invoked as the ISR for all PCI interrupts. It uses the IRQ parameter to determine which set of interrupt handlers to invoke for the PCI interrupt.

Each interrupt handler must check the device-dependent interrupt status bit to determine the source of the interrupt, because this routine simply executes all interrupt handlers in the linked list, corresponding to the PCI interrupt which has occurred.

#### Return Codes

None.

## 6.1.10 PCI Data-Access APIs

### 6.1.10.1 pciIOOutByte

#### Prototype

```
STATUS pciIOOutUINT8(void* pciAddr, UINT8 data)
```

#### Parameters

- pciAddr — Address to write to.
- data — Value to write.

#### Description

This routine writes a byte of data to the specified address in a PCI device's I/O space.

#### Return Codes

- OK
- ERROR = If the address is out of bounds.

### 6.1.10.2 pciIOOutWord

#### Prototype

```
STATUS pciIOOutByte(void* pciAddr, UINT16 data)
```

#### Parameters

- pciAddr — Address to write to.
- data — Value to write.

#### Description

This routine writes a 16-bit word of data to the specified address in a PCI device's I/O space.

#### Return Codes

- OK
- ERROR = If the address is out of bounds.

### 6.1.10.3 pciIOOutLong

#### Prototype

```
STATUS pciIOOutLong(void* pciAddr, UINT32 data)
```

#### Parameters

- pciAddr — Address to write to.
- data — Value to write.

#### Description

This routine writes a 32-bit long word of data to the specified address in a PCI device's I/O space.

#### Return Codes

- OK
- ERROR = If the address is out of bounds.

### 6.1.10.4 pciIOInByte

#### Prototype

```
STATUS pciIOInByte(void* pciAddr, UINT8* data)
```

#### Parameters

- pciAddr — Address to read from.
- data — Pointer to read data into.

#### Description

This routine reads a byte of data from a PCI device's I/O space

#### Return Codes

- OK
- ERROR = If the address is out of bounds.

### 6.1.10.5 pciIOInWord

#### Prototype

```
STATUS pciIOInWord(void* pciAddr, UINT16* data)
```

#### Parameters

- pciAddr — Address to read from.
- data — Pointer to read data into.

#### Description

This routine reads a 16-bit word of data from a PCI device's I/O space

#### Return Codes

- OK
- ERROR = If the address is out of bounds.

### 6.1.10.6 pciIOInLong

#### Prototype

```
STATUS pciIOInLong(void* pciAddr, UINT32* data)
```

#### Parameters

- pciAddr — Address to read from.
- data — Pointer to read data into.

#### Description

This routine reads a 32-bit long word of data from a PCI device's I/O space

#### Return Codes

- OK
- ERROR = If the address is out of bounds

### 6.1.10.7 pciMemOutByte

#### Prototype

```
STATUS pciMemOutByte(void* pciAddr, UINT8 data)
```

#### Parameters

- pciAddr — Address to write to
- data — Data to write.

#### Description

This routine writes a byte of data to a PCI device's memory space.

#### Return Codes

- OK
- ERROR = If the address is out of bounds.

### 6.1.10.8 pciMemOutWord

#### Prototype

```
STATUS pciMemOutWord(void* pciAddr, UINT16 data)
```

#### Parameters

- pciAddr — Address to write to.
- data — Data to write.

#### Description

This routine writes a 16-bit word of data to a PCI device's memory space.

#### Return Codes

- OK
- ERROR = If the address is out of bounds.

### 6.1.10.9 pciMemOutLong

#### Prototype

```
STATUS pciMemOutLong(void* pciAddr, UINT32 data)
```

#### Parameters

- pciAddr — Address to write to.
- data — Data to write.

#### Description

This routine writes a 32-bit-long word of data to a PCI device's memory space.

#### Return Codes

- OK
- ERROR = If the address is out of bounds.

### 6.1.10.10 pciMemInByte

#### Prototype

```
STATUS pciMemInByte(void* pciAddr, UINT8* data)
```

#### Parameters

- pciAddr — Address to read from.
- data — Pointer to read data into.

#### Description

This routine reads a byte of data from a PCI device's memory space.

#### Return Codes

- OK
- ERROR = If the address is out of bounds.

### 6.1.10.11 pciMemInWord

#### Prototype

```
STATUS pciMemInWord(void* pciAddr, UINT16* data)
```

#### Parameters

- pciAddr — Address to read from.
- data — Pointer to read data into.

#### Description

This routine reads a 16-bit word of data from a PCI device's memory space.

#### Return Codes

- OK
- ERROR = If the address is out of bounds.

### 6.1.10.12 pciMemInLong

#### Prototype

```
STATUS pciMemInLong(void* pciAddr, UINT32* data)
```

#### Parameters

- pciAddr — Address to read from.
- data — Pointer to read data into.

#### Description

This routine reads a 32-bit-long word of data from a PCI device's memory space.

#### Return Codes

- OK
- ERROR = If the address is out of bounds.

## 6.1.11 PCI Error-Support APIs

### 6.1.11.1 pciErrorCallbackRegister

#### Prototype

```
void pciErrorCallbackRegister(PCICALLBACKFUNC)
```

#### Parameters

None.

#### Description

This function allows client-code modules to register a callback which will be invoked on the occurrence of an error on the PCI bus that results in an interrupt. The callback will be invoked in the interrupt context.

#### Return Codes

None.

## 6.2 ADSL on the Intel® IXDP425 / IXCDP1100 Development Platform

The ADSL driver supports the Alcatel* ADSL chipset (MTC20156 Transceiver/Integrated controller) and its corresponding MTC20154 analog front end. The driver supports both ATU-R and ATU-C operation.

The two main interfaces to the DSL chipset are the expansion bus (control signals and firmware download) and UTOPIA, over which ATM traffic passes.

The ADSL driver is provided as part of the IXP400 software.

### 6.2.1 Functionality Overview

The Alcatel ADSL driver supports the Alcatel MTC20156 DMT Transceiver with ATM Framer/ Integrated Controller Preliminary and the Alcatel MTC20154 analogue front end. The driver uses the CTRL-E interface, which implements a method to control and receive status information from the Alcatel DynaMiTe chipset.

The interface is capable of handling data to and from the chipset to implement the ADSL MIB. The CTRL-E has a parallel implementation. The method of communication between the ADSL modem chipset and the IXP42X product line processors is via the parallel, CTRL-E interface.

An API is provided to bring the ADSL line up in either ATU-C or ATU-R mode. The line is configured to negotiate the best possible line rate, given the conditions of the local loop at start-up. The line rate is not renegotiated, once the modems are in the “show-time” mode.

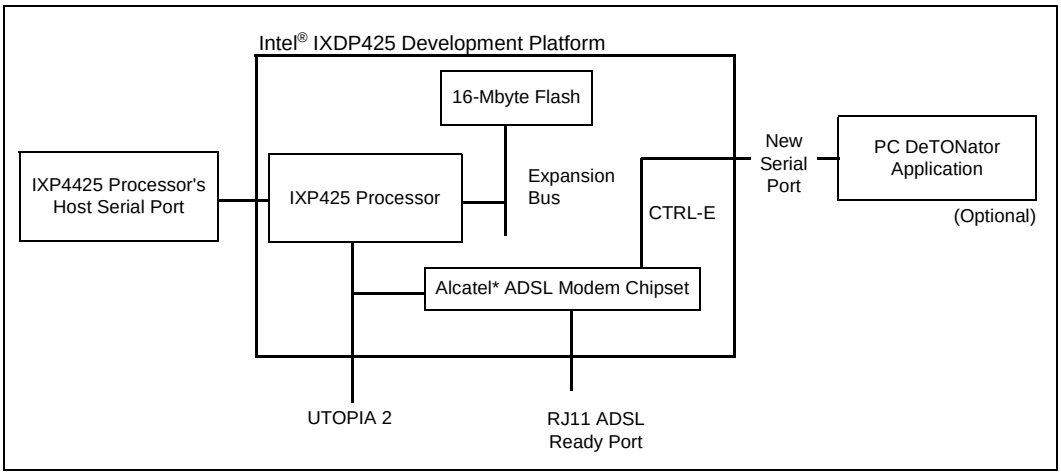
There is very little configuration information to be provided for the ATU-R side of the line. Almost all line configuration parameters are supplied to the ATU-C side. A generic set of start-up parameters is provided for the ATU-C modem.

APIs are provided to take the modem off line and to check the state of the line to see if the modem is in “show-time” mode.

#### 6.2.1.1 Controlling Alcatel* ADSL Modem Chipset Through Parallel CTRL-E

The Alcatel chipset is memory-mapped. The access routines provide memory read/write type functionality to the address space occupied by the Alcatel chipset. (See Figure 76.)

**Figure 76. Intel® - Alcatel* ADSL Joint Reference Design Board**




## 6.2.2 Interface

The following APIs are available to the user. The arguments and return codes are detailed in tAppendix A, “Application Programming Interfaces.”

- ixAdsl Module:
  - IxADSL_CPE_Open
  - IxADSL_CPE_LineClose
  - IxADSL_CO_Open
  - IxADSL_CO_LineClose
  - IxADSL_LineOpen
  - IxADSL_LineClose
  - IxADSL_LineStateReg
  - IxADSL_GetLineState
  - IxADSL_GetLineRateUpstream
  - IxADSL_GetLineRateDownstream
  - IxADSL_setVendorCode
  - IxADSL_Selftest
  - ixADSL_Show
  - ixAdslDriverInfoGet
  - ixAdslDriverInfoSet
  - ixAdslDriverTrapBind

The ADSL driver runs on the IXDP425 / IXCDP1100 platform. A download of the Alcatel* code — via the parallel CTRL-E — is provided. This removes the requirement for a flash device on the board associated with the Alcatel chipset.

## 6.2.3 Management Interface Functions

The management information interface functions implemented for the ADSL component can accommodate multi-indexed MIB tables. You can see a list of supported MIB attributes in Section 6.2.3.2. A provision that must be made in using these functions is that the calling function must allocate and manage the buffers that are passed.

The structure, ManagerInfoData, is used to pass untyped data between SNMP and ADSL management components. The structure of ManagerInfoData is shown below:

```
typedef struct {
void *dataPtr;/* A pointer to the data for/from management */
UINT32 dataLength;/* The length in bytes of the data pointed to */
} ManagerInfoData;
```

The dataLength element of this structure is used to accommodate octet strings that are not necessarily NULL terminated. The dataLength should therefore be checked before writing to the data buffer, to ensure the data sizes are consistent.

IxAdslManagerInfo is a typedef of an enumeration containing all the MIB objects that are supported by the ADSL component.

### 6.2.3.1 Supported Management Information Attributes

The tables in the next sections contain management information attributes specific to ADSL lines and channels. The attributes supported are a subset of RFC 2662 [6] and the mandatory objects needed to support RFC 1213 [5] as an interface.

The code supports a single ADSL line and either fast (ifType 125) or interleaved (ifType 124) channels. The channels are set up at initialization and remain of a particular type while the ADSL link is up. Both channels may not exist at the same time on the link.

### 6.2.3.2 Interfaces Table Support

The IxADSL component supports the interfaces table as defined in RFC 1213. The IANA have assigned three ifType(s) relative to ADSL and therefore it is possible to have an interface of each type.

There will always be a physical layer interface (adsl(94)) for the ADSL component. In the view of the stack table, the channels exist on top of this. Therefore, adsl(94) will be the stack lower level for either a fast (ifType 125) or interleaved (ifType 124) channel. Table 60 defines the MIB OBJECTs that are supported in the ifTable.

***Note:*** The concept of packets is undefined for ADSL as it uses a framing system. Therefore, any of the MIB objects that are based on packets are invalid for ADSL.

### Table 60. MIB Objects Supported in the ifTable (Sheet 1 of 2)

| Attribute | Access | Syntax | Values | Description |
|---|---|---|---|---|
| IxAdslIfDescr | TRO | OCTET | Textual String of Max 256 characters | Textual string describing the interface, e.g., Alcatel ADSL, software version 3.6.6. |
| IxAdslIfType | TRO | INT32 | • Physical - adsl(94)<br>• Fast - adslfast(125)<br>• Interleaved - adslinterleave(124) | Type of ADSL interface. The numerical values are assigned by the IANA. |
| IxAdslIfMtu | TRO | INT32 | (0..2147483647) | Size of the largest packet that can be sent/received on the interface, specified in octets. |
| IxAdslIfSpeed | TRO | UINT32 | (0..4294967295) | Transmit rate from the perspective of the agent:<br>• Physical - Line rate<br>• Fast - Channel rate<br>• Interleaved - Channel rate |
| IxAdslIfPhysAddress | TRO | OCTET | Contains an Octet string of zero length | Interface's address at its protocol sublayer. |
| IxAdslIfAdminStatus | TRW | INT32 | • up(1)<br>• down(2) | This is the desired state of the ADSL interface (Up/down). The NMS may use this variable to control the operation state of the interface. |
| IxAdslIfOperStatus | TRO | INT32 | • up(1)<br>• down(2) | Current operational status of the interface (Up/down). |
| IxAdslIfLastChange | TRO | UINT32 | (0..42494967295) | Value of sysUpTime at the time the interface entered its current operational state. |
| IxAdslIfInOctets | TRO | UINT32 | (0..42494967295) | Total number of octets received on the interface, including framing characters. |
| IxAdslIfInUcastPkts | TRO | UINT32 | (0..42494967295) | Number of packets, delivered by the sub-layer to a higher (sub-)layer, which were not addressed to a multicast or broadcast address at this sub-layer. |
| IxAdslIfInDiscards | TRO | UINT32 | (0..42494967295) | Number of inbound packets that were discarded even though no errors had been detected to prevent their being delivered to a high-layer protocol, e.g., free buffer space. |
| IxAdslIfInErrors | TRO | UINT32 | (0..42494967295) | Number of inbound packets that contained errors preventing them from being delivered to a higher-layer protocol. |
| IxAdslIfInUnknownProtos | TRO | UINT32 | (0..42494967295) | Number of packets received by the interface that were discarded because of an unknown or unsupported protocol. |
| IxAdslIfOutOctets | TRO | UINT32 | (0..42494967295) | Total number of octets transmitted out of the interface, including framing characters. |
| IxAdslIfOutUcastPkts | TRO | UINT32 | (0..42494967295) | Total number of packets that high-level protocols request to be transmitted, and which were not addressed to a multicast or broadcast address at this sub-layer, including those that were discarded or not sent. |
| IxAdslIfOutDiscards | TRO | UINT32 | (0..42494967295) | Number of outbound packets that were chosen to be discarded even though no errors had been detected to prevent their being transmitted, (for example, freeing buffer space). |
| IxAdslIfOutErrors | TRO | UINT32 | (0..42494967295) | For packet-oriented interfaces, the number of outbound packets that could not be transmitted because of errors. |

### Table 60. MIB Objects Supported in the ifTable (Sheet 2 of 2)

| Attribute | Access | Syntax | Values | Description |
|---|---|---|---|---|
| IxAdslIfName | TRO | OCTET | Textual string of Max 256 characters | Textual name of the interface. The value of this object should be suitable for use in commands entered at the “console” of the device. |
| IxAdslIfMulticastPkts | TRO | UINT32 | (0..42494967295) | Number of inbound packets, delivered by this layer to a higher layer, which were addressed to a multicast address. |
| IxAdslIfInBroadcastPkts | TRO | UINT32 | (0..42494967295) | Total number of outbound packets that high-level protocols requested to be transmitted, and were addressed to multicast multicast addresses, including those that were discarded or not sent. |
| IxAdslIfOutMulticastPkts | TRO | UINT32 | (0..42494967295) | Total number of outbound packets that higher-level protocols requested to be transmitted, and were addressed to multicast addresses, including those that were discarded or not sent. |
| IxAdslIfOutBroadcastPkts | TRO | UINT32 | (0..42494967295) | Total number of outbound packets that higher-level protocols requested to be transmitted, and were addressed to broadcast addresses, including those that were discarded or not sent. |
| IxAdslIfHighSpeed | TRO | UINT32 | (0..42494967295) | Speed of line in Mbps (IfSpeed/1,000,000). |
| IxAdslIfPromiscu-ousMode | TRO | INT32 | N/A | Not Applicable |
| IxAdslIfConnectorPresent | TRO | INT32 | • 1:true<br>• 2:false<br>Default values are:<br>• Physical - true(1)<br>• Fast - false(2)<br>• Interleaved - false(2) | • Value = “true(1)” — If the interface sub-layer has a physical connector<br>• Otherwise, value = ‘false(2)’ |
| IxAdslIfCounterDisconti-nuityTime | TRO | UINT32 | (0..42494967295) | Value of sysUpTime on the most recent occasion at which any one or more of this interface’s counters suffered a discontinuity. If no such discontinuities occurred since the last re-initialization of the local management subsystem, this object contains a zero value. |
| IxAdslIfStackLowerLevel | TRO | UINT32 | (0..42494967295) | This is the ifIndex of the interface that resides in the subsequent sub-layer below ADSL in the networking stack. |

## 6.2.3.3 IfTable Specific Traps

### Table 61. IfTable Specific Traps

| Name | Notification Event | Payload |
|---|---|---|
| IxAdsLinkDown | ADSL component notifies the SNMP agent in the event of its operational status going down. | ifIndex |
| IxAdslLinkUp | ADSL component notifies the SNMP agent in the event of its operational status going up. | ifIndex |

### 6.2.3.4 ADSL Line Table

Table 62 includes common attributes describing both ends of the ADSL line. It is required for all ADSL physical interfaces. ADSL physical interfaces are those ifEntries where ifType is equal to adsl(94).

**Table 62. ADSL Line Table**

| Attribute | Access | Syntax | Values | Description |
|---|---|---|---|---|
| IxAdslLineCoding | TRO | INT32 | N/A | Specifies the ADSL coding type used on this line. |
| IxAdslLineType | TRO | INT32 | 1. noChannel, no channels exist.<br>2. fastOnly, fast channel only exists.<br>3. interleavedOnly, interleaved channel only exists.<br>4. fastOrInterleaved, either fast or interleaved channels can exist, but only one at any time<br>5. fastAndInterleaved (5), both fast or interleaved | Defines the type of ADSL physical line entity that exists, by defining whether and how the line is channelized. If the Line is channelized, the ifStackTable can be used to determine the ifIndex of the associated channel(s). |

### 6.2.3.5 ADSL ATU-C Physical Table

Table 63 provides one row for each ATUC. Each row contains the physical-layer parameters table for that ATUC.

The ADSL physical interfaces are those ifEntries where ifType is equal to adsl(94).

**Table 63. ADSL ATU-C Physical Table**

| Attribute | Access | Syntax | Values | Description |
|---|---|---|---|---|
| IxAdslAtucInvVendorID | TRO | OCTET | Size: 0-16 Characters | Vendor ID code is a copy of the binary vendor identification field defined by the PHY and expressed as readable characters. |
| IxAdslAtucInvVersionNumber | TRO | OCTET | Size: 0-16 Characters | Vendor specific version number sent by this ATU as part of the initialization messages. It is a copy of the binary version number field defined by the PHY and expressed as readable characters. |
| IxAdslAtucCurrStatus | TRO | OCTET | 0. noDefect. There are no defects on the line.<br>1. lossOfFraming. ATUC failure due to not receiving valid frame.<br>2. lossOfSignal. ATUC failure due to not receiving signal.<br>3. lossOfPower. ATUC failure due to loss of power.<br>**NOTE:** The agent may still function.<br>4. lossOfSignalQuality. Loss of Signal Quality is declared when the Noise Margin falls below the minimum noise margin, or the bit-error-rate exceeds 10^-7.<br>5. lossOfLink. ATUC failure due to inability to link with ATUR.<br>6. dataInitFailure. ATUC failure during initialization due to bit errors corrupting start-up exchange data.<br>7. configInitFailure. ATUC failure during initialization due to peer ATU not able to support requested configuration<br>8. protocolInitFailure. ATUC failure during initialization due to incompatible protocol used by the peer ATU.<br>9. noPeerAtuPresent. ATUC failure during initialization due to no activation sequence detected from peer ATU. | Indicates current state of the ATUC line. This is a bit-map of possible conditions. It returns a char that contains the bitmap value. |
| IxAdslAtucCurrOutputPwr | TRO | INT32 | • Range: -310 to 310<br>• Units: tenth dBm | Measured total output power transmitted by this ATU. This is the measurement that was reported during the last activation sequence. |
| IxAdslAtucCurrAttainableRate | TRO | UINT32 | Units: bps | Indicates the maximum currently attainable data rate by the ATU. This value will be equal or greater than the current line rate. |

### 6.2.3.6 ADSL ATU-R Physical Table

Table 64 provides one row for each ATUR. Each row contains the physical-layer parameters table for that ATUR.

The ADSL physical interfaces are those ifEntries where ifType is equal to adsl(94).

**Table 64. ADSL ATU-R Physical Table**

| Attribute | Access | Syntax | Values | Description |
|---|---|---|---|---|
| IxAdslAturInvSerialNumber | TRO | OCTET | Size 0-32 characters | Vendor specific string that identifies the vendor equipment. |
| IxAdslAturInvVendorID | TRO | OCTET | Size 0-16 characters | Vendor ID code is a copy of the binary vendor identification field defined by the PHY and expressed as readable characters. |
| IxAdslAturInvVersionNumber | TRO | OCTET | Size 0-16 characters | Vendor specific version number sent by this ATU as part of the initialization messages. It is a copy of the binary version number field defined by the PHY and expressed as readable characters. |
| IxAdslAturCurrSnrMgn | TRO | INT32 | • –640 to +640<br>• Units: 0.1 dB | Noise margin, as seen by this ATU with respect to its received signal in tenth dB. |
| IxAdslAturCurrAtn | TRO | UINT32 | • 0 to 630<br>• Units: 0.1 dB | Difference in the total power transmitted by the peer ATU and the total power received by this ATU. |
| IxAdslAturCurrStatus | TRO | OCTET | 0. noDefect. There no defects on the line.<br>1. lossOfFraming. ATUR failure due to not receiving valid frame.<br>2. lossOfSignal. ATUR failure due to not receiving signal.<br>3. lossOfPower. ATUR failure due to loss of power.<br>4. lossOfSignalQuality. Loss of Signal Quality is declared when the noise margin falls below the minimum noise margin, or the bit-error-rate exceeds 10^-7. | Indicates current state of the ATUR line. This is a bitmap of possible conditions. It returns a char that contains the bitmap value Due to the isolation of the ATUR when line problems occur, many state conditions like loss of power, loss of quality signal, and initialization errors, cannot be determined. While trouble shooting ATUR, also use object, adslAtucCurrStatus. |
| IxAdslAturCurrOutputPwr | TRO | INT32 | • -310 to 310<br>• Units: 0.1 dB | Measured total output power transmitted by this ATU. This is the measurement that was reported during the last activation sequence. |
| IxAdslAturCurrAttainableRate | TRO | UINT32 | Units: bps | Indicates the maximum currently attainable data rate by the ATU. This value will be equal or greater than the current line rate. This is the Rx rate on an ATU-R |

### 6.2.3.7 ADSL ATU-R Channel Table

Table 65 provides one row for each ATU-R channel. ADSL channel interfaces are those ifEntries where ifType is equal to adslInterleave(124) or adslFast(125)

**Table 65. ADSL ATU-R Channel Table**

| Attribute | Access | Syntax | Values | Description |
|---|---|---|---|---|
| IxAdslAturChanInterleaveDelay | TRO | UINT32 | Units: ms | Interleave delay for this channel.<br>Interleave delay applies only to the interleave channel and defines the mapping (relative spacing) between subsequent input bytes at the interleaver input and their placement in the bit stream at the interleaver output. Larger numbers provide greater separation between consecutive input bytes in the output bit stream allowing for improved impulse noise immunity at the expense of payload latency. In the case where the ifType is Fast(125), use noSuchObject. |
| IxAdslAturChanCurrTxRate | TRO | UINT32 | Units: bps | Actual transmit rate on this channel. |
| IxAdslAturChanPrevTxRate | TRO | UINT32 | Units: bps | Rate at the time of the last adslAturRateChangeTrap event. It is also set at initialization to prevent a trap being sent. Rate changes less than adslAturThresh(*)RateDown or less than adslAturThresh(*)RateUp will not cause a trap or cause this object to change. (*) == Fast or Interleave.<br>See AdslLineAlarmConfProfileEntry. |
| IxAdslAturChanCrcBlockLength | TRO | UINT32 | Units: Byte | Indicates the length of the channel data-block on which the CRC operates. Refer to line-code-specific MIBs |

### 6.2.3.8 ADSL ATU-R Performance Data Table

Table 66 provides one row for each ATUR. The ADSL physical interfaces are those ifEntries where ifType is equal to adsl(94).

**Table 66. ADSL ATU-R Performance Data Table**

| Attribute | Access | Syntax | Values | Description |
|---|---|---|---|---|
| IxAdslAturPerfLofs | TRO | UINT32 | Units: Seconds | Count of the number of seconds since the agent was last reset that contain Loss of Framing failures. |
| IxAdslAturPerfLoss | TRO | UINT32 | Units: Seconds | Count of the number of seconds since the agent was last reset that contain Loss of Signal failures. |
| IxAdslAturPerfLprs | TRO | UINT32 | Units: Seconds | Count of the number of seconds since the agent was last reset that contain Loss of Power failures. |
| IxAdslAturPerfESs | TRO | UINT32 | Units: Seconds | Count of the number of errored seconds since agent reset. The errored second parameter is a count of one-second intervals containing one or more CRC anomalies, or one or more los or sef defects. |

## 6.2.4 Modem Self Test

Access is provided to modem self test features, called from the bridge application prior to bringing up the bridge.

## 6.2.5 Debug

An API is provided to display any statistics considered useful in the debug of simple line conditions.

## 6.2.6 ADSL MIB Compliance

### 6.2.6.1 Mandatory Groups for ATU-C Agent Compliance

#### adslLineMibAtucCompliance MODULE-COMPLIANCE

**Status**
Current.

**Description**
The compliance statement for SNMP entities which manage ADSL ATU-C interfaces.

**Module**
This Module.

**Mandatory Groups**
{
adslLineGroup,
adslPhysicalGroup,
adslChannelGroup,
adslAtucPhysPerfIntervalGroup,
adslAturPhysPerfIntervalGroup,
adslLineConfProfileGroup,
adslLineAlarmConfProfileGroup,
adslLineConfProfileControlGroup
}

#### adslLineGroup OBJECT-GROUP

{
adslLineCoding - supported
adslLineType - supported
adslLineSpecific — Not supported
}

### adslPhysicalGroup OBJECT-GROUP

{
adslAtucInvSerialNumber — Supported
adslAtucInvVendorID — Not supported
adslAtucInvVersionNumber — Not supported
adslAtucCurrSnrMgn — Supported
adslAtucCurrAtn — Supported
adslAtucCurrStatus — Not supported
adslAtucCurrOutputPwr — Not supported
adslAtucCurrAttainableRate — Supported
adslAturInvSerialNumber — Supported
adslAturInvVendorID — Supported
adslAturInvVersionNumber — Supported
adslAturCurrSnrMgn — Supported
adslAturCurrAtn — Supported
adslAturCurrStatus — Supported
adslAturCurrOutputPwr — Supported
adslAturCurrAttainableRate — Supported
}

### adslChannelGroup OBJECT-GROUP

{
adslAtucChanInterleaveDelay — Not supported
adslAtucChanCurrTxRate — Not supported
adslAtucChanPrevTxRate — Not supported
adslAtucChanCrcBlockLength — Not supported
adslAturChanInterleaveDelay — Supported
adslAturChanCurrTxRate — Supported
adslAturChanPrevTxRate — Supported
adslAturChanCrcBlockLength — Not supported
}

#### Status

Current.

#### Description

A collection of objects providing configuration information about an ADSL channel.

### adslAtucPhysPerfIntervalGroup

Not supported

### adslAturPhysPerfIntervalGroup OBJECT-GROUP

{
adslAturPerfValidIntervals — Supported
adslAturPerfInvalidIntervals — Supported
adslAturPerfCurr15MinTimeElapsed — Supported
adslAturPerfCurr15MinLofs — Supported
adslAturPerfCurr15MinLoss — Supported
adslAturPerfCurr15MinLprs — Supported
adslAturPerfCurr15MinESs — Supported
adslAturPerfCurr1DayTimeElapsed — Supported
adslAturPerfCurr1DayLofs — Supported
adslAturPerfCurr1DayLoss — Supported
adslAturPerfCurr1DayLprs — Supported
adslAturPerfCurr1DayESs — Supported
adslAturPerfPrev1DayMoniSecs — Supported
adslAturPerfPrev1DayLofs — Supported
adslAturPerfPrev1DayLoss — Supported
adslAturPerfPrev1DayLprs — Supported
adslAturPerfPrev1DayESs — Supported
adslAturIntervalLofs — Supported
adslAturIntervalLoss — Supported
adslAturIntervalLprs — Supported
adslAturIntervalESs — Supported
adslAturIntervalValidData — Supported
}

#### Status

Current

#### Description

A collection of objects providing current 15-minute, 1-day; and previous 1-day performance counts on ADSL Line (ATU-R end).

### adslLineConfProfileGroup

Not supported

### adslLineAlarmConfProfileGroup OBJECT-GROUP

{
adslAtucThresh15MinLofs — Not supported
adslAtucThresh15MinLoss — Not supported
adslAtucThresh15MinLols — Not supported
adslAtucThresh15MinLprs — Not supported
adslAtucThresh15MinESs — Not supported
adslAtucThreshFastRateUp — Not supported
adslAtucThreshInterleaveRateUp — Not supported
adslAtucThreshFastRateDown — Not supported
adslAtucThreshInterleaveRateDown — Not supported
adslAtucInitFailureTrapEnable — Not supported
adslAturThresh15MinLofs — Supported
adslAturThresh15MinLoss — Supported

adslAturThresh15MinLprs — Supported
adslAturThresh15MinESs — Supported
adslAturThreshFastRateUp – not supported as there is no dynamic rate adaptation support in the PHY.
adslAturThreshInterleaveRateUp — Not supported as there is no dynamic rate adaptation support in the PHY.
adslAturThreshFastRateDown — Not supported as there is no dynamic rate adaptation support in the PHY.
adslAturThreshInterleaveRateDown — Not supported as there is no dynamic rate adaptation support in the PHY.
}

**Status**
Current.

**Description**
A collection of objects providing alarm provisioning information about an ADSL Line.

#### adslLineConfProfileControlGroup

Not supported

### 6.2.6.2 Mandatory Groups for ATU-R Agent Compliance

#### adslLineMibAturCompliance MODULE-COMPLIANCE

**Status**
Current

**Description**
The compliance statement for SNMP entities which manage ADSL ATU-R interfaces.

**Module**
This module

**Mandatory Groups**
{
adslAturLineGroup,
adslAturPhysicalGroup,
adslAturChannelGroup,
adslAturAtucPhysPerfIntervalGroup,
adslAturAturPhysPerfIntervalGroup,
adslAturLineAlarmConfProfileGroup,
adslAturLineConfProfileControlGroup
}

### adslAturLineGroup OBJECT-GROUP

**Objects**
{
adslLineCoding — Supported
}

**Description**
A collection of objects providing configuration information about an ADSL Line on the ATU-R side.

### adslAturPhysicalGroup OBJECT-GROUP

**Objects**
{
adslAtucInvVendorID — Not supported
adslAtucInvVersionNumber — Not supported
adslAtucCurrStatus — Not supported
adslAtucCurrOutputPwr — Not supported
adslAtucCurrAttainableRate — Supported
adslAturInvSerialNumber — Supported
adslAturInvVendorID — Supported
adslAturInvVersionNumber — Supported
adslAturCurrSnrMgn — Supported
adslAturCurrAtn — Supported
adslAturCurrStatus — Supported
adslAturCurrOutputPwr — Supported
adslAturCurrAttainableRate — Supported
}

**Status**
Current.

**Description**
A collection of objects providing physical configuration information of the ADSL Line on the ATU-R side.

### adslAturChannelGroup OBJECT-GROUP

**Objects**
{
adslAtucChanInterleaveDelay — Not supported
adslAtucChanCurrTxRate — Not supported
adslAtucChanPrevTxRate — Not supported
adslAturChanInterleaveDelay — Supported
adslAturChanCurrTxRate — Supported
adslAturChanPrevTxRate — Supported
adslAturChanCrcBlockLength — Not supported
}

**Status**
Current.

### Description
A collection of objects providing configuration information about an ADSL channel on the ATU-R side.

### adslAturAtucPhysPerfIntervalGroup

Not supported.

### adslAturAturPhysPerfIntervalGroup

All supported.

### adslAturLineAlarmConfProfileGroup OBJECT-GROUP

### Objects
{
adslAtucThresh15MinLofs — Not supported
adslAtucThresh15MinLoss — Not supported
adslAtucThresh15MinESs — Not supported
adslAtucThreshFastRateUp — Not supported
adslAtucThreshInterleaveRateUp — Not supported
adslAtucThreshFastRateDown — Not supported
adslAtucThreshInterleaveRateDown — Not supported
adslAtucInitFailureTrapEnable — Not supported
adslAturThresh15MinLofs — Supported
adslAturThresh15MinLoss — Supported
adslAturThresh15MinLprs — Supported
adslAturThresh15MinESs — Supported
adslAturThreshFastRateUp — Not supported as there is no dynamic rate adaptation support in the PHY.
adslAturThreshInterleaveRateUp — Not supported as there is no dynamic rate adaptation support in the PHY.
adslAturThreshFastRateDown — Not supported as there is no dynamic rate adaptation support in the PHY.
adslAturThreshInterleaveRateDown — Not supported as there is no dynamic rate adaptation support in the PHY.
}

### Status
Current.

### Description
A collection of objects providing alarm provisioning information about an ADSL Line provided by the ATU-R agent.

#### adslAturLineConfProfileControlGroup OBJECT-GROUP

**Objects**
{
adslLineAlarmConfProfile — Not supported
adslLineAlarmConfProfileRowStatus — Not supported
}

**Status**
Current.

**Description**
A collection of objects providing profile control for the ADSL system by the ATU-R agent.

### 6.2.6.3 ADSL MIB Trap Notification Support

The following is a list of the ADSL MIB Trap Notifications.
ADSL MIB compliance is independent of support for the following Notification Traps.

#### adslNotificationsGroup NOTIFICATION-GROUP

**Notifications**
{
adslAtucPerfLofsThreshTrap, — Not supported
adslAtucPerfLossThreshTrap, — Not supported
adslAtucPerfLprsThreshTrap, — Not supported
adslAtucPerfESsThreshTrap, — Not supported
adslAtucRateChangeTrap, — Not supported
adslAtucPerfLolsThreshTrap, — Not supported
adslAtucInitFailureTrap, — Not supported.
adslAturPerfLofsThreshTrap, — Supported
adslAturPerfLossThreshTrap, — Supported
adslAturPerfLprsThreshTrap, — Supported
adslAturPerfESsThreshTrap, — Supported
adslAturRateChangeTrap — Supported
}

**Status**
Current.

**Description**
The collection of ADSL notifications.

This page intentionally left blank.

# Application Programming Interfaces A

This appendix documents the Intel® IXP400 Software v1.2.2's access-layer API, codelet modules, and data structures.

## A.1 Intel® IXP42X product line Access-Layer API and Codelet Module Index


- ADSL Driver (IxAdsl) API . . . . . 233
- Assertion Macros (IxAssert) API . . . . . 245
- ATM Driver Access (IxAtmdAcc) API . . . . . 245
- ATM Driver Access (IxAtmdAcc) Control API . . . . . 264
- ATM Driver Access (IxAtmdAcc) UTOPIA Control API . . . . . 282
- ATM Manager (IxAtmm) API . . . . . 284
- ATM Transmit Scheduler (IxAtmSch) API . . . . . 295
- ATM Types (IxAtmTypes) . . . . . 302
- Security (IxCryptoAcc) API . . . . . 306
- DMA Types (IxDmaTypes) . . . . . 316
- DMA Access Driver (IxDmaAcc) API . . . . . 319
- Ethernet Access (IxEthAcc) API . . . . . 322
- Ethernet Access Fast Path (IxEthAccFpathDep) API . . . . . 346
- Ethernet Database (IxEthDB) API . . . . . 347
- Ethernet Database Port Definitions (IxEthDBPortDefs) . . . . . 356
- Ethernet NPE (IxEthNpe) API . . . . . 357
- Fast Path Access (IxFpathAcc) API . . . . . 367
- HSS Access (IxHssAcc) API . . . . . 377
- NPE-A (IxNpeA) API . . . . . 398
- NPE-Downloader (IxNpeDl) API . . . . . 408
- NPE Message Handler (IxNpeMh) API . . . . . 415
- OS Memory Buffer Management (IxOsBuffMgt) API . . . . . 423
- OS Memory Buffer Pool Management (IxOsBuffPoolMgt) API . . . . . 423
- OS Services (IxOsServices) API . . . . . 432
- Queue Manager (IxQMgr) API . . . . . 440
- Timer Control (IxTimerCtrl) API . . . . . 460

- Types (IxTypes) . . . . . 464
- UART Access (IxUARTAcc) API . . . . . 465
- Version ID (IxVersionId) . . . . . 469
- USB Access (IxUsb) API . . . . . 469
- Codelet Overview . . . . . 476
  - AAL5 Main Codelet (IxAal5CodeletMain) API . . . . . 476
  - Aal5 HSS Codelet (IxAal5HssCodelet) API . . . . . 477
  - Atm Codelet (IxAtmCodelet) API . . . . . 478
  - Crypto Access Codelet (IxCryptoAccCodelet) API . . . . . 485
  - Demo Utils (IxDemoUtils) API . . . . . 491
  - DMA Access Codelet (IxAal5Demo) API. . . . . 496
  - Ethernet Aal5 (IxEthAal5App) API. . . . . 498
  - Ethernet Access Codelet (IxEthAccCodelet) API . . . . . 500
  - Fast Path Access Codelet (IxFpathAccCodelet) API. . . . . 504
  - HSS Access Codelet (IxHssAccCodelet) API . . . . . 513
  - HSS Access Codelet Channelised (IxHssAccCodeletChan) . . . . . 522
  - HSS Access Codelet Common (IxHssAccCodeletCom) API . . . . . 523
  - HSS Access Codelet Config (IxHssAccCodeletConfig) API . . . . . 525
  - HSS Access Codelet MBuf Pool (IsHssAccCodeletMbuf) API . . . . . 526
  - HSS Access Codelet Packetised (IxHssAccCodeletPkt) API . . . . . 527
  - Oam Codelet (IxOamCodelet) API . . . . . 528
  - UART Access Codelet (IxUARTAccCodelet) API . . . . . 533
  - USB RNDIS Codelet (IxUSBRNDIS) API . . . . . 536
  - USB RNDIS End Driver Codelet (IxUSBRNDIS) API . . . . . 542
  - USB RNDIS Vendor Codelet (IxUSBRNDIS) . . . . . 542

# A.2 Intel® IXP42X product line Access-Layer API and Codelet Data Structure Index


- Aal5Statistics Struct Reference . . . . . 543
- ActiveIpFlow Struct Reference . . . . . 543
- ChannelisedStats Struct Reference . . . . . 543
- GeneralStats Struct Reference . . . . . 544
- ix_s_ossl_thread_main_info_t Struct Reference . . . . . 545
- ix_s_ossl_time_t Struct Reference . . . . . 545
- IxAtmdAccUtopiaConfig Struct Reference . . . . . 545
- IxAtmdAccUtopiaConfig::UtRxConfig_ Struct Reference . . . . . 547
- IxAtmdAccUtopiaConfig::UtRxDefineIdle_ Struct Reference . . . . . 550
- IxAtmdAccUtopiaConfig::UtRxEnableFields_ Struct Reference . . . . . 551
- IxAtmdAccUtopiaConfig::UtRxStatsConfig_ Struct Reference . . . . . 555
- IxAtmdAccUtopiaConfig::UtRxTransTable0_ Struct Reference . . . . . 556
- IxAtmdAccUtopiaConfig::UtRxTransTable1_ Struct Reference . . . . . 557
- IxAtmdAccUtopiaConfig::UtRxTransTable2_ Struct Reference . . . . . 558
- IxAtmdAccUtopiaConfig::UtRxTransTable2_ Struct Reference . . . . . 558
- IxAtmdAccUtopiaConfig::UtRxTransTable3_ Struct Reference . . . . . 558
- IxAtmdAccUtopiaConfig::UtRxTransTable4_ Struct Reference . . . . . 559
- IxAtmdAccUtopiaConfig::UtRxTransTable5_ Struct Reference . . . . . 560
- IxAtmdAccUtopiaConfig::UtSysConfig_ Struct Reference . . . . . 560
- IxAtmdAccUtopiaConfig::UtTxConfig_ Struct Reference . . . . . 562
- IxAtmdAccUtopiaConfig::UtTxDefineIdle_ Struct Reference . . . . . 564
- IxAtmdAccUtopiaConfig::UtTxEnableFields_ Struct Reference . . . . . 565
- IxAtmdAccUtopiaConfig::UtTxStatsConfig_ Struct Reference . . . . . 568
- IxAtmdAccUtopiaConfig::UtTxTransTable0_ Struct Reference . . . . . 569
- IxAtmdAccUtopiaConfig::UtTxTransTable1_ Struct Reference . . . . . 569
- IxAtmdAccUtopiaConfig::UtTxTransTable2_ Struct Reference . . . . . 570
- IxAtmdAccUtopiaConfig::UtTxTransTable3_ Struct Reference . . . . . 571
- IxAtmdAccUtopiaConfig::UtTxTransTable4_ Struct Reference . . . . . 571
- IxAtmdAccUtopiaConfig::UtTxTransTable5_ Struct Reference . . . . . 572
- IxAtmdAccUtopiaStatus Struct Reference . . . . . 572
- IxAtmdAccUtopiaStatus::UtRxCellConditionStatus_ Struct Reference . . . . . 573
- IxAtmdAccUtopiaStatus::UtTxCellConditionStatus_ Struct Reference . . . . . 574
- IxAtmmPortCfg Struct Reference . . . . . 575

- IxAtmmVc Struct Reference . . . . . 575
- IxAtmScheduleTable Struct Reference . . . . . 576
- IxAtmScheduleTableEntry Struct Reference . . . . . 577
- IxAtmTrafficDescriptor Struct Reference . . . . . 578
- IxCryptoAccAuthCtx Struct Reference . . . . . 579
- IxCryptoAccCipherCtx Struct Reference . . . . . 579
- IxCryptoAccCtx Struct Reference . . . . . 580
- IxEthAccMacAddr Struct Reference . . . . . 581
- IxEthDBMacAddr Struct Reference . . . . . 581
- IxEthDBPortDefinition Struct Reference . . . . . 582
- IxEthEthObjStats Struct Reference . . . . . 582
- IxFpCodeletFastPathConnection Struct Reference . . . . . 583
- IxFpCodeletTemplate Struct Reference . . . . . 583
- IxHssAccCodeletStats Struct Reference . . . . . 584
- IxHssAccConfigParams Struct Reference . . . . . 584
- IxHssAccPktHdlcFraming Struct Reference . . . . . 585
- IxHssAccPortConfig Struct Reference . . . . . 585
- IxMbufPool Struct Reference . . . . . 586
- IxNpeA_AtmCellHeader Struct Reference . . . . . 587
- IxNpeA_AtmVcFp Struct Reference . . . . . 587
- IxNpeA_NpePacketDescriptor Struct Reference . . . . . 588
- IxNpeA_RxAtmVc Struct Reference . . . . . 588
- IxNpeA_TxAtmVc Struct Reference . . . . . 589
- IxNpeDlVersionId Struct Reference . . . . . 590
- IxNpeMhMessage Struct Reference . . . . . 590
- ixUARTDev Struct Reference . . . . . 590
- ixUARTStats Struct Reference . . . . . 591
- LearningIpFlow Struct Reference . . . . . 591
- PacketisedStats Struct Reference . . . . . 591

# A.3 Intel® IXP42X product line Access-Layer APIs

## A.3.1 ADSL Driver (IxAdsl) API

The section documents the public API for the Intel® IXP42X product line ADSL Driver.

### A.3.1.1 Defines

- #define IX_ADSL_SUCCESS 0
  Line was opened and is in 'showtime state'.
- #define IX_ADSL_OPEN_FAILED 1
  Line did not open properly.
- #define IX_ADSL_UNSUPPORTED_MODE 2
  Illegal mode type for OPEN state.
- #define IX_ADSL_COMMS_FAILURE 3
  Serial read/write failure for Serial CTRLE only.
- #define IX_ADSL_CLOSE_FAILED 4
  Line failed to close properly.
- #define IX_ADSL_ALREADY_DOWN 5
  Line is already down.
- #define IF_TYPE_ADSL_PHYS 94
  Physical interface channel type used for ADSL.
- #define IF_TYPE_ADSL_INTLV 124
  Interleave interface channel type used for ADSL.
- #define IF_TYPE_ADSL_FAST 125
  The fast interface channel type used for ADSL.
- #define IX_ADSL_SELFTEST_OK 0
  Selftest passed.
- #define IX_ADSL_SELFTEST_FAILED 1
  Selftest failed.
- #define IX_ADSL_SELFTEST_UNIMPLEMENTED 2
  Selftest un-implemented.
- #define IX_ADSL_LINE_UNKNOWN_STATE (0)
  ADSL Line unknown state.
- #define IX_ADSL_LINE_UP (1)
  The line is in show time state.
- #define IX_ADSL_LINE_WAIT_FOR_ACTIVATING (2)
  The line is waiting for the peer to activate.

- #define IX_ADSL_LINE_ACTIVATING (3)
  The line is negotiating with its peer.
- #define IX_ADSL_LINE_DOWN (4)
  The line is down.

### A.3.1.2 Typedefs

- typedef enum t_ADSL_LINE_TYPE t_ADSL_LINE_TYPE
  Used to indicate the type of ADSL line type used.
- typedef enum t_ADSL_LINE_STATE t_ADSL_LINE_STATE
  Used to indicate the ADSL line state either UP or DOWN.
- typedef enum t_ADSL_PHY_TYPE t_ADSL_PHY_TYPE
  Used to indicate whether the ADSL physical is either CO or CPE.
- typedef void(* IX_ADSL_LINE_CHANGE_FNC )(unsigned int lineNum, t_ADSL_LINE_STATE lineState)
  Type definition callback function to indicate of the changes on the line state.
- typedef void(* IxAdslDriverTrapCallbackPtr_t )(const IxAdslManagerTraps entity, UINT32 ifIndex)
  Type definition function pointer to a callback in the Snmp component.

### A.3.1.3 Functions

- INT32 ixAdslDriverInfoGet (unsigned int lineNum, const IxAdslManagerInfo entity, UINT32 ifIndex, ManagerInfoData *dataBufferToWriteInto)
  Retrieves information for the given ADSL line.
- INT32 ixAdslDriverInfoSet (unsigned int lineNum, const IxAdslManagerInfo entity, UINT32 ifIndex, ManagerInfoData *dataBufferToRead)
  Sets operation for the given ADSL line.
- INT32 ixAdslDriverTrapBind (IxAdslDriverTrapCallbackPtr_t funcPtr)
  The Trap interface is used to prevent circular dependencies.
- unsigned int IxADSL_CPE_Open (unsigned int lineNum, enum t_ADSL_LINE_TYPE)
  Opens the given ADSL line in ATU-R mode.
- unsigned int IxADSL_CPE_LineClose (unsigned int lineNum)
  Closes a previously opened ATU-R ADSL line.
- unsigned int IxADSL_CO_Open (unsigned int lineNum, enum t_ADSL_LINE_TYPE)
  Opens the given ADSL line in ATU-C mode.
- unsigned int IxADSL_CO_LineClose (unsigned int lineNum)
  Closes a previously opened ATU-C ADSL line. The line will be put in the idle state.
- unsigned int IxADSL_LineOpen (unsigned int lineNum, t_ADSL_PHY_TYPE phyType, unsigned char *initFirmware, unsigned int initFirmwarewLen, unsigned char *phyFirmware, unsigned int phyFirmwareLen, t_ADSL_LINE_TYPE lineType)
  Opens the given ADSL line in the specified mode.

- unsigned int IxADSL_LineStateReg (unsigned int lineNum, IX_ADSL_LINE_CHANGE_FNC callback_fn)
- unsigned int IxADSL_GetLineState (unsigned int lineNum)
  Returns the current state of the given ADSL line.
- unsigned int IxADSL_GetLineRateUpstream (unsigned int lineNum)
  Return the current upstream line speed of the given ADSL line.
- unsigned int IxADSL_GetLineRateDownstream (unsigned int lineNum)
  Return the current downstream line speed of the given ADSL line.
- void IxADSL_EnableDyingGasp (unsigned int lineNum)
  Enables the function that informs ATU-C when condition that leads to shutdown of the given Adsl line has been detected.
- void IxADSL_setVendorCode (unsigned int lineNum, unsigned char itu_vendorid_countrycode, unsigned char itu_vendorid_vendorcode_1, unsigned char itu_vendorid_vendorcode_2, unsigned char itu_vendorid_vendorcode_3, unsigned char itu_vendorid_vendorcode_4, unsigned char itu_vendorid_vendorspecific_1, unsigned char itu_vendorid_vendorspecific_2)
  Sets the vendor specific bytes in the given ADSL line.
- unsigned int IxADSL_SelfTest (unsigned int lineNum)
  Runs the Modem self test.
- void IxADSL_DumpStats (unsigned int lineNum)
  Shows the current statistics associated with the given ADSL Line.
- void ixAdslShow (unsigned int lineNum)
  Shows the current statistics associated with the given ADSL Line.

### A.3.1.4 Detailed Description

This module is the public API for the Intel® IXP42X product line ADSL Driver.

### A.3.1.5 Typedef Documentation

#### A.3.1.5.1 typedef enum t_ADSL_LINE_STATE t_ADSL_LINE_STATE

Used to indicate the ADSL line state either UP or DOWN.

```
typedef enum t_ADSL_LINE_STATE {
 IX_ADSL_LINE_UP, - ADSL line state is up
 IX_ADSL_LINE_DOWN, - ADSL line state is down
 IX_ADSL_INVALID_STATE - ADSL line state is invalid
 } t_ADSL_LINE_STATE;
```

#### A.3.1.5.2 typedef enum t_ADSL_LINE_TYPE t_ADSL_LINE_TYPE

Used to indicate the type of ADSL line type used.

```
typedef enum t_ADSL_LINE_TYPE {
 ADSL_NORMAL = 0, - Normal line type
 ADSL_GLITE = 1, - GLite line type
```

```
ADSL_DMT = 2, - DMT line type
ADSL_ANSI = 3, - ANSI line type
ADSL_MULTIMODE = 4, - Multimode line type
ADSL_INVALID_MODE = 5 - Invalid line type
}t_ADSL_LINE_TYPE;
```

#### A.3.1.5.3 typedef enum t_ADSL_PHY_TYPE t_ADSL_PHY_TYPE

Used to indicate whether the ADSL physical is either CO or CPE.

```
typedef enum t_ADSL_PHY_TYPE {
 IX_ADSL_CO, - ADSL type is CO
 IX_ADSL_CPE - ADSL type is CPE
 } t_ADSL_PHY_TYPE;
```

### A.3.1.6 Function Documentation

#### A.3.1.6.1 IxADSL_CO_LineClose (unsigned int *lineNum*)

Closes a previously opened ATU-C ADSL line. The line is put in the idle state.

**Parameters**

int lineNum — Shows which ADSL module is being used.

**Notes**

Blocking — Non-blocking.

Impacts On Global Data — Sets the admin state. Notifies a callback routine.

Pre-Conditions — Code should only be called from task level.

Post-Conditions — No cleanup after this call is required.

**Returns**

- IX_ADSL_SUCCESS — Line was closed.
- IX_ADSL_CLOSE_FAILED — Line did not close properly.
- IX_ADSL_ALREADY_DOWN — Line was not open before close.

#### A.3.1.6.2 IxADSL_CO_Open (unsigned int *lineNum*, enum *t_ADSL_LINE_TYPE*)

Opens the given ADSL line in ATU-C mode.

**Parameters**

int lineNum — Shows which ADSL module is being used.

enum t_ADSL_LINE_TYPE — Indicates type of ADSL to be opened.

### Notes

Opens the given ADSL line in ATU-C mode and puts it in “show-time” state, (i.e., available to carry user data). There are a substantial number of line parameters that would normally be configurable as an ATU-C device. The IxADSL_CO_Open API does not allow for any control of these parameters. It will open the line with a default recommended set for demonstration usage. The user may be given access to the ADSL driver layer should our device be targeted towards a DSLAM type applications.

Blocking — This call may block for several seconds while the link is established.

Impacts On Global Data — Sets the admin state.

Pre-Conditions — The code must only be called once the operating system is running, i.e. do not call as part of the hardware init as this code requires base services. The code should only be called from task level.

Post-Conditions — No cleanup after this call is required.

Exceptions — None.

### Returns

- IX_ADSL_SUCCESS — Line was opened and is in 'showtime' state.
- IX_ADSL_OPEN_FAILED — Line did not open properly.
- IX_ADSL_UNSUPPORTED_MODE — Illegal mode type for OPEN.
- IxADSL_CPE_LineClose (unsigned int *lineNum*) — This closes a previously opened ATU-R ADSL line.

### Parameters

***int lineNum*** — Shows which ADSL module is being used.

If line-close is successful notify the callback that the line is now closed. The line will be put in the idle state.

### Notes

Blocking — This call may block for several seconds, as the negotiation to bring down the line takes place.

Impacts On Global Data — Sets the admin state.

Pre-Conditions — The code must only be called once the operating system is running, i.e. do not call as part of the hardware init as this code requires base services. The code should only be called from task level.

Post-Conditions — No cleanup after this call is required.

Exceptions — None.

#### Returns

- IX_ADSL_SUCCESS — Line was closed
- IX_ADSL_CLOSE_FAILED — Line did not close properly.
- IX_ADSL_ALREADY_DOWN — Line was not open before close.

### A.3.1.6.3 IxADSL_CPE_Open (unsigned int *lineNum*, enum *t_ADSL_LINE_TYPE*)

Opens the given ADSL line in ATU-R mode.

#### Parameters

***int lineNum*** parameter showing which ADSL module is being used.

***enum t_ADSL_LINE_TYPE*** Indicates type of ADSL to be opened.

Opens the given ADSL line in ATU-R mode and puts it in the 'showtime' state (i.e. available to carry user data). Note that when the CPE is asked to open its line in NORMAL mode and if the CO is configured for G.LITE, then the CPE will come up in G.LITE mode.

#### Notes

Blocking — This call may block for several seconds while the link is established.

Impacts On Global Data — Sets the admin state.

Pre-Conditions — The code must only be called once the operating system is running (i.e. do not call as part of the hardware init as this code requires base services). The code should only be called from task level.

Post-Conditions — No cleanup after this call is required.

Exceptions — None.

#### Returns

- IX_ADSL_SUCCESS — Line was opened and is in 'showtime' state.
- IX_ADSL_OPEN_FAILED — Line did not open properly.
- IX_ADSL_UNSUPPORTED_MODE — Illegal mode type for OPEN.

### A.3.1.6.4 IxADSL_DumpStats (unsigned int *lineNum*)

This function will show the current statistics associated with the given ADSL Line.

The functionality is the same as ixAdslShow.

### A.3.1.6.5 IxADSL_EnableDyingGasp (unsigned int *lineNum*)

Enables the function that informs ATU-C when condition that leads to shutdown of the given ADSL line has been detected.

#### Parameters

***int lineNum*** parameter showing which ADSL module is being used.

#### Notes

Blocking — This call is a non-blocking.

Impacts On Global Data — None.

Pre-Conditions — Task level calls only. Only Available in ATU-R.

Post-Conditions — None.

Exceptions — None.

#### Returns

None.

### A.3.1.6.6 IxADSL_GetLineRateDownstream (unsigned int *lineNum*)

Return the current downstream line speed of the given ADSL line.

#### Parameters

***int lineNum*** parameter showing which ADSL module is being used.

This is the speed of the line from the ATU-C devices assuming that the device is configured as an ATU-R device, i.e. the Rx speed of this device, and, and visa versa when the device is configured as an ATU-C.

#### Notes

Blocking — This call is a non-blocking.

Impacts On Global Data — None.

Pre-Conditions — Task level calls only.

Post-Conditions — None.

Exceptions — None.

#### Returns

- Integer — bit rate in kbits/second. N.B. Returns zero if the line is not in 'showtime' state.

### A.3.1.6.7 IxADSL_GetLineRateUpstream (unsigned int *lineNum*)

Return the current upstream line speed of the given ADSL line.

#### Parameters

***int lineNum*** parameter showing which ADSL module is being used.

This represents the speed of the line to the ATU-C devices assuming that the device is configured as an ATU-R device, i.e. the Tx speed from this device, and visa versa when the device is configured as an ATU-C.

#### Notes

Blocking — This call is a non-blocking.

Impacts On Global Data — None.

Pre-Conditions — Task level calls only.

Post-Conditions — None.

Exceptions — None.

#### Returns

- Integer — bit rate in kbits/second. N.B. Returns zero if the line is not in 'showtime' state.

### A.3.1.6.8 IxADSL_GetLineState (unsigned int *lineNum*)

Returns the current state of the given ADSL line.

#### Parameters

***int lineNum*** Parameter showing which ADSL module is being used.

#### Notes

Blocking — This call is a non-blocking.

Impacts On Global Data — None.

Pre-Conditions — Task level calls only.

Post-Conditions — None.

Exceptions — None.

#### Returns

- IX_ADSL_LINE_UP — The line is in show time state.
- IX_ADSL_LINE_WAIT_FOR_ACTIVATING — The line is waiting for the peer to activate.
- IX_ADSL_LINE_ACTIVATING — The line is negotiating with its peer.
- IX_ADSL_LINE_DOWN — The line is idle.

### A.3.1.6.9 unsigned int IxADSL_LineOpen (..)

unsigned int IxADSL_LineOpen (unsigned int lineNum, t_ADSL_PHY_TYPE phyType, unsigned char * initFirmware, unsigned int initFirmwarewLen, unsigned char * phyFirmware, unsigned int phyFirmwareLen, t_ADSL_LINE_TYPE lineType)

Opens the given ADSL line in the specified mode.

#### Parameters

***int lineNum*** — Shows which ADSL module is being used.

***t_ADSL_PHY_TYPE phyType*** — Type of Phy used CO or CPE.

***unsigned char *initFirmware*** — Pointer to an array containing firmware to initialize Alcatel* PHY.

***unsigned int initFwLen*** — Indicates the length in bytes of Alcatel PHY initialization firmware.

***unsigned char *phyFirmware*** — Pointer to an array containing the CO or CPE firmware as indicated by the phyType parameter.

***unsigned int phyFwLen*** — Indicates length in bytes of the main CO or CPE firmware load to be downloaded onto the Alcatel PHY.

***t_ADSL_LINE_TYPE lineType*** — Indicates type of ADSL to be opened.

Opens the given ADSL line in the specified mode and puts it in the 'showtime' state, i.e. available to carry user data. Called by IxADSL_CO_Open and IxADSL_CPE_Open.

#### Notes

Blocking — This call may block for several seconds while the link is established.

Impacts On Global Data — Sets the admin state.

Pre-Conditions — The code must only be called once the operating system is running, i.e. do not call as part of the hardware init as this code requires base services. The code should only be called from task level.

Post-Conditions — No cleanup after this call is required.

Exceptions — None.

#### Returns

- IX_ADSL_SUCCESS — Line was opened and is in 'showtime' state.
- IX_ADSL_OPEN_FAILED — Line did not open properly.
- IX_ADSL_UNSUPPORTED_MODE — Illegal mode type for OPEN.

### A.3.1.6.10 IxADSL_SelfTest (unsigned int *lineNum*)

Runs the Modem self test.

#### Parameters

***int lineNum*** parameter showing which ADSL module is being used.

This should only be called when the modem is off-line (line is closed).

#### Notes

Blocking — This call may block for several seconds as the selftest completes.

Impacts On Global Data — None

Pre-Conditions — Task level calls only. If the given ADSL line is open when the call is made, the line shall be forced closed prior to the selftest execution.

Post-Conditions — This shall leave the line in the closed state.

Exceptions — None.

#### Returns

- IX_ADSL_SELFTEST_OK — Selftest passed.
- IX_ADSL_SELFTEST_FAILED — Selftest failed.
- IX_ADSL_SELFTEST_UNIMPLEMENTED — Selftest unimplemented.

### A.3.1.6.11 void IxADSL_setVendorCode (..)

void IxADSL_setVendorCode (unsigned int lineNum, unsigned char itu_vendorid_countrycode, unsigned char itu_vendorid_vendorcode_1, unsigned char itu_vendorid_vendorcode_2, unsigned char itu_vendorid_vendorcode_3, unsigned char itu_vendorid_vendorcode_4, unsigned char itu_vendorid_vendorspecific_1, unsigned char itu_vendorid_vendorspecific_2)

Set the vendor specific bytes in the given ADSL line.

The vendor ID must be set before the line is open, if not the default vendor ID shall be sent to the peer modem upon request.

#### Parameters

***int lineNum*** — parameter showing which ADSL module is being used.

***unsigned char itu_vendorid_countrycode*** — Vendor country code that are predefined in standards.

***unsigned char itu_vendorid_vendorcode_1*** — Vendor code 1 that are predefined in standards.

***unsigned char itu_vendorid_vendorcode_2*** — Vendor code 2 that are predefined in standards.

***unsigned char itu_vendorid_vendorcode_3*** — Vendor code 3 that are predefined in standards.

***unsigned char itu_vendorid_vendorcode_4*** — Vendor code 4 that are predefined in standards.

***unsigned char itu_vendorid_vendorspecific_1*** — Vendor specific 1 that are predefined in standards.

***unsigned*** char itu_vendorid_vendorspecific_12 Vendor specific 2 that are predefined in standards.

#### Notes

Vendor specific values are taken by the PHY and linked together to form a single code.

The vendor ID must be set before the line is open, if not the default vendor ID shall be sent to the peer modem upon request.

Blocking — This call is a non-blocking.

Impacts On Global Data — Sets an internal ADSL global data structure. This shall be used for all subsequent ADSL line open commands.

Pre-Conditions — Task level calls only.

Post-Conditions — None.

Exceptions — None.

#### Returns

None

### A.3.1.6.12 ixAdslDriverInfoGet (..)

ixAdslDriverInfoGet (unsigned int *lineNum*, const IxAdslManagerInfo *entity*, UINT32 *ifIndex*, ManagerInfoData * *dataBufferToWriteInto*)

Retrieve information for the given ADSL line.

This function returns to its caller the information for the given ADSL line associated with the management entity in the form of a ManagerInfoData structure.

#### Parameters

***int lineNum*** — Parameter showing which ADSL module is being used.

***IxAdslManagerInfo entity*** — Is the enum value that corresponds to the MIB entity being set.

***UINT32 index*** — A variable containing the required index for the entity.

***ManagerInfoData *dataBuffer*** — A buffer to fill with the pertinent data.

#### Returns

- IX_SUCCESS — request completed successfully.
- IX_FAILED — request failed for unspecified reason.
- IX_MGRINF_INVALID — request invalid, unspecified reason.
- IX_MGRINF_NOSUCHINDEX — instance of index given doesn't exist.
- IX_MGRINF_NOSUCHENTITY — entity passed in does not exist.
- IX_MGRINF_INDEXOUTOFRANGE — index passed not within valid range.
- IX_MGRINF_NOMORE — index passed is the last valid index.
- IX_MGRINF_NOTSUPPORTED — state to be supported in future.

### A.3.1.6.13 ixAdslDriverInfoSet (..)

ixAdslDriverInfoSet (unsigned int lineNum, const IxAdslManagerInfo entity, UINT32 ifIndex, ManagerInfoData * dataBufferToRead)

Set operation for the given ADSL line.

The set operation changes, for the given ADSL line, the contents of the component's local store of information, pertaining to the specified entity, using the input dataBuffer contents.

#### Parameters

***int lineNum*** — Parameter showing which ADSL module is being used.

***IxAdslManagerInfo entity*** — Is the enum value that corresponds to the MIB entity being set.

***UINT32 index*** — A variable containing the required index for the entity.

***ManagerInfoData *dataBuffer*** — A buffer to fill with the pertinent data.

#### Returns

- IX_SUCCESS — request completed successfully.
- IX_FAILED — request failed for unspecified reason.
- IX_MGRINF_INVALID — request invalid, unspecified reason.
- IX_MGRINF_NOSUCHINDEX — instance of index given doesn't exist.
- IX_MGRINF_NOSUCHENTITY — entity passed in does not exist.
- IX_MGRINF_INDEXOUTOFRANGE — index passed not within valid range.
- IX_MGRINF_NOMORE — index passed is the last valid index.
- IX_MGRINF_NOTSUPPORTED — state to be supported in future.

### A.3.1.6.14 ixAdslDriverTrapBind (..)

ixAdslDriverTrapBind (IxAdslDriverTrapCallbackPtr_t funcPtr)

The Trap interface is used to prevent circular dependencies.

ADSL Management registers the function pointer as a subscriber to all events. IxAdslDriverTrapBind stores this function pointer internally and invokes it when an event listed in IxAdslManagerTraps occurs to notify the SNMP agent.

#### Parameters

***IxAdslDriverTrapCallbackPtr_t funcPtr*** — This function takes a function pointer to a callback in the Snmp component. Where the prototype for the function pointer IxAdslTrapCallbackPtr_t

#### Returns

- IX_SUCCESS — Request completed successfully.
- IX_FAILED — Request failed for unspecified reason.

### A.3.1.6.15 ixAdslShow (..)

ixAdslShow (unsigned int lineNum)

This function will show the current statistics associated with the given ADSL Line.

#### Parameters

***int lineNum*** parameter showing which ADSL module is being used.

**Notes**

Blocking — This call is non-blocking.

Impacts On Global Data — None.

Pre-Conditions — Task level calls only.

Post-Conditions — None.

Exceptions — None.

**Returns**

None.

## A.3.2 Assertion Macros (IxAssert) API

Assertion support.

### A.3.2.1 Defines

#define IX_ASSERT(c) assert(c)

Assert macro, assert the condition is true. This will not be compiled out. N.B. will result in a system crash if it is false. <ToDo> implement, assert() is called as an interim <ToDo> Provide function to override behavior of assert.

#define IX_ENSURE(c, str) if (!c) printf(str "\n")

Ensure macro, ensure the condition is true. This will be conditionally compiled out and may be used for unit/integration test purposes. <ToDo> implement.

### A.3.2.2 Detailed Description

Assertion support.

## A.3.3 ATM Driver Access (IxAtmdAcc) API

The public API for the Intel® IXP42X product line ATM Driver Data component.

### A.3.3.1 Defines

- #define IX_ATMDACC_WARNING 2
  Warning return code.
- #define IX_ATMDACC_BUSY 3
  Busy return code.
- #define IX_ATMDACC_RESOURCES_STILL_ALLOCATED 4
  Disconnect return code.
- #define IX_ATMDACC_DEFAULT_REPLENISH_COUNT 0
  Default resources usage for RxVcFree replenish mechanism.

- #define IX_ATMDACC_OAM_TX_VPI 0

  The reserved value used for the dedicated OAM Tx connection. This “well known” value is used by atmdAcc and its clients to discriminate the OAM channel, and should be chosen so that it does not coincide with the VPI value used in an AAL0/AAL5 connection. Any attempt to connect a service type other than OAM on this VPI will fail.

- #define IX_ATMDACC_OAM_TX_VCI 0

  The reserved value used for the dedicated OAM Tx connection. This “well known” value is used by atmdAcc and its clients to discriminate the OAM channel, and should be chosen so that it does not coincide with the VCI value used in an AAL0/AAL5 connection. Any attempt to connect a service type other than OAM on this VCI will fail.

#define IX_ATMDACC_OAM_RX_PORT IX_UTOPIA_PORT_0

The reserved dummy PORT used for all dedicated OAM Rx connections. Note that this is not a real port but must have a value that lies within the valid range of port values.

#define IX_ATMDACC_OAM_RX_VPI 0

The reserved value used for the dedicated OAM Rx connection. This value should be chosen so that it does not coincide with the VPI value used in an AAL0/AAL5 connection. Any attempt to connect a service type other than OAM on this VPI will fail.

#define IX_ATMDACC_OAM_RX_VCI 0

The reserved value used for the dedicated OAM Rx connection. This value should be chosen so that it does not coincide with the VCI value used in an AAL0/AAL5 connection. Any attempt to connect a service type other than OAM on this VCI will fail.

### A.3.3.2 Typedefs

- typedef unsigned int IxAtmdAccUserId

  User-supplied Id.

- typedef void(* IxAtmdAccRxVcRxCallback )(IxAtmLogicalPort port, IxAtmdAccUserId userId, IxAtmdAccPduStatus status, IxAtmdAccClpStatus clp, IX_MBUF *mbufPtr)

  Rx callback prototype.

- typedef void(* IxAtmdAccRxVcFreeLowCallback )(IxAtmdAccUserId userId)

  Callback prototype for free buffer level is low.

- typedef void(* IxAtmdAccTxVcBufferReturnCallback )(IxAtmdAccUserId userId, IX_MBUF *mbufPtr)

  Buffer callback prototype.

### A.3.3.3 Enumerations

- enum IxAtmdAccPduStatus { IX_ATMDACC_AAL0_VALID = 0, IX_ATMDACC_OAM_VALID, IX_ATMDACC_AAL2_VALID, IX_ATMDACC_AAL5_VALID, IX_ATMDACC_AAL5_PARTIAL, IX_ATMDACC_AAL5_CRC_ERROR, IX_ATMDACC_MBUF_RETURN }

  IxAtmdAcc Pdu status.

- enum IxAtmdAccAalType { IX_ATMDACC_AAL5, IX_ATMDACC_AAL2, IX_ATMDACC_AAL0_48, IX_ATMDACC_AAL0_52, IX_ATMDACC_OAM, IX_ATMDACC_MAX_SERVICE_TYPE }

  IxAtmdAcc AAL Service Type.

- enum IxAtmdAccClpStatus { IX_ATMDACC_CLP_NOT_SET = 0, IX_ATMDACC_CLP_SET = 1 }

  IxAtmdAcc CLP indication.

### A.3.3.4 Functions

- PUBLIC IX_STATUS ixAtmdAccInit (void)

  Initialize the IxAtmdAcc Component.

- PUBLIC void ixAtmdAccShow (void)

  Show IxAtmdAcc configuration on a per port basis.

- PUBLIC void ixAtmdAccStatsShow (void)

  Show all IxAtmdAcc stats.

- PUBLIC void ixAtmdAccStatsReset (void)

  Reset all IxAtmdAcc stats.

- PUBLIC IX_STATUS ixAtmdAccRxVcConnect (IxAtmLogicalPort port, unsigned int vpi, unsigned int vci, IxAtmdAccAalType aalServiceType, IxAtmRxQueueId rxQueueId, IxAtmdAccUserId userCallbackId, IxAtmdAccRxVcRxCallback rxCallback, unsigned int minimumReplenishCount, IxAtmConnId *connIdPtr, IxAtmNpeRxVcId *npeVcIdPtr)

  Connect to a Aal Pdu receive service for a particular port/vpi/vci, and service type.

- PUBLIC IX_STATUS ixAtmdAccRxVcFreeReplenish (IxAtmConnId connId, IX_MBUF *mbufPtr)

  Provide free mbufs for data reception on a connection.

- PUBLIC IX_STATUS ixAtmdAccRxVcFreeLowCallbackRegister (IxAtmConnId connId, unsigned int numberOfMbufs, IxAtmdAccRxVcFreeLowCallback callback)

  Configure the RX Free threshold value and register a callback to handle threshold notifications.

- PUBLIC IX_STATUS ixAtmdAccRxVcFreeEntriesQuery (IxAtmConnId connId, unsigned int *numberOfMbufsPtr)

  Get the number of rx mbufs the system can accept to replenish the rx reception mechanism on a particular channel.

- PUBLIC IX_STATUS ixAtmdAccRxVcEnable (IxAtmConnId connId)

  Start the RX service on a VC.

- PUBLIC IX_STATUS ixAtmdAccRxVcDisable (IxAtmConnId connId)

  Stop the RX service on a VC.

- PUBLIC IX_STATUS ixAtmdAccRxVcTryDisconnect (IxAtmConnId connId)

  Disconnect a VC from the RX service.

- PUBLIC IX_STATUS ixAtmdAccTxVcConnect (IxAtmLogicalPort port, unsigned int vpi, unsigned int vci, IxAtmdAccAalType aalServiceType, IxAtmdAccUserId userId, IxAtmdAccTxVcBufferReturnCallback bufferFreeCallback, IxAtmConnId *connIdPtr)

  Connect to a Aal Pdu transmit service for a particular port/vpi/vci and service type.

- PUBLIC IX_STATUS ixAtmdAccTxVcPduSubmit (IxAtmConnId connId, IX_MBUF *mbufPtr, IxAtmdAccClpStatus clp, unsigned int numberOfCells)

  Submit a Pdu for transmission on connection.

- PUBLIC IX_STATUS ixAtmdAccTxVcTryDisconnect (IxAtmConnId connId)

  Disconnect from a Aal Pdu transmit service for a particular port/vpi/vci.

### A.3.3.5 Detailed Description

The public API for the Intel® IXP42X product line ATM Driver Data component.

IxAtmdAcc is the low level interface by which AAL0/AAL5 and OAM data gets transmitted to, and received from the UTOPIA bus.

For AAL0/AAL5 services transmit and receive connections may be established independently for unique combinations of port, VPI, and VCI.

Two AAL0 services supporting 48 or 52 byte cell data are provided. Submitted AAL0 PDUs must be a multiple of the cell data size (48/52). AAL0_52 is a raw cell service the client must format the PDU with an ATM cell header (excluding HEC) at the start of each cell, note that AtmdAcc does not validate the cell headers in a submitted PDU.

OAM cells cannot be received over the AAL0 service but instead are received over a dedicated OAM service.

For the OAM service an “OAM Tx channel” may be enabled for a port by establishing a single dedicated OAM Tx connection on that port. A single “OAM Rx channel” for all ports may be enabled by establishing a dedicated OAM Rx connection.

The OAM service allows buffers containing 52 byte OAM F4/F5 cells to be transmitted and received over the dedicated OAM channels. HEC is appended/removed, and CRC-10 performed by the NPE. The OAM service offered by AtmdAcc is a raw cell transport service. It is assumed that ITU I.610 procedures that make use of this service are implemented above AtmdAcc.

Note that the dedicated OAM connections are established on reserved VPI, VCI, and (in the case of Rx) port values defined below. These values are used purely to discriminate the dedicated OAM channels and do not identify a particular OAM F4/F5 flow. F4/F5 flows may be realized for particular VPI/VCIs by manipulating the VPI,VCI fields of the ATM cell headers of cells in the buffers passed to AtmdAcc. Note that AtmdAcc does not validate the cell headers in a submitted OAM PDU.

This part is related to the User datapath processing

### A.3.3.6 Define Documentation

#### A.3.3.6.1 #define IX_ATMDACC_BUSY 3

Busy return code.

This constant is used to tell IxAtmDAcc user that the request is correct, but cannot be processed because the IxAtmAcc resources are already used. The user has to retry its request later

#### A.3.3.6.2 #define IX_ATMDACC_DEFAULT_REPLENISH_COUNT 0

Default resources usage for RxVcFree replenish mechanism.

This constant is used to tell IxAtmDAcc to allocate and use the minimum of resources for Rx free replenish.

**See also:**

ixAtmdAccRxVcConnect.

#### A.3.3.6.3 #define IX_ATMDACC_OAM_TX_VPI 0

The reserved value used for the dedicated OAM Tx connection. This “well known” value is used by atmdAcc and its clients to discriminate the OAM channel, and should be chosen so that it does not coincide with the VPI value used in an AAL0/AAL5 connection. Any attempt to connect a service type other than OAM on this VPI will fail.

#### A.3.3.6.4 #define IX_ATMDACC_RESOURCES_STILL_ALLOCATED 4

Disconnect return code.

This constant is used to tell IxAtmDAcc user that the disconnect functions are not complete because the resources used by the driver are not yet released. The user has to retry the disconnect call later.

#### A.3.3.6.5 #define IX_ATMDACC_WARNING 2

Warning return code.

This constant is used to tell IxAtmDAcc user about a special case.

### A.3.3.7 Typedef Documentation

#### A.3.3.7.1 typedef void(..)

typedef void(* IxAtmdAccRxVcFreeLowCallback)(IxAtmdAccUserId userId)

Callback prototype for free buffer level is low.

IxAtmdAccRxVcFreeLowCallback is the prototype of the user function which get called on a per-VC basis, when more mbufs are needed to continue the ATM data reception. This function is likely to supply more available mbufs by one or many calls to the replenish function ixAtmdAccRxVcFreeReplenish()

This function is called when the number of available buffers for reception is going under the threshold level as defined in ixAtmdAccRxVcFreeLowCallbackRegister()

This function is called inside an Qmgr dispatch context. No system resource or interrupt-unsafe feature should be used inside this callback.

**See also:**

ixAtmdAccRxVcFreeLowCallbackRegister, IxAtmdAccRxVcFreeLowCallback, ixAtmdAccRxVcFreeReplenish, ixAtmdAccRxVcFreeEntriesQuery, ixAtmdAccRxVcConnect

### Parameters

***userId*** — (in) user Id provided in the call to ixAtmdAccRxVcConnect().

### Returns

None.

## A.3.3.7.2 typedef void(..)

typedef void(* IxAtmdAccRxVcRxCallback)(IxAtmLogicalPort port, IxAtmdAccUserId userId, IxAtmdAccPduStatus status, IxAtmdAccClpStatus clp, IX_MBUF * mbufPtr)

Rx callback prototype.

IxAtmdAccRxVcRxCallback is the prototype of the Rx callback user function called once per PDU to pass a receive Pdu to a user on a particular connection. The callback is likely to push the mbufs to a protocol layer, and recycle the mbufs for a further use.

### Notes

This function is called ONLY in the context of the ixAtmdAccRxDispatch() function

### See also:

ixAtmdAccRxDispatch, ixAtmdAccRxVcConnect

### Parameters

***port*** — (in) the port on which this PDU was received a logical PHY port [*IX_UTOPIA_PORT_0* .. *IX_UTOPIA_MAX_PORTS* - 1]

**userId** — (in) user Id provided in the call to *ixAtmdAccRxVcConnect()*

***status*** — (in) an indication about the PDU validity. In the case of AAL0 the only possible value is AAL0_VALID, in this case the client may optionally determine that an rx timeout occurred by checking if the mbuf is completely or only partially filled, the later case indicating a timeout. In the case of OAM the only possible value is OAM valid. The status is set to *IX_ATMDACC_MBUF_RETURN* when the mbuf is released during a disconnect process.

***clp*** — (in) clp indication for this PDU. For AAL5/AAL0_48 this information is set if the clp bit of any rx cell is set For AAL0-52/OAM the client may inspect the CLP in individual cell headers in the PDU, and this parameter is set to 0.

***mbufPtr*** — (in) depending on the service type a pointer to an mbuf (AAL5/AAL0/OAM) or mbuf chain (AAL5 only), that comprises the complete PDU data.

This parameter is guaranteed not to be a null pointer.

#### A.3.3.7.3 typedef void(..)

typedef void(* IxAtmdAccTxVcBufferReturnCallback)(IxAtmdAccUserId userId, IX_MBUF * mbufPtr)

Buffer callback prototype.

This function is called to relinquish ownership of a transmitted buffer chain to the user.

**Note**

In the case of a chained mbuf the AmtdAcc component can chain many user buffers together and pass ownership to the user in one function call.

**Parameters**

***userId*** — (in) user If provided at registration of this callback.

***mbufPtr*** — (in) a pointer to mbufs or chain of mbufs and is guaranteed not to be a null pointer.

#### A.3.3.7.4 IxAtmdAccUserId

User-supplied Id.

IxAtmdAccUserId is passed through callbacks and allows the IxAtmdAcc user to identify the source of a call back. The range of this user-owned Id is [0...2^32-1)].

The user provides this own Ids on a per-channel basis as a parameter in a call to ixAtmdAccRxVcConnect() or ixAtmdAccRxVcConnect().

**See also:**

ixAtmdAccRxVcConnect, ixAtmdAccTxVcConnect

### A.3.3.8 Enumeration Type Documentation

#### A.3.3.8.1 enum IxAtmdAccAalType

IxAtmdAcc AAL Service Type.

IxAtmdAccAalType defines the type of traffic to run on this VC

**Enumeration Values**

***Ix_atmdacc_aal5*** — ITU-T AAL5.

***IX_ATMDACC_AAL2*** — ITU-T AAL2 **reserved** for future use.

***IX_ATMDACC_AAL0_48*** — AAL0 48 byte payloads (cell header is added by NPE).

***IX_ATMDACC_AAL0_52*** — AAL0 52 byte cell data (HEC is added by NPE).

***IX_ATMDACC_OAM*** — OAM cell transport service (HEC is added by NPE).

***IX_ATMDACC_MAX_SERVICE_TYPE*** — not a service, used for parameter validation.

#### A.3.3.8.2 enum IxAtmdAccClpStatus

IxAtmdAcc CLP indication.

IxAtmdAccClpStatus defines the CLP status of the current PDU.

**Enumeration Values**

***Ix_atmdacc_clp_not_set*** — CLP indication is not set.

***IX_ATMDACC_CLP_SET*** — CLP indication is set.

#### A.3.3.8.3 enum IxAtmdAccPduStatus

IxAtmdAcc Pdu status.

IxAtmdAccPduStatus is used during a RX operation to indicate the status of the received PDU.

**Enumeration Values**

***Ix_atmdacc_aal0_valid*** — aal0 pdu.

***IX_ATMDACC_OAM_VALID*** — OAM pdu.

***IX_ATMDACC_AAL2_VALID*** — aal2 pdu **reserved** for future use.

***IX_ATMDACC_AAL5_VALID*** — aal5 pdu complete and trailer is valid.

***IX_ATMDACC_AAL5_PARTIAL*** — aal5 pdu not complete, trailer is missing.

***IX_ATMDACC_AAL5_CRC_ERROR*** — aal5 pdu not complete, crc error/length error.

***IX_ATMDACC_MBUF_RETURN*** — empty buffer returned to the user.

### A.3.3.9 Function Documentation

#### A.3.3.9.1 ixAtmdAccInit (void)

Initialize the IxAtmdAcc Component.

This function initializes the IxAtmdAcc component. This function shall be called before any other function of the API. Its role is to initialize all internal resources of the IxAtmdAcc component.

The ixQmgr component needs to be initialized prior the use of ixAtmdAccInit().

**Parameters**

***none*** — Failing to initialize the IxAtmdAcc API before any use of it will result in a failed status.

**Returns**

- IX_SUCCESS initialization is complete
- IX_FAIL unable to process this request either because this IxAtmdAcc is already initialized or some unspecified error has occurred.

### A.3.3.9.2 ixAtmdAccRxVcConnect (..)

ixAtmdAccRxVcConnect (IxAtmLogicalPort port, unsigned int vpi, unsigned int vci, IxAtmdAccAalType aalServiceType, IxAtmRxQueueId rxQueueId, IxAtmdAccUserId userCallbackId, IxAtmdAccRxVcRxCallback rxCallback, unsigned int minimumReplenishCount, IxAtmConnId * connIdPtr, IxAtmNpeRxVcId * npeVcIdPtr)

Connect to a Aal Pdu receive service for a particular port/vpi/vci, and service type.

This function allows a user to connect to an Aal5/Aal0/OAM Pdu receive service for a particular port/vpi/vci. It registers the callback and allocates internal resources and a Connection Id to be used in further API calls related to this VCC.

The function will setup VC receive service on the specified rx queue.

This function is blocking and makes use internal locks, and hence should not be called from an interrupt context.

On return from ixAtmdAccRxVcConnect() with a failure status, the connection Id parameter is unspecified. Its value cannot be used. A connId is the reference by which IxAtmdAcc refers to a connected VC. This identifier is the result of a successful call to a connect function. This identifier is invalid after a successful call to a disconnect function.

Calling this function for the same combination of Vpi, Vci and more than once without calling ixAtmdAccRxVcTryDisconnect() will result in a failure status.

If this function returns success the user should supply receive buffers by calling ixAtmdAccRxVcFreeReplenish() and then call ixAtmdAccRxVcEnable() to begin receiving pdus.

There is a choice of two receive Qs on which the VC pdus could be receive. The user must associate the VC with one of these. Essentially having two qs allows more flexible system configuration such as have high priority traffic on one q (e.g. voice) and low priority traffic on the other (e.g. data). The high priority Q could be serviced in preference to the low priority Q. One queue may be configured to be serviced as soon as there is traffic, the other queue may be configured to be serviced by a polling mechanism running at idle time.

Two AAL0 services supporting 48 or 52 byte cell data are provided. Received AAL0 PDUs will be be a multiple of the cell data size (48/52). AAL0_52 is a raw cell service and includes an ATM cell header (excluding HEC) at the start of each cell.

A single “OAM Rx channel” for all ports may be enabled by establishing a dedicated OAM Rx connection.

The OAM service allows buffers containing 52 byte OAM F4/F5 cells to be transmitted and received over the dedicated OAM channels. HEC is appended/removed, and CRC-10 performed by the NPE. The OAM service offered by AtmdAcc is a raw cell transport service. It is assumed that ITU I.610 procedures that make use of this service are implemented above AtmdAcc.

Note that the dedicated OAM connections are established on reserved VPI,VCI, and (in the case of Rx) port values. These values are used purely to discriminate the dedicated OAM channels and do not identify a particular OAM F4/F5 flow. F4/F5 flows may be realized for particular VPI/VCIs by manipulating the VPI,VCI fields of the ATM cell headers of cells in the buffers passed to AtmdAcc.

Calling this function prior to enable the port will fail.

### See also:

ixAtmdAccRxDispatch, ixAtmdAccRxVcEnable, ixAtmdAccRxVcDisable, ixAtmdAccRxVcTryDisconnect, ixAtmdAccPortEnable

### Parameters

***port*** (in) VC identification — logical PHY port [*IX_UTOPIA_PORT_0* .. *IX_UTOPIA_MAX_PORTS* - 1]

***vpi*** (in) VC identification — ATM Vpi [0..255] or IX_ATMDACC_OAM_VPI

***vci*** (in) VC identification — ATM Vci [0..65535] or IX_ATMDACC_OAM_VCI

***aalServiceType*** — (in) type of service: AAL5, AAL0_48, AAL0_52, or OAM

***rxQueueId*** (in) this identifies which of two Qs the VC should use when incoming traffic is processed.

***userCallbackId*** (in) user Id used later as a parameter to the supplied rxCallback.

***rxCallback*** (in) function called when mbufs are received. This parameter cannot be a null pointer.

***bufferFreeCallback*** — (in) function to be called to return ownership of buffers to IxAtmdAcc user.

***minimumReplenishCount*** — (in) For AAL5/AAL0 the number of free mbufs to be used with this channel. Use a high number when the expected traffic rate on this channel is high, or when the user's mbufs are small, or when the RxVcFreeLow Notification has to be invoked less often. When this value is IX_ATMDACC_DEFAULT_REPLENISH_COUNT, the minimum of resources will be used. Depending on traffic rate, pdu size and mbuf size, rxfree queue size, polling/interrupt rate, this value may require to be replaced by a different value in the range 1-128 For OAM the rxFree queue size is fixed by atmdAcc and this parameter is ignored.

***connIdPtr*** — (out) pointer to a connection Id This parameter cannot be a null pointer.

***npeVcIdPtr*** — (out) pointer to an npe Vc Id This parameter cannot be a null pointer.

### Returns

- IX_SUCCESS — successful call to IxAtmdAccRxVcConnect
- IX_ATMDACC_BUSY — cannot process this request : no VC is available

### Note

IX_FAIL parameter error, VC already in use, attempt to connect AAL service on reserved OAM VPI/VCI, attempt to connect OAM service on VPI/VCI other than the reserved OAM VPI/VCI, port is not initialized, or some other error occurs during processing.

## A.3.3.9.3 ixAtmdAccRxVcDisable (..)

ixAtmdAccRxVcDisable (IxAtmConnId connId)

Stop the RX service on a VC.

This functions stops the traffic reception for a particular VC connection.

Once invoked, incoming Pdus are discarded by the hardware. Any Pdus pending will be freed to the user.

Hence once this function returns no more receive callbacks will be called for that VC. However, buffer free callbacks will be invoked until such time as all buffers supplied by the user have been freed back to the user.

Calling this function doe not invalidate the connId. ixAtmdAccRxVcEnable() can be invoked to enable Pdu reception again.

If the traffic is already stopped, this function returns IX_SUCCESS.

This function is not reentrant and should not be used inside an interrupt context.

**See also:**

ixAtmdAccRxVcConnect, ixAtmdAccRxVcEnable, ixAtmdAccRxVcDisable

**Parameters**

***connId*** — (in) connection Id as resulted from a successful call to *IxAtmdAccRxVcConnect()*

**Returns**

- IX_SUCCESS — successful call to *ixAtmdAccRxVcDisable()*.
- IX_FAIL — invalid parameters or some unspecified internal error occurred

### A.3.3.9.4 ixAtmdAccRxVcEnable (..)

ixAtmdAccRxVcEnable (IxAtmConnId connId)

Start the RX service on a VC.

This functions kicks-off the traffic reception for a particular VC. Once invoked, incoming PDUs will be made available by the hardware and are eventually directed to the IxAtmdAccRxVcRxCallback callback registered for the connection.

If the traffic is already running, this function returns IX_SUCCESS. This function can be invoked many times.

IxAtmdAccRxVcFreeLowCallback event will occur only after ixAtmdAccRxVcEnable() function is invoked.

Before using this function, the ixAtmdAccRxVcFreeReplenish() function has to be used to replenish the RX Free queue. If not, incoming traffic may be discarded.and in the case of interrupt driven reception the IxAtmdAccRxVcFreeLowCallback callback may be invoked as a side effect during a replenish action.

This function is not reentrant and should not be used inside an interrupt context.

For an VC connection this function can be called after a call to ixAtmdAccRxVcDisable() and should not be called after ixAtmdAccRxVcTryDisconnect()

**See also:**

ixAtmdAccRxVcDisable, ixAtmdAccRxVcConnect, ixAtmdAccRxVcFreeReplenish

#### Parameters

***connId*** — (in)connection Id as resulted from a successful call to *IxAtmdAccRxVcConnect()*

#### Returns

- IX_SUCCESS successful call to ixAtmdAccRxVcEnable
- IX_FAIL invalid parameters or some unspecified internal error occurred

### A.3.3.9.5 ixAtmdAccRxVcFreeEntriesQuery (..)

ixAtmdAccRxVcFreeEntriesQuery (IxAtmConnId connId, unsigned int * numberOfMbufsPtr)

Get the number of rx mbufs the system can accept to replenish the rx reception mechanism on a particular channel.

The ixAtmdAccRxVcFreeEntriesQuery function is used to retrieve the current number of available mbuf entries for reception, on a per-VC basis. This function can be used to know the number of mbufs which can be provided using ixAtmdAccRxVcFreeReplenish().

This function can be used from a timer context, or can be associated with a threshold event, or can be used inside an active polling mechanism which is under user control.

This function is reentrant and does not use system resources and can be invoked from an interrupt context.

#### Parameters

***connId*** — (in) connection Id as resulted from a successful call to *IxAtmdAccRxVcConnect()*

***numberOfMbufsPtr*** — (out) Pointer to the number of available entries. This parameter cannot be a null pointer.

#### Returns

- IX_SUCCESS — the current number of mbufs not yet used for incoming traffic
- IX_FAIL — invalid parameter

#### See also:

ixAtmdAccRxVcFreeReplenish

### A.3.3.9.6 ixAtmdAccRxVcFreeLowCallbackRegister (..)

ixAtmdAccRxVcFreeLowCallbackRegister (IxAtmConnId connId, unsigned int numberOfMbufs, IxAtmdAccRxVcFreeLowCallback callback)

Configure the RX Free threshold value and register a callback to handle threshold notifications.

The function ixAtmdAccRxVcFreeLowCallbackRegister sets the threshold value for a particular RX VC. When the number of buffers reaches this threshold the callback is invoked.

This function should be called once per VC before RX traffic is enabled.This function will fail if the current level of the free buffers is equal or less than the threshold value.

### See also:

ixAtmdAccRxVcFreeLowCallbackRegister, IxAtmdAccRxVcFreeLowCallback, ixAtmdAccRxVcFreeReplenish, ixAtmdAccRxVcFreeEntriesQuery, ixAtmdAccRxVcConnect

### Parameters

***connId*** — (in) connection Id as resulted from a successful call to *IxAtmdAccRxVcConnect()*

***numberOfMbufs*** — (in) threshold number of buffers. This number has to be a power of 2, one of the values 0,1,2,4,8,16,32.... The maximum value cannot be more than half of the rxFree queue size (which can be retrieved using *ixAtmdAccRxVcFreeEntriesQuery()* before any use of the *ixAtmdAccRxVcFreeReplenish()* function)

***callback*** — (in) function telling the user that the number of free buffers has reduced to the threshold value.

### Returns

- IX_SUCCESS — Threshold set successfully.
- IX_FAIL — parameter error or the current number of free buffers is less than or equal to the threshold supplied or some unspecified error has occurred.

### Note

The callback will be called when the threshold level will drop from exactly (numberOfMbufs + 1) to (numberOfMbufs).

## A.3.3.9.7 ixAtmdAccRxVcFreeReplenish (..)

ixAtmdAccRxVcFreeReplenish (IxAtmConnId connId, IX_MBUF * mbufPtr)

Provide free mbufs for data reception on a connection.

This function provides mbufs for data reception by the hardware. This function needs to be called by the user on a regular basis to ensure no packet loss. Providing free buffers is a connection-based feature; each connection can have different requirements in terms of buffer size number of buffers, recycling rate. This function could be invoked from within the context of a IxAtmdAccRxVcFreeLowCallback callback for a particular VC.

Mbufs provided through this function call can be chained. They will be unchained internally. A call to this function with chained mbufs or multiple calls with unchained mbufs are equivalent, but calls with unchained mbufs are more efficient.

Mbufs provided to this interface need to be able to hold at least one full cell payload (48/52 bytes, depending on service type). Chained buffers with a size less than the size supported by the hardware will be returned through the rx callback provided during the connect step.

Failing to invoke this function prior to enabling the RX traffic can result in packet loss.

This function is not reentrant for the same connId.

This function does not use system resources and can be invoked from an interrupt context.

### Notes

Over replenish is detected, and extra mbufs are returned through the rx callback provided during the connect step.

Mbuf provided to the replenish function should have a length greater or equal to 48/52 bytes according to service type.

The memory cache of mMbuf payload should be invalidated prior to Mbuf submission. Flushing the Mbuf headers is handled by IxAtmdAcc.

When a chained mbuf is provided, this function process the mbufs up to the hardware limit and invokes the user-supplied callback to release extra buffers.

In the case where the VC on which the connection is made is a fastpath VC, buffers may be used to hold exception PDUs where the AAL5 crc is wrong or the pdu cannot be stored in the fast path mbuf. Buffers provided to the replenish function need to be the same size or bigger than the Fast Path buffers. If not, the Pdu passed trough the user callback will be invalidated.

### See also:

ixAtmdAccRxVcFreeLowCallbackRegister, IxAtmdAccRxVcFreeLowCallback, ixAtmdAccRxVcConnect

### Parameters

***connId*** — (in) connection Id as returned from a successful call to *IxAtmdAccRxVcConnect()*

***mbufPtr*** — (in) pointer to a mbuf structure to be used for data reception. The mbuf pointed to by this parameter can be chained to an other mbuf.

### Returns

- IX_SUCCESS — successful call to ixAtmdAccRxVcFreeReplenish() and the mbuf is now ready to use for incoming traffic.
- IX_ATMDACC_BUSY — cannot process this request because the max number of outstanding free buffers has been reached or the internal resources have exhausted for this VC. The user is responsible for retrying this request later.
- IX_FAIL — cannot process this request because of parameter errors or some unspecified internal error has occurred.

### Notes

It is not always guaranteed the replenish step to be as fast as the hardware is consuming Rx Free mbufs. There is nothing in IxAtmdAcc to guarantee that replenish reaches the rxFree threshold level. If the threshold level is not reached, the next rxFree low notification for this channel will not be triggered.

The preferred ways to replenish can be as follows (depending on applications and implementations).

- Replenish in a rxFree low notification until the function ixAtmdAccRxVcFreeReplenish() returns IX_ATMDACC_BUSY
- Query the queue level using

### See also:

ixAtmdAccRxVcFreeEntriesQuery, then, replenish using ixAtmdAccRxVcFreeReplenish(), then query the queue level again, and replenish if the threshold is still not reached.

Trigger replenish from an other event source and use rxFree starvation to throttle the Rx traffic.

## A.3.3.9.8 ixAtmdAccRxVcTryDisconnect (..)

ixAtmdAccRxVcTryDisconnect (IxAtmConnId connId)

Disconnect a VC from the RX service.

This function deregisters the VC and guarantees that all resources associated with this VC are free. After its execution, the connection Id is not available.

This function will fail until such time as all resources allocated to the VC connection have been freed. The user is responsible to delay and call again this function many times until a success status is returned.

This function needs internal locks and should not be called from an interrupt context.

### Parameters

***connId*** — (in) connection Id as resulted from a successful call to *IxAtmdAccRxVcConnect()*

### Returns

- IX_SUCCESS — successful call to ixAtmdAccRxVcDisable
- IX_ATMDACC_RESOURCES_STILL_ALLOCATED — not all resources associated with the connection have been freed
- IX_FAIL — cannot process this request because of a parameter error

## A.3.3.9.9 ixAtmdAccShow (void)

Show IxAtmdAcc configuration on a per port basis.

### Parameters

None

### Returns

None

### Note

Display use printf() and are redirected to stdout.

## A.3.3.9.10 ixAtmdAccStatsReset (void)

Reset all IxAtmdAcc stats.

### Parameters

None

#### Returns

None

### A.3.3.9.11 ixAtmdAccStatsShow (void)

Show all IxAtmdAcc stats.

#### Parameters

None

#### Returns

None

#### Note

Stats display use printf() and are redirected to stdout.

### A.3.3.9.12 ixAtmdAccTxVcConnect (..)

ixAtmdAccTxVcConnect (IxAtmLogicalPort port, unsigned int vpi, unsigned int vci, IxAtmdAccAalType aalServiceType, IxAtmdAccUserId userId, IxAtmdAccTxVcBufferReturnCallback bufferFreeCallback, IxAtmConnId * connIdPtr)

Connect to a Aal Pdu transmit service for a particular port/vpi/vci and service type.

This function allows a user to connect to an Aal5/Aal0/OAM Pdu transmit service for a particular port/vpi/vci. It registers the callback and allocates internal resources and a Connection Id to be used in further API calls related to this VC.

The function will setup VC transmit service on the specified on the specified port. A connId is the reference by which IxAtmdAcc refers to a connected VC. This identifier is the result of a successful call to a connect function. This identifier is invalid after a successful call to a disconnect function.

This function needs internal locks, and hence should not be called from an interrupt context.

On return from ixAtmdAccTxVcConnect() with a failure status, the connection Id parameter is unspecified. Its value cannot be used.

Calling this function for the same combination of port, Vpi, Vci and more than once without calling ixAtmdAccTxVcTryDisconnect() will result in a failure status.

Two AAL0 services supporting 48 or 52 byte cell data are provided. Submitted AAL0 PDUs must be a multiple of the cell data size (48/52). AAL0_52 is a raw cell service the client must format the PDU with an ATM cell header (excluding HEC) at the start of each cell, note that AtmdAcc does not validate the cell headers in a submitted PDU.

For the OAM service an “OAM Tx channel” may be enabled for a port by establishing a single dedicated OAM Tx connection on that port.

The OAM service allows buffers containing 52 byte OAM F4/F5 cells to be transmitted and received over the dedicated OAM channels. HEC is appended/removed, and CRC-10 performed by the NPE. The OAM service offered by AtmdAcc is a raw cell transport service. It is assumed that ITU I.610 procedures that make use of this service are implemented above AtmdAcc.

Note that the dedicated OAM connections are established on reserved VPI,VCI, and (in the case of Rx) port values. These values are used purely to discriminate the dedicated OAM channels and do not identify a particular OAM F4/F5 flow. F4/F5 flows may be realized for particular VPI/VCIs by manipulating the VPI,VCI fields of the ATM cell headers of cells in the buffers passed to AtmdAcc.

Calling this function before enabling the port will fail.

#### See also:

ixAtmdAccTxVcTryDisconnect, ixAtmdAccPortTxScheduledModeEnable, ixAtmdAccPortEnable

#### Parameters

***port*** (in) VC identification — logical PHY port [*IX_UTOPIA_PORT_0* .. *IX_UTOPIA_MAX_PORTS* - 1]

***vpi*** (in) VC identification — ATM Vpi [0..255] or IX_ATMDACC_OAM_VPI

***vci*** (in) VC identification — ATM Vci [0..65535] or IX_ATMDACC_OAM_VCI

***aalServiceType*** — (in) type of service AAL5, AAL0_48, AAL0_52, or OAM

***userId*** — (in) user id to be used later during callbacks related to this channel

***bufferFreeCallback*** — (in) function called when mbufs transmission is complete. This parameter cannot be a null pointer.

***connIdPtr*** — (out) Pointer to a connection Id. This parameter cannot be a null pointer.

#### Returns

- IX_SUCCESS successful call to *IxAtmdAccRxVcConnect()*.
- IX_ATMDACC_BUSY cannot process this request because no VC is available.
- IX_FAIL parameter error, VC already in use, attempt to connect AAL service on reserved OAM VPI/VCI, attempt to connect OAM service on VPI/VCI other than the reserved OAM VPI/VCI, port is not initialized, or some other error occurs during processing.

#### Note

Unscheduled mode is not supported in IXP400 software v1.0. Therefore, the function ixAtmdAccPortTxScheduledModeEnable() needs to be called for this port before any establishing a Tx Connection.

### A.3.3.9.13 ixAtmdAccTxVcPduSubmit (..)

ixAtmdAccTxVcPduSubmit (IxAtmConnId connId, IX_MBUF * mbufPtr, IxAtmdAccClpStatus clp, unsigned int numberOfCells)

Submit a PDU for transmission on connection.

A data user calls this function to submit an mbufs containing a Pdu to be transmitted. The buffer supplied can be chained and the Pdu it contains must be complete.

The transmission behavior of this call depends on the operational mode of the port on which the connection is made.

In unscheduled mode the mbuf will be submitted to the hardware immediately if sufficient resource is available. Otherwise the function will return failure.

In scheduled mode the buffer is queued internally in IxAtmdAcc. The cell demand is made known to the traffic shaping entity. Cells from the buffers are MUXed onto the port some time later as dictated by the traffic shaping entity. The traffic shaping entity does this by sending transmit schedules to IxAtmdAcc via ixAtmdAccPortTxProcess() function call.

Note that the dedicated OAM channel is scheduled just like any other channel. This means that any OAM traffic relating to an active AAL0/AAL5 connection will be scheduled independently of the AAL0/AAL5 traffic for that connection.

When transmission is complete, the TX Done mechanism will give the ownership of these buffers back to the customer. The tx done mechanism must be in operation before transmission is attempted.

For AAL0/OAM submitted AAL0 PDUs must be a multiple of the cell data size (48/52). AAL0_52 and OAM are raw cell services, and the client must format the PDU with an ATM cell header (excluding HEC) at the start of each cell, note that AtmdAcc does not validate the cell headers in a submitted PDU.

### See also:

IxAtmdAccTxVcBufferReturnCallback, ixAtmdAccTxDoneDispatch

### Parameters

***connId*** — (in) connection Id as resulted from a successful call to ixAtmdAccTxVcConnect().

***mbufPtr*** — (in) pointer to a chained structure of mbufs to transmit. This parameter cannot be a null pointer.

***clp*** — (in) clp indication for this PDU. All cells of this pdu will be sent with the clp bit set.

***numberOfCells*** — (in) number of cells in the PDU.

### Returns

- IX_SUCCESS — successful call to ixAtmdAccTxVcPduSubmit() The pdu pointed by the mbufPtr parameter will be transmitted.
- IX_ATMDACC_BUSY — unable to process this request because internal resources are all used. The caller is responsible for retrying this request later.
- IX_FAIL — unable to process this request because of error in the parameters (wrong connId supplied, or wrong mbuf pointer supplied), the total length of all buffers in the chain should be a multiple of the cell size (48/52 depending on the service type), or unspecified error during processing.

### Notes

This function in not re-entrant for the same VC (for example, two thread cannot send PDUs for the same VC). But two threads can safely call this function with a different connection Id.

In unscheduled mode, this function is not re-entrant on a per port basis. The size of pdus is limited to 8Kb.

0-length mbufs should be removed from the chain before submission. The total length of the pdu (sdu + padding +trailer) has to be updated in the header of the first mbuf of a chain of mbufs.

Aal5 trailer information (UUI, CPI, SDU length) has to be supplied before submission.

The payload memory cache should be flushed, if needed, prior to transmission. Mbuf headers are flushed by IxAtmdAcc.

This function does not use system resources and can be used inside an interrupt context.

## A.3.3.9.14 ixAtmdAccTxVcTryDisconnect (..)

ixAtmdAccTxVcTryDisconnect (IxAtmConnId connId)

Disconnect from a Aal Pdu transmit service for a particular port/vpi/vci.

This function deregisters the VC and guarantees that all resources associated with this VC are free. After its execution, the connection Id is not available.

This function will fail until such time as all resources allocated to the VC connection have been freed. The user is responsible to delay and call again this function many times until a success status is returned.

After its execution, the connection Id is not available.

### Parameters

***connId*** — (in) connection Id as resulted from a successful call to ixAtmdAccTxVcConnect()

### Returns

- IX_SUCCESS — successful call to ixAtmdAccTxVcTryDisconnect().
- IX_ATMDACC_RESOURCES_STILL_ALLOCATED — not all resources associated with the connection have been freed. This condition will disappear after Tx and Extend is complete for this channel.
- IX_FAIL — unable to process this request because of errors in the parameters (wrong connId supplied).

### Notes

This function needs internal locks and should not be called from an interrupt context

If the *IX_ATMDACC_RESOURCES_STILL_ALLOCATED* error does not clear after a while, this may be linked to a previous problem of cell overscheduling. Disabling the port and retry a disconnect will free the resources associated with this channel.

**See also:**

ixAtmdAccPortTxProcess

## A.3.4 ATM Driver Access (IxAtmdAcc) Control API

The public API for the Intel® IXP42X product line ATM Driver Control component.

### A.3.4.1 Modules

Intel® IXP42X product line ATM Driver Access (IxAtmdAcc) UTOPIA Control API

The public API for the Intel® IXP42X product line *ATM Driver Control component.*

### A.3.4.2 Defines

- #define IX_ATMDACC_PORT_DISABLE_IN_PROGRESS 5
  Port enable return code.
- #define IX_ATMDACC_ALLPDUS 0xffffffff
  All PDUs.

### A.3.4.3 Typedefs

- typedef IX_STATUS(* IxAtmdAccRxDispatcher )(IxAtmRxQueueId rxQueueId, unsigned int numberOfPdusToProcess, unsigned int *reservedPtr)
  Callback prototype for notification of available PDUs for an Rx Q.
- typedef IX_STATUS(* IxAtmdAccTxDoneDispatcher )(unsigned int numberOfPdusToProcess, unsigned int *reservedPtr)
  Callback prototype for transmitted mbuf when threshold level is crossed.
- typedef void(* IxAtmdAccPortTxLowCallback )(IxAtmLogicalPort port, unsigned int numberOfAvailableCells)
  Notification that the threshold number of scheduled cells remains in a port's transmit Q.
- typedef IX_STATUS(* IxAtmdAccTxVcDemandUpdateCallback )(IxAtmLogicalPort port, int vcId, unsigned int numberOfCells)
  Prototype to submit cells for transmission.
- typedef void(* IxAtmdAccTxVcDemandClearCallback )(IxAtmLogicalPort port, int vcId)
  prototype to remove all currently queued cells from a registered VC.
- typedef IX_STATUS(* IxAtmdAccTxSchVcIdGetCallback )(IxAtmLogicalPort port, unsigned int vpi, unsigned int vci, IxAtmConnId connId, int *vcId)
  Prototype to get a scheduler vc id.

### A.3.4.4 Functions

- PUBLIC IX_STATUS ixAtmdAccRxDispatcherRegister (IxAtmRxQueueId queueId, IxAtmdAccRxDispatcher callback)

  Register a notification callback to be invoked when there is at least one entry on a particular Rx queue.

- PUBLIC IX_STATUS ixAtmdAccRxDispatch (IxAtmRxQueueId rxQueueId, unsigned int numberOfPdusToProcess, unsigned int *numberOfPdusProcessedPtr)

  Control function which executes Rx processing for a particular Rx stream.

- PUBLIC IX_STATUS ixAtmdAccRxLevelQuery (IxAtmRxQueueId rxQueueId, unsigned int *numberOfPdusPtr)

  Query the number of entries in a particular RX queue.

- PUBLIC IX_STATUS ixAtmdAccRxQueueSizeQuery (IxAtmRxQueueId rxQueueId, unsigned int *numberOfPdusPtr)

  Query the size of a particular RX queue.

- PUBLIC IX_STATUS ixAtmdAccPortTxFreeEntriesQuery (IxAtmLogicalPort port, unsigned int *numberOfCellsPtr)

  Get the number of available cells the system can accept for transmission.

- PUBLIC IX_STATUS ixAtmdAccPortTxCallbackRegister (IxAtmLogicalPort port, unsigned int numberOfCells, IxAtmdAccPortTxLowCallback callback)

  Configure the Tx port threshold value and register a callback to handle threshold notifications.

- PUBLIC IX_STATUS ixAtmdAccPortTxScheduledModeEnable (IxAtmLogicalPort port, IxAtmdAccTxVcDemandUpdateCallback vcDemandUpdateCallback, IxAtmdAccTxVcDemandClearCallback vcDemandClearCallback, IxAtmdAccTxSchVcIdGetCallback vcIdGetCallback)

  Put the port into Scheduled Mode.

- PUBLIC IX_STATUS ixAtmdAccPortTxProcess (IxAtmLogicalPort port, IxAtmScheduleTable *scheduleTablePtr)

  Transmit queue cells to the H/W based on the supplied schedule table.

- PUBLIC IX_STATUS ixAtmdAccTxDoneDispatch (unsigned int numberOfPdusToProcess, unsigned int *numberOfPdusProcessedPtr)

  Process a number of pending transmit done pdus from the hardware.

- PUBLIC IX_STATUS ixAtmdAccTxDoneLevelQuery (unsigned int *numberOfPdusPtr)

  Query the current number of transmit pdus ready for recycling.

- PUBLIC IX_STATUS ixAtmdAccTxDoneQueueSizeQuery (unsigned int *numberOfPdusPtr)

  Query the TxDone queue size.

- PUBLIC IX_STATUS ixAtmdAccTxDoneDispatcherRegister (unsigned int numberOfPdus, IxAtmdAccTxDoneDispatcher notificationCallback)

  Configure the Tx Done stream threshold value and register a callback to handle threshold notifications.

- PUBLIC IX_STATUS ixAtmdAccUtopiaConfigSet (const IxAtmdAccUtopiaConfig *ixAtmdAccUtopiaConfigPtr)

  Send the configuration structure to the Utopia interface.

- PUBLIC IX_STATUS ixAtmdAccUtopiaStatusGet (IxAtmdAccUtopiaStatus *ixAtmdAccUtopiaStatus)

  Get the Utopia interface configuration.
- PUBLIC IX_STATUS ixAtmdAccPortEnable (IxAtmLogicalPort port)

  Enable a PHY logical port.
- PUBLIC IX_STATUS ixAtmdAccPortDisable (IxAtmLogicalPort port)

  Disable a PHY logical port.
- PUBLIC BOOL ixAtmdAccPortDisableComplete (IxAtmLogicalPort port)

  Disable a PHY logical port.

### A.3.4.5 Detailed Description

The public API for the Intel® IXP42X product line ATM Driver Control component.

IxAtmdAcc is the low level interface by which AAL PDU get transmitted to, and received from the UTOPIA bus.

This part is related to the Control configuration.

### A.3.4.6 Define Documentation

#### A.3.4.6.1 #define IX_ATMDACC_ALLPDUS 0xffffffff

All PDUs.

This constant is used to tell IxAtmDAcc to process all PDUs from the RX queue or the TX Done.

**See also:**

IxAtmdAccRxDispatcher, IxAtmdAccTxDoneDispatcher

#### A.3.4.6.2 #define IX_ATMDACC_PORT_DISABLE_IN_PROGRESS 5

Port enable return code.

This constant is used to tell IxAtmDAcc user that the port disable functions are not complete. The user can call ixAtmdAccPortDisableComplete() to find out when the disable has finished. The port enable can then proceed.

### A.3.4.7 Typedef Documentation

#### A.3.4.7.1 typedef void(..)

typedef void(* IxAtmdAccPortTxLowCallback)(IxAtmLogicalPort port, unsigned int numberOfAvailableCells)

Notification that the threshold number of scheduled cells remains in a port's transmit Q.

The is the prototype for of the user notification function which gets called on a per-port basis, when the number of remaining scheduled cells to be transmitted decreases to the threshold level. The number of cells passed as a parameter can be used for scheduling purposes as the maximum number of cells that can be passed in a schedule table to the ixAtmdAccPortTxProcess() function.

#### See also:

ixAtmdAccPortTxCallbackRegister, ixAtmdAccPortTxProcess, ixAtmdAccPortTxFreeEntriesQuery

#### Parameters

***port*** — (in) logical PHY port [*IX_UTOPIA_PORT_0* .. *IX_UTOPIA_MAX_PORTS* - 1].

***numberOfAvailableCells*** — (in) number of available cell entries.for the port.

#### Note

This functions shall not use system resources when used inside an interrupt context.

### A.3.4.7.2 typedef IX_STATUS(..)

typedef IX_STATUS(* IxAtmdAccRxDispatcher)(IxAtmRxQueueId rxQueueId, unsigned int numberOfPdusToProcess, unsigned int *reservedPtr)

Callback prototype for notification of available PDUs for an Rx Q.

This is a prototype for a function which is called when there is at least one Pdu available for processing on a particular Rx Q.

This function should call ixAtmdAccRxDispatch() with the appropriate number of parameters to read and process the Rx Q.

#### See also:

ixAtmdAccRxDispatch, ixAtmdAccRxVcConnect, ixAtmdAccRxDispatcherRegister

#### Parameters

***rxQueueId*** — (in) indicates which RX queue to has Pdus to process.

***numberOfPdusToProcess*** — (in) indicates the minimum number of PDUs available to process all PDUs from the queue.

***reservedPtr*** — (out) pointer to a int location which can be written to, but does not retain written values. This is provided to make this prototype compatible with ixAtmdAccRxDispatch().

#### Returns

- int — ignored.

### A.3.4.7.3 typedef IX_STATUS(..)

typedef IX_STATUS(* IxAtmdAccTxDoneDispatcher)(unsigned int numberOfPdusToProcess, unsigned int *reservedPtr)

Callback prototype for transmitted mbuf when threshold level is crossed.

IxAtmdAccTxDoneDispatcher is the prototype of the user function which get called when pdus are completely transmitted. This function is likely to call the ixAtmdAccTxDoneDispatch() function.

This function is called when the number of available pdus for reception is crossing the threshold level as defined in ixAtmdAccTxDoneDispatcherRegister().

This function is called inside an Qmgr dispatch context. No system resource or interrupt-unsafe feature should be used inside this callback.

Transmitted buffers recycling implementation is a system-wide mechanism and needs to be set before any traffic is started. If this threshold mechanism is not used, the user is responsible for polling the transmitted buffers with ixAtmdAccTxDoneDispatch() and ixAtmdAccTxDoneLevelQuery() functions.

**See also:**

ixAtmdAccTxDoneDispatcherRegister, ixAtmdAccTxDoneDispatch, ixAtmdAccTxDoneLevelQuery

**Parameters**

***numberOfPdusToProcess*** — (in) The current number of pdus currently available for recycling.

***reservedPtr*** — (out) pointer to a int location which can be written to but does not retain written values. This is provided to make this prototype compatible with ixAtmdAccTxDoneDispatch().

**Returns**

- IX_SUCCESS This is provided to make this prototype compatible with ixAtmdAccTxDoneDispatch().
- IX_FAIL invalid parameters or some unspecified internal error occurred. This is provided to make this prototype compatible with ixAtmdAccTxDoneDispatch().

### A.3.4.7.4 typedef IX_STATUS(..)

typedef IX_STATUS(* IxAtmdAccTxSchVcIdGetCallback)(IxAtmLogicalPort port, unsigned int vpi, unsigned int vci, IxAtmConnId connId, int *vcId)

prototype to get a scheduler vc id.

IxAtmdAccTxSchVcIdGetCallback is the prototype of the function to get a scheduler vcId

**See also:**

IxAtmdAccTxVcDemandUpdateCallback, IxAtmdAccTxVcDemandClearCallback, IxAtmdAccTxSchVcIdGetCallback, ixAtmdAccPortTxScheduledModeEnable

**Parameters**

***port*** — (in) Specifies the ATM logical port on which the VC is established.

***vpi*** — (in) For AAL0/AAL5 specifies the ATM vpi on which the VC is established. For OAM specifies the dedicated “OAM Tx channel” VPI.

***vci*** — (in) For AAL0/AAL5 specifies the ATM vci on which the VC is established. For OAM specifies the dedicated “OAM Tx channel” VCI.

***connId*** — (in) specifies the IxAtmdAcc connection Id already associated with this VC.

***vcId*** — (out) pointer to a vcId.

#### Returns

- IX_SUCCESS — the function is returning a Scheduler vcId for this VC.
- IX_FAIL — the function cannot process scheduling for this VC. the contents of vcId is unspecified.

### A.3.4.7.5 typedef void(..)

typedef void(* IxAtmdAccTxVcDemandClearCallback)(IxAtmLogicalPort port, int vcId)

prototype to remove all currently queued cells from a registered VC.

IxAtmdAccTxVcDemandClearCallback is the prototype of the function to remove all currently queued cells from a registered VC. The pending cell count for the specified VC is reset to zero. After the use of this callback, the scheduler shall not schedule more cells for this VC.

This callback function is called during a VC disconnection ixAtmdAccTxVcTryDisconnect().

#### See also:

IxAtmdAccTxVcDemandUpdateCallback, IxAtmdAccTxVcDemandClearCallback, IxAtmdAccTxSchVcIdGetCallback, ixAtmdAccPortTxScheduledModeEnable, ixAtmdAccTxVcTryDisconnect

#### Parameters

***port*** — (in) Specifies the ATM port on which the VC to be cleared is established.

***vcId*** — (in) Identifies the VC to be cleared. This is the value returned by the IxAtmdAccTxSchVcIdGetCallback call.

#### Returns

None

### A.3.4.7.6 typedef IX_STATUS(..)

typedef IX_STATUS(* IxAtmdAccTxVcDemandUpdateCallback)(IxAtmLogicalPort port, int vcId, unsigned int numberOfCells)

Prototype to submit cells for transmission.

IxAtmdAccTxVcDemandUpdateCallback is the prototype of the callback function used by AtmD to notify an ATM Scheduler that the user of a VC has submitted cells for transmission.

#### See also:

IxAtmdAccTxVcDemandUpdateCallback, IxAtmdAccTxVcDemandClearCallback, IxAtmdAccTxSchVcIdGetCallback, ixAtmdAccPortTxScheduledModeEnable

#### Parameters

***port*** — (in) Specifies the ATM port on which the VC to be updated is established.

***vcId*** — (in) Identifies the VC to be updated. This is the value returned by the IxAtmdAccTxSchVcIdGetCallback call.

***numberOfCells*** — (in) Indicates how many ATM cells should be added to the queue for this VC.

#### Returns

- IX_SUCCESS — function is registering the cell demand for this VC.
- IX_FAIL — function cannot register cell for this VC : the scheduler maybe overloaded or misconfigured.

## A.3.4.8 Function Documentation

### A.3.4.8.1 ixAtmdAccPortDisable (..)

ixAtmdAccPortDisable (IxAtmLogicalPort port)

Disable a PHY logical port.

This function disable the transmission over one port.

When a port is disabled, the cell transmission to the UTOPIA interface is stopped.

#### Parameters

***port*** — (in) logical PHY port [*IX_UTOPIA_PORT_0 .. IX_UTOPIA_MAX_PORTS* - 1]

#### Returns

- IX_SUCCESS — disable is complete.
- IX_ATMDACC_WARNING — port already disabled.
- IX_FAIL — disable failed, wrong parameter.

#### Notes

This function needs internal locks and should not be called from an interrupt context.

The response from hardware is done through the txDone mechanism to ensure the synchronization with tx resources. Therefore, the txDone mechanism needs to be serviced to make a PortDisable complete.

#### See also:

ixAtmdAccPortEnable, ixAtmdAccPortDisableComplete, ixAtmdAccTxDoneDispatch

### A.3.4.8.2 ixAtmdAccPortDisableComplete (..)

ixAtmdAccPortDisableComplete (IxAtmLogicalPort port)

disable a PHY logical port.

This function indicates if the port disable for a port has completed. This function will return TRUE if the port has never been enabled.

#### Parameters

***port*** — (in) logical PHY port [*IX_UTOPIA_PORT_0* .. *IX_UTOPIA_MAX_PORTS* - 1].

#### Returns

- TRUE — Disable is complete.
- FALSE — Disable failed, wrong parameter.

#### Note

This function needs internal locks and should not be called from an interrupt context.

#### See also:

ixAtmdAccPortEnable, ixAtmdAccPortDisable

### A.3.4.8.3 ixAtmdAccPortEnable (..)

ixAtmdAccPortEnable (IxAtmLogicalPort port)

Enable a PHY logical port.

This function enables the transmission over one port. It should be called before accessing any resource from this port and before the establishment of a VC.

When a port is enabled, the cell transmission to the UTOPIA interface is started. If there is no traffic already running, idle cells are sent over the interface.

This function can be called multiple times.

#### Parameters

***port*** — (in) logical PHY port [*IX_UTOPIA_PORT_0* .. *IX_UTOPIA_MAX_PORTS* - 1]

#### Returns

- IX_SUCCESS — enable is complete.
- IX_ATMDACC_WARNING — port already enabled.
- IX_FAIL — enable failed, wrong parameter, or cannot initialize this port (the port is maybe already in use, or there is a hardware issue).

#### Note

This function needs internal locks and should not be called from an interrupt context.

#### See also:

ixAtmdAccPortDisable

### A.3.4.8.4 ixAtmdAccPortTxCallbackRegister (..)

ixAtmdAccPortTxCallbackRegister (IxAtmLogicalPort port, unsigned int numberOfCells, IxAtmdAccPortTxLowCallback callback)

Configure the Tx port threshold value and register a callback to handle threshold notifications.

This function sets the threshold in cells.

#### See also:

ixAtmdAccPortTxCallbackRegister, ixAtmdAccPortTxProcess, ixAtmdAccPortTxFreeEntriesQuery

#### Parameters

***port*** — (in) logical PHY port [*IX_UTOPIA_PORT_0* .. *IX_UTOPIA_MAX_PORTS* - 1].

***numberOfCells*** — (in) threshold value which triggers the callback invocation, This number has to be one of the values 0,1,2,4,8,16,32 .... The maximum value cannot be more than half of the txVc queue size (which can be retrieved using ixAtmdAccPortTxFreeEntriesQuery() before any Tx traffic is sent for this port).

***callback*** — (in) callback function to invoke when the threshold level is reached. This parameter cannot be a null pointer.

#### Returns

- IX_SUCCESS — Successful call to ixAtmdAccPortTxCallbackRegister().
- IX_FAIL — error in the parameters, Tx channel already set for this port threshold level is not correct or within the range regarding the queue size: or unspecified error during processing.

#### Notes

This callback function get called when the threshold level drops from (numberOfCells+1) cells to (numberOfCells) cells.

This function should be called during system initialization, outside an interrupt context.

### A.3.4.8.5 ixAtmdAccPortTxFreeEntriesQuery (..)

ixAtmdAccPortTxFreeEntriesQuery (IxAtmLogicalPort port, unsigned int * numberOfCellsPtr)

Get the number of available cells the system can accept for transmission.

The function is used to retrieve the number of cells that can be queued for transmission to the hardware.

This number is based on the worst schedule table where one cell is stored in one schedule table entry, depending on the pdus size and mbuf size and fragmentation.

This function doesn't use system resources and can be used from a timer context, or can be associated with a threshold event, or can be used inside an active polling mechanism.

### Parameters

***port*** — (in) logical PHY port [*IX_UTOPIA_PORT_0 .. IX_UTOPIA_MAX_PORTS* - 1]

***numberOfCellsPtr*** — (out) number of available cells. This parameter cannot be a null pointer.

### See also:

ixAtmdAccPortTxProcess

### Returns

- IX_SUCCESS numberOfCellsPtr contains the number of cells that can be scheduled for this port.
- IX_FAIL error in the parameters, or some processing error occurred.

## A.3.4.8.6 ixAtmdAccPortTxProcess (..)

ixAtmdAccPortTxProcess (IxAtmLogicalPort port, IxAtmScheduleTable * scheduleTablePtr)

Transmit queue cells to the H/W based on the supplied schedule table.

This function ixAtmdAccPortTxProcess() process the schedule table provided as a parameter to the function. As a result cells are sent to the underlaying hardware for transmission.

The schedule table is executed in its entirety or not at all. So the onus is on the caller not to submit a table containing more cells than can be transmitted at that point. The maximum numbers that can be transmitted is guaranteed to be the number of cells as returned by the function ixAtmdAccPortTxFreeEntriesQuery().

When the scheduler is invoked on a threshold level, IxAtmdAcc gives the minimum number of cells (to ensure the callback will fire again later) and the maximum number of cells that ixAtmdAccPortTxProcess() will be able to process (assuming the ATM scheduler is able to produce the worst-case schedule table, i.e. one entry per cell).

When invoked outside a threshold level, the overall number of cells of the schedule table should be less than the number of cells returned by the ixAtmdAccPortTxFreeEntriesQuery() function.

After invoking the ixAtmdAccPortTxProcess() function, it is the user choice to query again the queue level with the function ixAtmdAccPortTxFreeEntriesQuery() and, depending on a new cell number, submit an other schedule table.

IxAtmdAcc will check that the number of cells in the schedule table is compatible with the current transmit level.

Obsolete or invalid connection Id will be silently discarded.

This function is not reentrant for the same port.

This function does not use system resources and can be used inside an interrupt context.

This function is used as a response to the hardware requesting more cells to transmit.

### See also:

ixAtmdAccPortTxScheduledModeEnable, ixAtmdAccPortTxFreeEntriesQuery, ixAtmdAccPortTxCallbackRegister, ixAtmdAccPortEnable

### Parameters

***port*** — (in) logical PHY port [*IX_UTOPIA_PORT_0* .. *IX_UTOPIA_MAX_PORTS* - 1].

***scheduleTablePtr*** — (in) pointer to a scheduler update table. The content of this table is not modified by this function. This parameter cannot be a null pointer.

### Returns

- IX_SUCCESS — the schedule table process is complete and cells are transmitted to the hardware.
- IX_ATMDACC_WARNING — Traffic will be dropped: the schedule table exceed the hardware capacity If this error is ignored, further traffic and schedule will work correctly. Overscheduling does not occur when the schedule table does not contain more entries that the number of free entries returned by ixAtmdAccPortTxFreeEntriesQuery(). However, Disconnect attempts just after this error will fail permanently with the error code IX_ATMDACC_RESOURCES_STILL_ALLOCATED, and it is necessary to disable the port to make ixAtmdAccTxVcTryDisconnect() successful.
- IX_FAIL — a wrong parameter is supplied, or the format of the schedule table is invalid, or the port is not Enabled, or an internal severe error occurred. No cells is transmitted to the hardware.

### Note

If the failure is linked to an overschedule of data cells the result is an inconsistency in the output traffic (one or many cells may be missing and the traffic contract is not respected).

## A.3.4.8.7 ixAtmdAccPortTxScheduledModeEnable (..)

ixAtmdAccPortTxScheduledModeEnable (IxAtmLogicalPort port,
IxAtmdAccTxVcDemandUpdateCallback vcDemandUpdateCallback,
IxAtmdAccTxVcDemandClearCallback vcDemandClearCallback,
IxAtmdAccTxSchVcIdGetCallback vcIdGetCallback)

Put the port into Scheduled Mode.

This function puts the specified port into scheduled mode of transmission which means an external s/w entity controls the transmission of cells on this port. This facilitates traffic shaping on the port.

Any buffers submitted on a VC for this port will be queued in IxAtmdAcc. The transmission of these buffers to and by the hardware will be driven by a transmit schedule submitted regularly in calls to ixAtmdAccPortTxProcess() by traffic shaping entity.

The transmit schedule is expected to be dynamic in nature based on the demand in cells for each VC on the port. Hence the callback parameters provided to this function allow IxAtmdAcc to inform the shaping entity of demand changes for each VC on the port.

By default a port is in Unscheduled Mode so if this function is not called, transmission of data is done without scheduling rules, on a first-come, first-out basis.

Once a port is put in scheduled mode it cannot be reverted to un-scheduled mode. Note that unscheduled mode is not supported in IXP400 software v1.0.

#### Notes

This function should be called before any VCs have be connected on a port. Otherwise this function call will return failure.

This function uses internal locks and should not be called from an interrupt context.

#### See also:

IxAtmdAccTxVcDemandUpdateCallback, IxAtmdAccTxVcDemandClearCallback, IxAtmdAccTxSchVcIdGetCallback, ixAtmdAccPortTxProcess

#### Parameters

***port*** — (in) logical PHY port [*IX_UTOPIA_PORT_0 .. IX_UTOPIA_MAX_PORTS* - 1]

***vcDemandUpdateCallback*** — (in) callback function used to update the number of outstanding cells for transmission. This parameter cannot be a null pointer.

***vcDemandClearCallback*** — (in) callback function used to remove all clear the number of outstanding cells for a VC. This parameter cannot be a null pointer.

***vcIdGetCallback*** — (in) callback function used to exchange vc Identifiers between IxAtmdAcc and the entity supplying the transmit schedule. This parameter cannot be a null pointer.

#### Returns

- IX_SUCCESS — scheduler registration is complete and the port is now in scheduled mode.
- IX_FAIL — failed (wrong parameters, or traffic is already enabled on this port, possibly without ATM shaping).

### A.3.4.8.8 ixAtmdAccRxDispatch (..)

ixAtmdAccRxDispatch (IxAtmRxQueueId rxQueueId, unsigned int numberOfPdusToProcess, unsigned int * numberOfPdusProcessedPtr)

Control function which executes Rx processing for a particular Rx stream.

The IxAtmdAccRxDispatch() function is used to process received Pdus available from one of the two incoming RX streams. When this function is invoked, the incoming traffic (up to the number of PDUs passed as a parameter) will be transferred to the IxAtmdAcc users through the callback IxAtmdAccRxVcRxCallback, as registered during the ixAtmdAccRxVcConnect() call.

The user receive callbacks will be executed in the context of this function.

Failing to use this function on a regular basis when there is traffic will block incoming traffic and can result in Pdus being dropped by the hardware.

This should be used to control when received pdus are handed off from the hardware to Aal users from a particular stream. The function can be used from a timer context, or can be registered as a callback in response to an rx stream threshold event, or can be used inside an active polling mechanism which is under user control.

### Note

The signature of this function is directly compatible with the callback prototype which can be register with ixAtmdAccRxDispatcherRegister().

### See also:

ixAtmdAccRxDispatcherRegister, IxAtmdAccRxVcRxCallback, ixAtmdAccRxVcFreeEntriesQuery

### Parameters

***rxQueueId*** — (in) indicates which RX queue to process.

***numberOfPdusToProcess*** — (in) indicates the maximum number of PDU to remove from the RX queue. A value of IX_ATMDACC_ALLPDUS indicates to process all PDUs from the queue. This includes at least the PDUs in the queue when the function is invoked. Because of real-time constraints, there is no guarantee that the queue will be empty when the function exits. If this parameter is greater than the number of entries of the queues, the function will succeed and the parameter numberOfPdusProcessedPtr will reflect the exact number of PDUs processed.

***numberOfPdusProcessedPtr*** — (out) indicates the actual number of PDU processed during this call. This parameter cannot be a null pointer.

### Returns

- IX_SUCCESS — the number of PDUs as indicated in numberOfPdusProcessedPtr are removed from the RX queue and the VC callback are called.
- IX_FAIL — invalid parameters or some unspecified internal error occurred.

## A.3.4.8.9 ixAtmdAccRxDispatcherRegister (..)

ixAtmdAccRxDispatcherRegister (IxAtmRxQueueId queueId, IxAtmdAccRxDispatcher callback)

Register a notification callback to be invoked when there is at least one entry on a particular Rx queue.

This function registers a callback to be invoked when there is at least one entry in a particular queue. The registered callback is called every time when the hardware adds one or more pdus to the specified Rx queue.

This function cannot be used when a Rx Vc using this queue is already existing.

### Note

The callback function can be the API function ixAtmdAccRxDispatch() : every time the threshold level of the queue is reached, the ixAtmdAccRxDispatch() is invoked to remove all entries from the queue.

### See also:

ixAtmdAccRxDispatch, IxAtmdAccRxDispatcher

### Parameters

***queueId*** — (in) RX queue identification.

***callback*** — (in) function triggering the delivery of incoming traffic. This parameter cannot be a null pointer.

#### Returns

- IX_SUCCESS — Successful call to ixAtmdAccRxDispatcherRegister().
- IX_FAIL — error in the parameters, or there is an already active RX VC for this queue or some unspecified internal error occurred.

### A.3.4.8.10 ixAtmdAccRxLevelQuery (..)

ixAtmdAccRxLevelQuery (IxAtmRxQueueId rxQueueId, unsigned int * numberOfPdusPtr)

Query the number of entries in a particular RX queue.

This function is used to retrieve the number of pdus received by the hardware and ready for distribution to users.

#### Parameters

***rxQueueId*** — (in) indicates which of two RX queues to query.

***numberOfPdusPtr*** — (out) Pointer to store the number of available PDUs in the RX queue. This parameter cannot be a null pointer.

#### Returns

- IX_SUCCESS — the value in numberOfPdusPtr specifies the number of incoming pdus waiting in this queue
- IX_FAIL — an error occurs during processing. The value in numberOfPdusPtr is unspecified.

#### Note

This function is reentrant, doesn't use system resources and can be used from an interrupt context.

### A.3.4.8.11 ixAtmdAccRxQueueSizeQuery (..)

ixAtmdAccRxQueueSizeQuery (IxAtmRxQueueId rxQueueId, unsigned int * numberOfPdusPtr)

Query the size of a particular RX queue.

This function is used to retrieve the number of pdus the system is able to queue when reception is complete.

#### Parameters

***rxQueueId*** — (in) indicates which of two RX queues to query.

***numberOfPdusPtr*** — (out) Pointer to store the number of pdus the system is able to queue in the RX queue. This parameter cannot be a null pointer.

#### Returns

- IX_SUCCESS — the value in numberOfPdusPtr specifies the number of pdus the system is able to queue.
- IX_FAIL — an error occurs during processing. The value in numberOfPdusPtr is unspecified.

### Note

This function is reentrant, doesn't use system resources and can be used from an interrupt context.

## A.3.4.8.12 ixAtmdAccTxDoneDispatch (..)

ixAtmdAccTxDoneDispatch (unsigned int numberOfPdusToProcess, unsigned int * numberOfPdusProcessedPtr)

Process a number of pending transmit done pdus from the hardware.

As a by-product of Atm transmit operation buffers which transmission is complete need to be recycled to users. This function is invoked to service the outstanding list of transmitted buffers and pass them to VC users.

Users are handed back pdus by invoking the free callback registered during the ixAtmdAccTxVcConnect() call.

There is a single Tx done stream servicing all active Atm Tx ports which can contain a maximum of 64 entries. If this stream fills port transmission will stop so this function must be call sufficiently frequently to ensure no disruption to the transmit operation.

This function can be used from a timer context, or can be associated with a TxDone level threshold event (see ixAtmdAccTxDoneDispatcherRegister() ), or can be used inside an active polling mechanism under user control.

For ease of use the signature of this function is compatible with the TxDone threshold event callback prototype.

This functions can be used inside an interrupt context.

### See also:

ixAtmdAccTxDoneDispatcherRegister, IxAtmdAccTxVcBufferReturnCallback, ixAtmdAccTxDoneLevelQuery

### Parameters

***numberOfPdusToProcess*** — (in) maximum number of pdus to remove from the TX Done queue.

***numberOfPdusProcessedPtr*** — (out) number of pdus removed from the TX Done queue. This parameter cannot be a null pointer.

### Returns

- IX_SUCCESS — the number of pdus as indicated in numberOfPdusToProcess are removed from the TX Done hardware and passed to the user through the Tx Done callback registered during a call to ixAtmdAccTxVcConnect().
- IX_FAIL — invalid parameters or numberOfPdusProcessedPtr is a null pointer or some unspecified internal error occurred.

## A.3.4.8.13 ixAtmdAccTxDoneDispatcherRegister (..)

ixAtmdAccTxDoneDispatcherRegister (unsigned int numberOfPdus, IxAtmdAccTxDoneDispatcher notificationCallback)

Configure the Tx Done stream threshold value and register a callback to handle threshold notifications.

This function sets the threshold level in term of number of pdus at which the supplied notification function should be called.

The higher the threshold value is, the less events will be necessary to process transmitted buffers.

Transmitted buffers recycling implementation is a system-wide mechanism and needs to be set prior any traffic is started. If this threshold mechanism is not used, the user is responsible for polling the transmitted buffers thanks to ixAtmdAccTxDoneDispatch() and ixAtmdAccTxDoneLevelQuery() functions.

This function should be called during system initialization outside an interrupt context.

#### See also:

ixAtmdAccTxDoneDispatcherRegister, ixAtmdAccTxDoneDispatch, ixAtmdAccTxDoneLevelQuery

#### Parameters

***numberOfPdus*** — (in) The number of TxDone pdus which triggers the callback invocation This number has to be a power of 2, one of the values 0,1,2,4,8,16,32... The maximum value cannot be more than half of the txDone queue size (which can be retrieved using *ixAtmdAccTxDoneQueueSizeQuery())*

***notificationCallback*** — (in) The function to invoke. (This parameter can be *ixAtmdAccTxDoneDispatch()).This* parameter must not be a null pointer.

#### Returns

- IX_SUCCESS — Successful call to ixAtmdAccTxDoneDispatcherRegister.
- IX_FAIL — Error in the parameters.

#### Notes

The notificationCallback will be called exactly when the threshold level will increase from (numberOfPdus) to (numberOfPdus+1).

If there is no Tx traffic, there is no guarantee that TxDone Pdus will be released to the user (when txDone level is permanently under the threshold level. One of the preferred way to return resources to the user is to use a mix of txDone notifications, used together with a slow rate timer and an exclusion mechanism protecting from re-entrancy.

The TxDone threshold will only hand back buffers when the threshold level is crossed. Setting this threshold to a great number reduce the interrupt rate and the CPU load, but also increase the number of outstanding mbufs and has a system wide impact when these mbufs are needed by other components.

### A.3.4.8.14 ixAtmdAccTxDoneLevelQuery (..)

ixAtmdAccTxDoneLevelQuery (unsigned int * numberOfPdusPtr)

Query the current number of transmit pdus ready for recycling.

This function is used to get the number of transmitted pdus which the hardware is ready to hand back to user.

This function can be used from a timer context, or can be associated with a threshold event, on can be used inside an active polling mechanism

**See also:**

ixAtmdAccTxDoneDispatch

**Parameters**

***numberOfPdusPtr*** — (out) Pointer to the number of pdus transmitted at the time of this function call, and ready for recycling This parameter cannot be a null pointer.

**Returns**

- IX_SUCCESS — numberOfPdusPtr contains the number of pdus ready for recycling at the time of this function call.

**Returns**

- IX_FAIL — wrong parameter (null pointer as parameter).or unspecified processing error occurs. The value in numberOfPdusPtr is unspecified.

### A.3.4.8.15 ixAtmdAccTxDoneQueueSizeQuery (..)

ixAtmdAccTxDoneQueueSizeQuery (unsigned int * numberOfPdusPtr)

Query the TxDone queue size.

This function is used to get the number of pdus which the hardware is able to store after transmission is complete.

The returned value can be used to set a threshold and enable a callback to be notified when the number of pdus is going over the threshold.

**See also:**

ixAtmdAccTxDoneDispatcherRegister

**Parameters**

***numberOfPdusPtr*** — (out) Pointer to the number of pdus the system is able to queue after transmission.

**Returns**

- IX_SUCCESS — numberOfPdusPtr contains the number of pdus the system is able to queue after transmission.
- IX_FAIL — wrong parameter (null pointer as parameter).or unspecified processing error occurs. The value in numberOfPdusPtr is unspecified.

**Note**

This function is reentrant, doesn't use system resources and can be used from an interrupt context.

### A.3.4.8.16 ixAtmdAccUtopiaConfigSet (..)

ixAtmdAccUtopiaConfigSet (const IxAtmdAccUtopiaConfig * ixAtmdAccUtopiaConfigPtr)

Send the configuration structure to the UTOPIA interface.

This function downloads the IxAtmdAccUtopiaConfig structure to the UTOPIA and has the following effects:

- Setup the UTOPIA interface.
- Initialize the NPE.
- Reset the UTOPIA cell counters and status registers to known values.

This action has to be done once at initialization. A lock is preventing the concurrent use of ixAtmdAccUtopiaStatusGet() and @A ixAtmdAccUtopiaConfigSet().

#### Parameters

***ixAtmdAccNPEConfigPtr*** — (in) pointer to a structure to download to UTOPIA. This parameter cannot be a null pointer.

#### Returns

- IX_SUCCESS — successful download
- IX_FAIL — error in the parameters, or configuration is not complete or failed

#### See also:

ixAtmdAccUtopiaStatusGet

### A.3.4.8.17 ixAtmdAccUtopiaStatusGet (..)

ixAtmdAccUtopiaStatusGet (IxAtmdAccUtopiaStatus * ixAtmdAccUtopiaStatus)

Get the UTOPIA interface configuration.

This function reads the UTOPIA registers and the Cell counts and fills the IxAtmdAccUtopiaStatus structure

A lock is preventing the concurrent use of ixAtmdAccUtopiaStatusGet() and @A ixAtmdAccUtopiaConfigSet()

#### Parameters

***ixAttmdAccUtopiaStatus*** — (out) pointer to structure to be updated from internal hardware counters. This parameter cannot be a NULL pointer.

#### Returns

- IX_SUCCESS — successful read.
- IX_FAIL — error in the parameters null pointer, or configuration read is not complete or failed.

**See also:**

ixAtmdAccUtopiaConfigSet

## A.3.5 ATM Driver Access (IxAtmdAcc) UTOPIA Control API

The public API for the Intel® IXP42X product line ATM Driver Control component.

### A.3.5.1 Data Structures

- struct IxAtmdAccUtopiaConfig
  UTOPIA configuration.
- struct IxAtmdAccUtopiaConfig
  UTOPIA configuration.
- struct IxAtmdAccUtopiaConfig::UtTxConfig_
  UTOPIA Tx Config Register.
- struct IxAtmdAccUtopiaConfig::UtTxStatsConfig_
  UTOPIA Tx stats Register.
- struct IxAtmdAccUtopiaConfig::UtTxDefineIdle_
  UTOPIA Tx idle cells Register.
- struct IxAtmdAccUtopiaConfig::UtTxEnableFields_
  UTOPIA Tx ienable fields Register.
- struct IxAtmdAccUtopiaConfig::UtTxTransTable0_
  UTOPIA Tx translation table Register.
- struct IxAtmdAccUtopiaConfig::UtTxTransTable1_
  UTOPIA Tx translation table Register.
- struct IxAtmdAccUtopiaConfig::UtTxTransTable2_
  UTOPIA Tx translation table Register.
- struct IxAtmdAccUtopiaConfig::UtTxTransTable3_
  UTOPIA Tx translation table Register.
- struct IxAtmdAccUtopiaConfig::UtTxTransTable4_
  UTOPIA Tx translation table Register.
- struct IxAtmdAccUtopiaConfig::UtTxTransTable5_
  UTOPIA Tx translation table Register.
- struct IxAtmdAccUtopiaConfig::UtRxConfig_
  UTOPIA Rx config Register.
- struct IxAtmdAccUtopiaConfig::UtRxStatsConfig_
  Utopia Rx stats config Register.
- struct IxAtmdAccUtopiaConfig::UtRxDefineIdle_
  UTOPIA Rx idle cells config Register.

- struct IxAtmdAccUtopiaConfig::UtRxEnableFields_
  UTOPIA Rx enable Register.
- struct IxAtmdAccUtopiaConfig::UtRxTransTable0_
  UTOPIA Rx translation table Register.
- struct IxAtmdAccUtopiaConfig::UtRxTransTable1_
  UTOPIA Rx translation table Register.
- struct IxAtmdAccUtopiaConfig::UtRxTransTable2_
  UTOPIA Rx translation table Register.
- struct IxAtmdAccUtopiaConfig::UtRxTransTable3_
  UTOPIA Rx translation table Register.
- struct IxAtmdAccUtopiaConfig::UtRxTransTable4_
  UTOPIA Rx translation table Register.
- struct IxAtmdAccUtopiaConfig::UtRxTransTable5_
  UTOPIA Rx translation table Register.
- struct IxAtmdAccUtopiaConfig::UtSysConfig_
  NPE setup Register.
- struct IxAtmdAccUtopiaStatus
  UTOPIA status.
- struct IxAtmdAccUtopiaStatus
  UTOPIA status.
- struct IxAtmdAccUtopiaStatus::UtTxCellConditionStatus_
  UTOPIA Tx Status Register.
- struct IxAtmdAccUtopiaStatus::UtRxCellConditionStatus_
  UTOPIA Rx Status Register.

### A.3.5.2 Detailed Description

The public API for the Intel® IXP42X product line ATM Driver Control component.

IxAtmdAcc is the low level interface by which AAL PDU get transmitted to, and received from the UTOPIA bus.

This part is related to the UTOPIA configuration.

## A.3.6 ATM Manager (IxAtmm) API

The Intel® IXP42X product line demo ATM Manager component Public API.

### A.3.6.1 Data Structures

- struct IxAtmmPortCfg

  Structure contains port-specific information required to initialize IxAtmm, and specifically, the Intel® IXP42X product line UTOPIA Level-2 device.

- struct IxAtmmVc

  This structure describes the required attributes of a virtual connection.

### A.3.6.2 Defines

- #define IX_ATMM_RET_ALREADY_INITIALIZED 2

  Component has already been initialized.

- #define IX_ATMM_RET_INVALID_PORT 3

  Specified port does not exist or is out of range.

- #define IX_ATMM_RET_INVALID_VC_DESCRIPTOR 4

  The VC description does not adhere to ATM standards.

- #define IX_ATMM_RET_VC_CONFLICT 5

  The VPI/VCI values supplied are either reserved, or they conflict with a previously registered VC on this port.

- #define IX_ATMM_RET_PORT_CAPACITY_IS_FULL 6

  The virtual connection cannot be established on the port because the remaining port capacity is not sufficient to support it.

- #define IX_ATMM_RET_NO_SUCH_VC 7

  No registered VC, as described by the supplied VCI/VPI or VC identifier values, exists on this port.

- #define IX_ATMM_RET_INVALID_VC_ID 8

  The specified VC identifier is out of range.

- #define IX_ATMM_RET_INVALID_PARAM_PTR 9

  A pointer parameter was NULL.

- #define IX_ATMM_UTOPIA_SPHY_ADDR 31

  The phy address when in SPHY mode.

### A.3.6.3 Typedefs

- typedef void(* IxAtmmVcChangeCallback )(IxAtmmVcChangeEvent eventType, IxAtmLogicalPort port, const IxAtmmVc *vcChanged)

  Callback type used with ixAtmmVcChangeCallbackRegister interface Defines a callback type which will be used to notify registered users of registration/deregistration events on a particular port.

### A.3.6.4 Enumerations

- enum IxAtmmVcDirection { IX_ATMM_VC_DIRECTION_TX = 0, IX_ATMM_VC_DIRECTION_RX, IX_ATMM_VC_DIRECTION_INVALID }

  Definition for use in the IxAtmmVc structure. Indicates the direction of a VC.

- enum IxAtmmVcChangeEvent { IX_ATMM_VC_CHANGE_EVENT_REGISTER = 0, IX_ATMM_VC_CHANGE_EVENT_DEREGISTER, IX_ATMM_VC_CHANGE_EVENT_INVALID }

  Definition for use with IxAtmmVcChangeCallback callback. Indicates that the event type represented by the callback for this VC.

- enum IxAtmmUtopiaLoopbackMode { IX_ATMM_UTOPIA_LOOPBACK_DISABLED = 0, IX_ATMM_UTOPIA_LOOPBACK_ENABLED, IX_ATMM_UTOPIA_LOOPBACK_INVALID }

  Definitions for use with ixAtmmUTOPIAInit interface to indicate that UTOPIA loopback should be enabled or disabled on initialization.

- enum IxAtmmPhyMode { IX_ATMM_MPHY_MODE = 0, IX_ATMM_SPHY_MODE, IX_ATMM_PHY_MODE_INVALID }

  Definitions for use with ixAtmmUtopiaInit interface to indicate that UTOPIA multi-phy/ single-phy mode is used.

### A.3.6.5 Functions

- IX_STATUS ixAtmmInit (void)

  Interface to initialize the IxAtmm software component. Can be called once only.

- IX_STATUS ixAtmmUtopiaInit (unsigned numPorts, IxAtmmPhyMode phyMode, IxAtmmPortCfg portCfgs[], IxAtmmUtopiaLoopbackMode loopbackMode)

  Interface to initialize the UTOPIA Level-2 ATM coprocessor for the specified number of physical ports. The function must be called before the ixAtmmPortInitialize interface can operate successfully.

- IX_STATUS ixAtmmPortInitialize (IxAtmLogicalPort port, unsigned txPortRate, unsigned rxPortRate)

  The interface is called following ixAtmmUtopiaInit () and before calls to any other IxAtmm interface. It serves to activate the registered ATM port with IxAtmm.

- IX_STATUS ixAtmmPortModify (IxAtmLogicalPort port, unsigned txPortRate, unsigned rxPortRate)

  A client may call this interface to change the existing port rate (expressed in bits/second) on an established ATM port.

- IX_STATUS ixAtmmPortQuery (IxAtmLogicalPort port, unsigned *txPortRate, unsigned *rxPortRate)

  The client may call this interface to request details on currently registered transmit and receive rates for an ATM port.

- IX_STATUS ixAtmmPortEnable (IxAtmLogicalPort port)

  The client call this interface to enable transmit for an ATM port. At initialization, all the ports are disabled.

- IX_STATUS ixAtmmPortDisable (IxAtmLogicalPort port)

  The client call this interface to disable transmit for an ATM port. At initialization, all the ports are disabled.

- IX_STATUS ixAtmmVcRegister (IxAtmLogicalPort port, IxAtmmVc *vcToAdd, IxAtmSchedulerVcId *vcId)

  This interface is used to register an ATM Virtual Connection on the specified ATM port.

- IX_STATUS ixAtmmVcDeregister (IxAtmLogicalPort port, IxAtmSchedulerVcId vcId)

  Function called by a client to deregister a VC from the system.

- IX_STATUS ixAtmmVcQuery (IxAtmLogicalPort port, unsigned vpi, unsigned vci, IxAtmmVcDirection direction, IxAtmSchedulerVcId *vcId, IxAtmmVc *vcDesc)

  This interface supplies information about an active VC on a particular port when supplied with the VPI, VCI and direction of that VC.

- IX_STATUS ixAtmmVcIdQuery (IxAtmLogicalPort port, IxAtmSchedulerVcId vcId, IxAtmmVc *vcDesc)

  This interface supplies information about an active VC on a particular port when supplied with a vcId for that VC.

- IX_STATUS ixAtmmVcChangeCallbackRegister (IxAtmmVcChangeCallback callback)

  This interface is invoked to supply a function to IxAtmm which will be called to notify the client if a new VC is registered with IxAtmm or an existing VC is removed.

- IX_STATUS ixAtmmVcChangeCallbackDeregister (IxAtmmVcChangeCallback callback)

  This interface is invoked to deregister a previously supplied callback function.

- IX_STATUS ixAtmmUtopiaStatusShow (void)

  Display utopia status counters.

- IX_STATUS ixAtmmUtopiaCfgShow (void)

  Display utopia information(config registers and status registers).

### A.3.6.6 Detailed Description

The Intel® IXP42X product line demo ATM Manager component Public API.

### A.3.6.7 Typedef Documentation

#### A.3.6.7.1 typedef void(..)

typedef void(* IxAtmmVcChangeCallback)(IxAtmmVcChangeEvent eventType, IxAtmLogicalPort port, const IxAtmmVc* vcChanged)

Callback type used with ixAtmmVcChangeCallbackRegister interface Defines a callback type which will be used to notify registered users of registration/deregistration events on a particular port.

#### Parameters

***IxAtmmVcChangeEvent eventType*** — Event indicating whether the VC supplied has been added or removed.

***IxAtmLogicalPort port*** — Specifies the port on which the event has occurred.

***IxAtmmVc* vcChanged*** — Pointer to a structure which gives details of the VC which has been added or removed on the port

### A.3.6.8 Function Documentation

#### A.3.6.8.1 ixAtmmInit (void)

Interface to initialize the IxAtmm software component. Can be called once only.

Must be called before any other IxAtmm API is called.

#### Parameters

None

#### Returns

- IX_SUCCESS — IxAtmm has been successfully initialized. Calls to other IxAtmm interfaces may now be performed.
- IX_FAIL — IxAtmm has already been initialized.

#### A.3.6.8.2 ixAtmmPortDisable (..)

ixAtmmPortDisable (IxAtmLogicalPort port)

The client call this interface to disable transmit for an ATM port. At initialization, all the ports are disabled.

#### Parameters

***IxAtmLogicalPort port*** — Value identifies the port.

#### Returns

- IX_SUCCESS — Transmission over this port is stopped.
- IX_FAIL — The port parameter is not valid, or the port is already disabled.

#### Notes

When a port is disabled, Rx and Tx VC Connect requests will fail.

This function call does not stop RX traffic. It is supposed that this function is invoked when a serious problem is detected (e.g. physical layer broken). Then, the RX traffic is not passing.

This function is blocking until the hw acknowledge that the transmission is stopped.

This function uses system resources and should not be used inside an interrupt context.

### See also:

ixAtmmPortEnable

#### A.3.6.8.3 ixAtmmPortEnable (..)

ixAtmmPortEnable (IxAtmLogicalPort port)

The client call this interface to enable transmit for an ATM port. At initialization, all the ports are disabled.

### Parameters

***IxAtmLogicalPort port*** — Value identifies the port.

### Returns

- IX_SUCCESS — Transmission over this port is started.

### Returns

- IX_FAIL — The port parameter is not valid, or the port is already enabled.

### Notes

When a port is disabled, Rx and Tx VC Connect requests will fail.

This function uses system resources and should not be used inside an interrupt context.

### See also:

ixAtmmPortDisable

#### A.3.6.8.4 ixAtmmPortInitialize (..)

ixAtmmPortInitialize (IxAtmLogicalPort port, unsigned txPortRate, unsigned rxPortRate)

The interface is called following ixAtmmUtopiaInit () and before calls to any other IxAtmm interface. It serves to activate the registered ATM port with IxAtmm.

The transmit and receive port rates are specified in bits per second. This translates to ATM cells per second according to the following formula: CellsPerSecond = portRate / (53*8) The Intel® IXP42X product line device supports only 53 byte cells. The client shall make sure that the off-chip physical layer device has already been initialized.

IxAtmm will configure IxAtmdAcc and IxAtmSch to enable scheduling on the port.

This interface must be called once for each active port in the system. The first time the interface is invoked, it will configure the mechanism by which the handling of transmit, transmit-done and receive are driven with the IxAtmdAcc component.

This function is reentrant.

### Note

The minimum tx rate that will be accepted is 424 bit/s which equates to 1 cell (53 bytes) per second.

### Parameters

***IxAtmLogicalPort port*** — Identifies the port which is to be initialized.

***unsigned txPortRate*** — Value specifies the transmit port rate for this port in bits/second. This value is used by the ATM Scheduler component is evaluating VC access requests for the port.

***unsigned rxPortRate*** — Value specifies the receive port rate for this port in bits/second.

### Returns

- IX_SUCCESS — The specified ATM port has been successfully initialized. IxAtmm is ready to accept VC registrations on this port.
- IX_ATMM_RET_ALREADY_INITIALIZED — ixAtmmPortInitialize has already been called successfully on this port. The current call is rejected.
- IX_ATMM_RET_INVALID_PORT — The port value indicated in the input is not valid. The request is rejected.
- IX_FAIL — IxAtmm could not initialize the port because the inputs are not understood.

### See also:

ixAtmmPortEnable, ixAtmmPortDisable

## A.3.6.8.5 ixAtmmPortModify (..)

ixAtmmPortModify (IxAtmLogicalPort port, unsigned txPortRate, unsigned rxPortRate)

A client may call this interface to change the existing port rate (expressed in bits/second) on an established ATM port.

### Parameters

***IxAtmLogicalPort port*** — Identifies the port which is to be initialized.

***unsigned txPortRate*** — Value specifies the`` transmit port rate for this port in bits/second. This value is used by the ATM Scheduler component is evaluating VC access requests for the port.

***unsigned rxPortRate*** — Value specifies the receive port rate for this port in bits/second.

### Returns

- IX_SUCCESS — The indicated ATM port rates have been successfully modified.
- IX_ATMM_RET_INVALID_PORT — The port value indicated in the input is not valid. The request is rejected.
- IX_FAIL — IxAtmm could not update the port because the inputs are not understood, or the interface was called before the port was initialized.

## A.3.6.8.6 ixAtmmPortQuery (..)

ixAtmmPortQuery (IxAtmLogicalPort port, unsigned * txPortRate, unsigned * rxPortRate)

The client may call this interface to request details on currently registered transmit and receive rates for an ATM port.

### Parameters

***IxAtmLogicalPort port*** — Value identifies the port from which the rate details are requested.

***OUT unsigned *txPortRate*** — Pointer to a value which will be filled with the value of the transmit port rate specified in bits/second.

***OUT unsigned *rxPortRate*** — Pointer to a value which will be filled with the value of the receive port rate specified in bits/second.

### Returns

- IX_SUCCESS — The information requested on the specified port has been successfully supplied in the output.
- IX_ATMM_RET_INVALID_PORT — The port value indicated in the input is not valid. The request is rejected.
- IX_ATMM_RET_INVALID_PARAM_PTR — A pointer parameter was NULL.
- IX_FAIL : IxAtmm could not update the port because the inputs are not understood, or the interface was called before the port was initialized.
- ixAtmmUtopiaCfgShow (void) — Display utopia information(config registers and status registers).

### Parameters

None

### Returns

- IX_SUCCESS — Show function was successful
- IX_FAIL — Internal failure

## A.3.6.8.7 ixAtmmUtopiaInit (..)

ixAtmmUtopiaInit (unsigned numPorts, IxAtmmPhyMode phyMode, IxAtmmPortCfg portCfgs[], IxAtmmUtopiaLoopbackMode loopbackMode

Interface to initialize the UTOPIA Level-2 ATM coprocessor for the specified number of physical ports. The function must be called before the ixAtmmPortInitialize interface can operate successfully.

### Parameters

***unsigned numPorts*** — Indicates the total number of logical ports that are active on the device. Up to 8 ports are supported.

***IxAtmmPhyMode phyMode*** — Put the UTOPIA coprocessor in SPHY or MPHY mode.

**IxAtmmPortCfg *portCfgs[]*** — Pointer to an array of elements detailing the UTOPIA specific port characteristics. The length of the array must be equal to the number of ports activated. ATM ports are referred to by the relevant offset in this array in all subsequent IxAtmm interface calls.

***IxAtmmUtopiaLoopbackMode loopbackMode*** — Value must be one of IX_ATMM_UTOPIA_LOOPBACK_ENABLED or IX_ATMM_UTOPIA_LOOPBACK_DISABLED indicating whether loopback should be enabled on the device. Loopback can only be supported on a single PHY, therefore the numPorts parameter must be 1 if loopback is enabled.

#### Returns

- IX_SUCCESS — Indicates that the UTOPIA device has been successfully initialized for the supplied ports.
- IX_ATMM_RET_ALREADY_INITIALIZED — The UTOPIA device has already been initialized.
- IX_FAIL — Supplied parameters are invalid or have been rejected by the UTOPIA-NPE device.

***Warning:*** This interface may only be called once. NB * : Port identifiers are assumed to range from 0 to (numPorts - 1) in all instances. In all subsequent calls to interfaces supplied by IxAtmm, the specified port value is expected to represent the offset in the portCfgs array specified in this interface. i.e. The first port in this array will subsequently be represented as port 0, the second port as port 1, and so on.

### A.3.6.8.8 ixAtmmUtopiaStatusShow (void)

Display utopia status counters.

#### Parameters

None

#### Returns

- IX_SUCCESS — Show function was successful.
- IX_FAIL — Internal failure.

### A.3.6.8.9 ixAtmmVcChangeCallbackDeregister (..)

ixAtmmVcChangeCallbackDeregister (IxAtmmVcChangeCallback callback)

This interface is invoked to deregister a previously supplied callback function.

#### Parameters

***IxAtmmVcChangeCallback callback*** — Callback which complies with the IxAtmmVcChangeCallback definition. This function will removed from the table of callbacks.

#### Returns

- IX_SUCCESS — The specified callback has been deregistered successfully from IxAtmm.
- IX_FAIL — Either the supplied callback is invalid, or is not currently registered with IxAtmm.

### A.3.6.8.10 ixAtmmVcChangeCallbackRegister (..)

ixAtmmVcChangeCallbackRegister (IxAtmmVcChangeCallback callback)

This interface is invoked to supply a function to IxAtmm which will be called to notify the client if a new VC is registered with IxAtmm or an existing VC is removed.

The callback, when invoked, will run within the context of the call to ixAtmmVcRegister or ixAtmmVcDeregister which caused the change of state.

A maximum of 32 callbacks may be registered in with IxAtmm.

#### Parameters

***IxAtmmVcChangeCallback callback*** — Callback which complies with the IxAtmmVcChangeCallback definition. This function will be invoked by IxAtmm with the appropriate parameters for the relevant VC when any VC has been registered or deregistered with IxAtmm.

#### Returns

- IX_SUCCESS — The specified callback has been registered successfully with IxAtmm and will be invoked when appropriate.
- IX_FAIL — Either the supplied callback is invalid, or IxAtmm has already registered 32 and cannot accommodate any further registrations of this type. The request is rejected.

***Warning:*** The client must not call either the ixAtmmVcRegister or ixAtmmVcDeregister interfaces from within the supplied callback function.

### A.3.6.8.11 ixAtmmVcDeregister (..)

ixAtmmVcDeregister (IxAtmLogicalPort port, IxAtmSchedulerVcId vcId)

Function called by a client to deregister a VC from the system.

With the removal of each new VC from a port, a series of registered callback functions are invoked by the IxAtmm component to notify possible external components of the change. The callback functions are registered using the ixAtmmVcChangeCallbackRegister.

The IxAtmSch component is notified of the removal of transmit VCs.

#### Parameters

***IxAtmLogicalPort port*** — Identifies port on which the VC to be removed is currently registered.

***IxAtmSchedulerVcId vcId*** — VC identifier value of the VC to be deregistered. This value was supplied to the client when the VC was originally registered. This value can also be queried from the IxAtmm component through the ixAtmmVcQuery interface.

### Returns

- IX_SUCCESS — The specified VC has been successfully removed from this port.
- IX_ATMM_RET_INVALID_PORT — The port value indicated in the input is not valid or has not been initialized. The request is rejected.
- IX_FAIL — There is no registered VC associated with the supplied identifier registered on this port.

## A.3.6.8.12 ixAtmmVcIdQuery (..)

ixAtmmVcIdQuery (IxAtmLogicalPort port, IxAtmSchedulerVcId vcId, IxAtmmVc * vcDesc)

This interface supplies information about an active VC on a particular port when supplied with a vcId for that VC.

### Parameters

***IxAtmLogicalPort port*** — Identifies port on which the VC to be queried is currently registered.

***IxAtmSchedulerVcId vcId*** — Value returned by ixAtmmVcRegister which uniquely identifies the requested VC on this port.

OUT IxAtmmVc *vcDesc — Pointer to an IxAtmmVc structure which will be filled with the specific details of the requested VC, if it exists on this port.

### Returns

- IX_SUCCESS — The specified VC has been found on this port and the requested details have been returned.
- IX_ATMM_RET_INVALID_PORT — The port value indicated in the input is not valid or has not been initialized. The request is rejected.
- IX_ATMM_RET_NO_SUCH_VC — No VC exists on the specified port which matches the supplied identifier. No data is returned.
- IX_ATMM_RET_INVALID_PARAM_PTR — A pointer parameter was NULL.

## A.3.6.8.13 ixAtmmVcQuery (..)

ixAtmmVcQuery (IxAtmLogicalPort port, unsigned vpi, unsigned vci, IxAtmmVcDirection direction, IxAtmSchedulerVcId * vcId, IxAtmmVc * vcDesc)

This interface supplies information about an active VC on a particular port when supplied with the VPI, VCI and direction of that VC.

### Parameters

***IxAtmLogicalPort port*** — Identifies port on which the VC to be queried is currently registered.

***unsigned vpi*** — ATM VPI value of the requested VC.

***unsigned vci*** — ATM VCI value of the requested VC.

***IxAtmmVcDirection direction*** — One of IX_ATMM_VC_DIRECTION_TX or IX_ATMM_VC_DIRECTION_RX indicating the direction (Tx or Rx) of the requested VC.

***OUT IxAtmSchedulerVcId *vcId*** — Pointer to an integer value which will be filled with the VC identifier value for the requested VC (as returned by ixAtmmVcRegister), if it exists on this port.

***OUT IxAtmmVc *vcDesc*** — Pointer to an IxAtmmVc structure which will be filled with the specific details of the requested VC, if it exists on this port.

#### Returns

- IX_SUCCESS — The specified VC has been found on this port and the requested details have been returned.
- IX_ATMM_RET_INVALID_PORT — The port value indicated in the input is not valid or has not been initialized. The request is rejected.
- IX_ATMM_RET_NO_SUCH_VC — No VC exists on the specified port which matches the search criteria (VPI, VCI, direction) given. No data is returned.
- IX_ATMM_RET_INVALID_PARAM_PTR — A pointer parameter was NULL.

### A.3.6.8.14 ixAtmmVcRegister (..)

ixAtmmVcRegister (IxAtmLogicalPort port, IxAtmmVc * vcToAdd, IxAtmSchedulerVcId * vcId)

This interface is used to register an ATM Virtual Connection on the specified ATM port.

Each call to this interface registers a unidirectional virtual connection with the parameters specified. If a bi-directional VC is needed, the function should be called twice (once for each direction, Tx & Rx) where the VPI and VCI and port parameters in each call are identical.

With the addition of each new VC to a port, a series of callback functions are invoked by the IxAtmm component to notify possible external components of the change. The callback functions are registered using the ixAtmmVcChangeCallbackRegister interface.

The IxAtmSch component is notified of the registration of transmit VCs.

#### Parameters

***IxAtmLogicalPort port*** — Identifies port on which the specified VC is to be registered.

IxAtmmVc *vcToAdd — Pointer to an IxAtmmVc structure containing a description of the VC to be registered. The client shall fill the vpi, vci and direction and relevant trafficDesc members of this structure before calling this function.

***OUT IxAtmSchedulerVcId *vcId*** — Pointer to an integer value which is filled with the per-port unique identifier value for this VC. This identifier will be required when a request is made to deregister or change this VC. VC identifiers for transmit VCs will have a value between 0-43, i.e. 32 data Tx VCs + 12 OAM Tx Port VCs. Receive VCs will have a value between 44-66, i.e. 32 data Rx VCs + 1 OAM Rx VC.

#### Returns

- IX_SUCCESS — The VC has been successfully registered on this port. The VC is ready for a client to configure IxAtmdAcc for receive and transmit operations on the VC.
- IX_ATMM_RET_INVALID_PORT — The port value indicated in the input is not valid or has not been initialized. The request is rejected.
- IX_ATMM_RET_INVALID_VC_DESCRIPTOR — The descriptor pointed to by vcToAdd is invalid. The registration request is rejected.

- IX_ATMM_RET_VC_CONFLICT — The VC requested conflicts with reserved VPI and/or VCI values or with another VC already activated on this port.
- IX_ATMM_RET_PORT_CAPACITY_IS_FULL — The VC cannot be registered in the port because the port capacity is insufficient to support the requested ATM traffic contract. The registration request is rejected.
- IX_ATMM_RET_INVALID_PARAM_PTR — A pointer parameter was NULL.

***Warning:*** IxAtmm has no capability of signaling or negotiating a virtual connection. Negotiation of the admission of the VC to the network is beyond the scope of this function. This is assumed to be performed by the calling client, if appropriate, before or after this function is called.

## A.3.7 ATM Transmit Scheduler (IxAtmSch) API

The Intel® IXP42X product line demo ATM scheduler component Public API.

### A.3.7.1 Defines

- #define IX_ATMSCH_RET_NOT_ADMITTED 2

  Indicates that CAC function has rejected VC registration due to insufficient line capacity.
- #define IX_ATMSCH_RET_QUEUE_FULL 3

  Indicates that the VC queue is full, no more demand can be queued at this time.
- #define IX_ATMSCH_RET_QUEUE_EMPTY 4

  Indicates that all VC queues on this port are empty and therefore there are no cells to be scheduled at this time.

### A.3.7.2 Functions

- IX_STATUS ixAtmSchInit (void)

  This function is used to initialize the ixAtmSch component. It should be called before any other IxAtmSch API function.
- IX_STATUS ixAtmSchPortModelInitialize (IxAtmLogicalPort port, unsigned int portRate, unsigned int minCellsToSchedule)

  This function shall be called first to initialize an ATM port before any other ixAtmSch API calls may be made for that port.
- IX_STATUS ixAtmSchPortRateModify (IxAtmLogicalPort port, unsigned int portRate)

  This function is called to modify the portRate on a previously initialized port, typically in the event that the line condition of the port changes.
- IX_STATUS ixAtmSchVcModelSetup (IxAtmLogicalPort port, IxAtmTrafficDescriptor *trafficDesc, IxAtmSchedulerVcId *vcId)

  A client calls this interface to set up an upstream (transmitting) virtual connection model (VC) on the specified ATM port. This function also provides the virtual * connection admission control (CAC) service to the client.

- IX_STATUS ixAtmSchVcConnIdSet (IxAtmLogicalPort port, IxAtmSchedulerVcId vcId, IxAtmConnId vcUserConnId)

  A client calls this interface to set the vcUserConnId for a VC on the specified ATM port. This vcUserConnId will default to IX_ATM_IDLE_CELLS_CONNID if this function is not called for a VC. Hence if the client does not call this function for a VC then only idle cells will be scheduled for this VC.

- IX_STATUS ixAtmSchVcModelRemove (IxAtmLogicalPort port, IxAtmSchedulerVcId vcId)

  Interface called by the client to remove a previously established VC on a particular port.

- IX_STATUS ixAtmSchVcQueueUpdate (IxAtmLogicalPort port, IxAtmSchedulerVcId vcId, unsigned int numberOfCells)

  The client calls this function to notify IxAtmSch that the user of a VC has submitted cells for transmission.

- IX_STATUS ixAtmSchVcQueueClear (IxAtmLogicalPort port, IxAtmSchedulerVcId vcId)

  The client calls this function to remove all currently queued cells from a registered VC. The pending cell count for the specified VC is reset to zero.

- IX_STATUS ixAtmSchTableUpdate (IxAtmLogicalPort port, unsigned int maxCells, IxAtmScheduleTable **rettable)

  The client calls this function to request an update of the schedule table for a particular ATM port.

- void ixAtmSchShow (void)

  Utility function which will print statistics on the current and accumulated state of VCs and traffic in the ATM scheduler component. Output is sent to the default output device.

- void ixAtmSchStatsClear (void)

  Utility function which will reset all counter statistics in the ATM scheduler to zero.

## A.3.7.3 Detailed Description

The Intel® IXP42X product line demo ATM scheduler component Public API.

## A.3.7.4 Function Documentation

### A.3.7.4.1 ixAtmSchInit (void)

This function is used to initialize the ixAtmSch component. It should be called before any other IxAtmSch API function.

#### Parameters

None

#### Returns

- **IX_SUCCESS** — Indicates that:
  - — ATM scheduler component has been successfully initialized.
  - — Scheduler is ready to accept Port modelling requests.
- **IX_FAIL** — Some internal error has prevented the scheduler component from initializing.

#### A.3.7.4.2 ixAtmSchPortModelInitialize (..)

ixAtmSchPortModelInitialize (IxAtmLogicalPort port, unsigned int portRate, unsigned int minCellsToSchedule)

This function shall be called first to initialize an ATM port before any other ixAtmSch API calls may be made for that port.

##### Parameters

***IxAtmLogicalPort port*** — The specific port to initialize. Valid values range from 0 to 7, representing a maximum of 12 possible ports.

***unsigned int portRate*** — Value indicating the upstream capacity of the indicated port. The value should be supplied in units of ATM (53 byte) cells per second.

***unsigned int minCellsToSchedule*** — This parameter specifies the minimum number of cells which the scheduler will put in a schedule table for this port. This value sets the worst case CDVT for VCs on this port i.e. CDVT = 1*minCellsToSchedule/portRate.

##### Returns

- **IX_SUCCESS** — Indicates that:
  - — ATM scheduler has been successfully initialized.
  - — Requested port model has been established.
  - — Scheduler is ready to accept VC modelling requests on the ATM port.
- **IX_FAIL** — Indicates that the requested port could not be initialized.

#### A.3.7.4.3 ixAtmSchPortRateModify (..)

ixAtmSchPortRateModify (IxAtmLogicalPort port, unsigned int portRate)

This function is called to modify the portRate on a previously initialized port, typically in the event that the line condition of the port changes.

##### Parameters

***IxAtmLogicalPort port*** — Specifies the ATM port which is to be modified.

***unsigned int portRate*** — Value indicating the new upstream capacity for this port in cells/second.

##### Returns

- **IX_SUCCESS** — Port rate has been successfully modified.
- **IX_FAIL** — Port rate could not be modified, either because the input data was invalid, or the new port rate is insufficient to support established ATM VC contracts on this port.

***Warning:*** The IxAtmSch component will validate the supplied port rate is sufficient to support all established VC contracts on the port. If the new port rate is insufficient to support all established contracts then the request to modify the port rate will be rejected. In this event, the user is expected to remove established contracts using the ixAtmSchVcModelRemove interface and then retry this interface.

**See also:**

ixAtmSchVcModelRemove()

### A.3.7.4.4 ixAtmSchShow (void)

Utility function which will print statistics on the current and accumulated state of VCs and traffic in the ATM scheduler component. Output is sent to the default output device.

**Parameters**

None

**Returns**

None

### A.3.7.4.5 ixAtmSchStatsClear (void)

Utility function which will reset all counter statistics in the ATM scheduler to zero.

**Parameters**

None

**Returns**

None

### A.3.7.4.6 ixAtmSchTableUpdate (..)

ixAtmSchTableUpdate (IxAtmLogicalPort port, unsigned int maxCells, IxAtmScheduleTable ** rettable)

The client calls this function to request an update of the schedule table for a particular ATM port.

This is called when the client decides it needs a new sequence of cells to send (probably because the transmit queue is near to empty for this ATM port). The scheduler will use its stored information on the cells submitted for transmit (i.e. data supplied via ixAtmSchVcQueueUpdate function) with the traffic descriptor information of all established VCs on the ATM port to decide the sequence of cells to be sent and fill the schedule table for a period of time into the future.

IxAtmSch will guarantee a minimum of minCellsToSchedule if there is at least one cell ready to send. If there are no cells then IX_ATMSCH_RET_QUEUE_EMPTY is returned.

This implementation of ixAtmSchTableUpdate uses no operating system or external facilities, either directly or indirectly. This allows clients to call this function form within an FIQ interrupt handler.

**Parameters**

***IxAtmLogicalPort port*** — Specifies the ATM port for which requested schedule table is to be generated.

***unsigned maxCells*** — Specifies the maximum number of cells that must be scheduled in the supplied table during any call to the interface.

***OUT IxAtmScheduleTable **table*** — A pointer to an area of storage is returned which contains the generated schedule table. The client should not modify the contents of this table.

#### Returns

- **IX_SUCCESS** — Schedule table has been published. Currently there is at least one VC queue that is nonempty.
- **IX_ATMSCH_RET_QUEUE_EMPTY** — Currently all VC queues on this port are empty. The schedule table returned is set to NULL. The client is not expected to invoke this function again until more cells have been submitted on this port through the ixAtmSchVcQueueUpdate function.
- **IX_FAIL** — Input are invalid. No action is taken.

IxAtmSch assumes that the calling software ensures that calls to ixAtmSchVcQueueUpdate, ixAtmSchVcQueueClear and ixAtmSchTableUpdate are both self and mutually exclusive for the same port.

***Warning:*** Subsequent calls to this function for the same port will overwrite the contents of previously supplied schedule tables. The client must be completely finished with the previously supplied schedule table before calling this function again for the same port.

#### See also:

ixAtmSchVcQueueUpdate(), ixAtmSchVcQueueClear(), ixAtmSchTableUpdate().

### A.3.7.4.7 ixAtmSchVcConnIdSet (..)

ixAtmSchVcConnIdSet (IxAtmLogicalPort port, IxAtmSchedulerVcId vcId, IxAtmConnId vcUserConnId)

A client calls this interface to set the vcUserConnId for a VC on the specified ATM port. This vcUserConnId will default to IX_ATM_IDLE_CELLS_CONNID if this function is not called for a VC. Hence if the client does not call this function for a VC then only idle cells will be scheduled for this VC.

#### Parameters

***IxAtmLogicalPort port*** — Specifies the ATM port on which the upstream VC is has been established.

***IxAtmSchedulerVcId vcId*** — This is the unique identifier for this virtual connection. A valid identification is a non-negative number and is all ports.

***IxAtmConnId vcUserConnId*** — The connId is used to refer to a VC in schedule table entries. It is treated as the Id by which the scheduler client knows the VC. It is used in any communications from the Scheduler to the scheduler user e.g. schedule table entries.

#### Returns

- **IX_SUCCESS** — ID has successfully been set.
- **IX_FAIL** — Input data are invalid. connId id is not established.

### A.3.7.4.8 ixAtmSchVcModelRemove (..)

ixAtmSchVcModelRemove (IxAtmLogicalPort port, IxAtmSchedulerVcId vcId)

Interface called by the client to remove a previously established VC on a particular port.

#### Parameters

***IxAtmLogicalPort port*** — Specifies the ATM port on which the VC to be removed is established.

***IxAtmSchedulerVcId vcId*** — Identifies the VC to be removed. This is the value returned by the ixAtmSchVcModelSetup call which established the relevant VC.

#### Returns

- **IX_SUCCESS** — VC has been successfully removed from this port. It is no longer modelled on this port.
- **IX_FAIL** — Input data are invalid. The VC is still being modeled by the traffic shaper.

#### See also:

ixAtmSchVcModelSetup()

### A.3.7.4.9 ixAtmSchVcModelSetup (..)

ixAtmSchVcModelSetup (IxAtmLogicalPort port, IxAtmTrafficDescriptor * trafficDesc, IxAtmSchedulerVcId * vcId)

A client calls this interface to set up an upstream (transmitting) virtual connection model (VC) on the specified ATM port. This function also provides the virtual * connection admission control (CAC) service to the client.

#### Parameters

***IxAtmLogicalPort port*** — Specifies the ATM port on which the upstream VC is to be established.

IxAtmTrafficDescriptor *trafficDesc — Pointer to a structure describing the requested traffic contract of the VC to be established. This structure contains the typical ATM traffic descriptor values (e.g. PCR, SCR, MBS, CDVT, etc.) defined by the ATM standard.

***OUT IxAtmSchedulerVcId vcId*** — This value will be filled with the port-unique identifier for this virtual connection. A valid identification is a non-negative number.

#### Returns

- **IX_SUCCESS** — VC has been successfully established on this port. The client may begin to submit demand on this VC.
- **IX_ATMSCH_RET_NOT_ADMITTED** — VC cannot be established on this port because there is insufficient upstream capacity available to support the requested traffic contract descriptor.
- **IX_FAIL** — Input data are invalid. VC has not been established.

### A.3.7.4.10 ixAtmSchVcQueueClear (..)

ixAtmSchVcQueueClear (IxAtmLogicalPort port, IxAtmSchedulerVcId vcId)

The client calls this function to remove all currently queued cells from a registered VC. The pending cell count for the specified VC is reset to zero.

This interface is structurally compatible with the IxAtmdAccSchQueueClear callback type definition required for the Intel® IXP42X product line ATM scheduler interoperability.

#### Parameters

***IxAtmLogicalPort port*** — Specifies the ATM port on which the VC to be cleared is established.

***IxAtmSchedulerVcId vcId*** — Identifies the VC to be cleared. This is the value returned by the ixAtmSchVcModelSetup call which established the relevant VC.

#### Returns

- **IX_SUCCESS** — VC queue has been successfully cleared.
- **IX_FAIL** — Input are invalid. No VC queue is modified.

***Warning:*** IxAtmSch assumes that the calling software ensures that calls to ixAtmSchVcQueueUpdate, ixAtmSchVcQueueClear and ixAtmSchTableUpdate are both self and mutually exclusive for the same port.

#### See also:

ixAtmSchVcQueueUpdate(), ixAtmSchVcQueueClear(), ixAtmSchTableUpdate().

### A.3.7.4.11 ixAtmSchVcQueueUpdate (..)

ixAtmSchVcQueueUpdate (IxAtmLogicalPort port, IxAtmSchedulerVcId vcId, unsigned int numberOfCells)

The client calls this function to notify IxAtmSch that the user of a VC has submitted cells for transmission.

This information is stored, aggregated from a number of calls to ixAtmSchVcQueueUpdate and eventually used in the call to ixAtmSchTableUpdate.

Normally IxAtmSch will update the VC queue by adding the number of cells to the current queue length. However, if IxAtmSch determines that the user has over-submitted for the VC and exceeded its transmission quota the queue request can be rejected. The user should resubmit the request later when the queue has been depleted.

This implementation of ixAtmSchVcQueueUpdate uses no operating system or external facilities, either directly or indirectly. This allows clients to call this function form within an interrupt handler.

This interface is structurally compatible with the IxAtmdAccSchQueueUpdate callback type definition required for the Intel® IXP42X product line ATM scheduler interoperability.

#### Parameters

***IxAtmLogicalPort port*** — Specifies the ATM port on which the VC to be updated is established.

***IxAtmSchedulerVcId vcId*** — Identifies the VC to be updated. This is the value returned by the ixAtmSchVcModelSetup call which established the relevant VC.

***unsigned int numberOfCells*** — Indicates how many ATM cells should be added to the queue for this VC.

#### Returns

- **IX_SUCCESS** — VC queue has been successfully updated.
- **IX_ATMSCH_RET_QUEUE_FULL** — VC queue has reached a preset limit. This indicates the client has over-submitted and exceeded its transmission quota. The request is rejected. The VC queue is not updated. The VC user is advised to resubmit the request later.
- **IX_FAIL** — Input are invalid. No VC queue is updated.

***Warning:*** IxAtmSch assumes that the calling software ensures that calls to ixAtmSchVcQueueUpdate, ixAtmSchVcQueueClear and ixAtmSchTableUpdate are both self and mutually exclusive for the same port.

#### See also:

ixAtmSchVcQueueUpdate(), ixAtmSchVcQueueClear(), ixAtmSchTableUpdate().

## A.3.8 ATM Types (IxAtmTypes)

The common set of types used in many Atm components.

### A.3.8.1 Data Structures

- struct IxAtmScheduleTable

  This structure defines a schedule table which gives details on which data (from which VCs) should be transmitted for a forthcoming period of time for a particular port and the order in which that data should be transmitted.
- struct IxAtmScheduleTableEntry

  ATM Schedule Table entry.
- struct IxAtmTrafficDescriptor

  Structure describing an ATM traffic contract for a Virtual Connection (VC).

### A.3.8.2 Defines

- #define IX_ATM_CELL_PAYLOAD_SIZE 48

  Size of a ATM cell payload.
- #define IX_ATM_CELL_SIZE 53

  Size of a ATM cell, including header.
- #define IX_ATM_CELL_SIZE_NO_HEC (IX_ATM_CELL_SIZE - 1)

  Size of a ATM cell, excluding HEC byte.

- #define IX_ATM_OAM_CELL_SIZE_NO_HEC IX_ATM_CELL_SIZE_NO_HEC
- #define IX_ATM_AAL0_48_CELL_PAYLOAD_SIZE IX_ATM_CELL_PAYLOAD_SIZE
- #define IX_ATM_AAL0_52_CELL_SIZE_NO_HEC IX_ATM_CELL_SIZE_NO_HEC
- #define IX_ATM_MAX_VPI 255
  Maximum value of an ATM VPI.
- #define IX_ATM_MAX_VCI 65535
  Maximum value of an ATM VCI.
- #define IX_ATM_MAX_NUM_AAL_VCS 32
- #define IX_ATM_MAX_NUM_VC IX_ATM_MAX_NUM_AAL_VCS
  Maximum number of active AAL5/AAL0 VCs in the system The use of this macro is depreciated, it is retained for backward compatibility with CSR 1.0 clients. For CSR 1.x and beyond the define IX_ATM_MAX_NUM_AAL_VC should be used.
- #define IX_ATM_MAX_NUM_OAM_TX_VCS IX_UTOPIA_MAX_PORTS
  Maximum number of active OAM Tx VCs in the system, 1 OAM VC per port.
- #define IX_ATM_MAX_NUM_OAM_RX_VCS 1
  Maximum number of active OAM Rx VCs in the system, 1 OAM VC shared across all ports.
- #define IX_ATM_MAX_NUM_AAL_OAM_TX_VCS (IX_ATM_MAX_NUM_AAL_VCS + IX_ATM_MAX_NUM_OAM_TX_VCS)
  Maximum number of active AAL5/AAL0/OAM Tx VCs in the system.
- #define IX_ATM_MAX_NUM_AAL_OAM_RX_VCS (IX_ATM_MAX_NUM_AAL_VCS + IX_ATM_MAX_NUM_OAM_RX_VCS)
  Maximum number of active AAL5/AAL0/OAM Rx VCs in the system.
- #define IX_ATM_IDLE_CELLS_CONNID 0
  VC Id used to indicate idle cells in the returned schedule table.
- #define IX_ATM_CEL_HEADER_VCI_GET
- #define IX_ATM_CEL_HEADER_VPI_GET
- #define IX_ATM_CEL_HEADER_PTI_GET

### A.3.8.3 Typedefs

- typedef unsigned int IxAtmCellHeader
- typedef unsigned int IxAtmConnId
  ATM VC data connection identifier.
- typedef int IxAtmSchedulerVcId
  ATM VC scheduling connection identifier.
- typedef unsigned int IxAtmNpeRxVcId
  ATM Rx VC identifier used by the ATM Npe.

### A.3.8.4 Enumerations

- enum IxAtmLogicalPort { IX_UTOPIA_PORT_0 = 0, IX_UTOPIA_PORT_1, IX_UTOPIA_PORT_2, IX_UTOPIA_PORT_3, IX_UTOPIA_PORT_4, IX_UTOPIA_PORT_5, IX_UTOPIA_PORT_6, IX_UTOPIA_PORT_7, IX_UTOPIA_PORT_8, IX_UTOPIA_PORT_9, IX_UTOPIA_PORT_10, IX_UTOPIA_PORT_11, IX_UTOPIA_MAX_PORTS }

  Logical Port Definitions.

- enum IxAtmServiceCategory { IX_ATM_CBR, IX_ATM_RTVBR, IX_ATM_VBR, IX_ATM_UBR, IX_ATM_ABR }

  Enumerated type representing available ATM service categories. For more information on these categories, see “Traffic Management Specification” v4.1, published by the ATM Forum - http://www.atmforum.com.

- enum IxAtmRxQueueId { IX_ATM_RX_A = 0, IX_ATM_RX_B, IX_ATM_MAX_RX_STREAMS }

  Rx Queue Type for RX traffic.

### A.3.8.5 Detailed Description

The common set of types used in many ATM components.

### A.3.8.6 Define Documentation

- #define IX_ATM_CEL_HEADER_PTI_GET

Value:

```
(IxAtmCellHeader cellHeader) \
    ((cellHeader) >> 1) & IX_OAM_PTI_BITS_MASK;
```

- #define IX_ATM_CEL_HEADER_VCI_GET

Value:

```
(IxAtmCellHeader cellHeader) \
    (((cellHeader) >> 4) & IX_OAM_VCI_BITS_MASK); \
```

- #define IX_ATM_CEL_HEADER_VPI_GET

Value:

```
(IxAtmCellHeader cellHeader) \
    (((cellHeader) >> 20) & IX_OAM_VPI_BITS_MASK);
```

### A.3.8.7 Typedef Documentation

#### A.3.8.7.1 IxAtmConnId

ATM VC data connection identifier.

This is generated by IxAtmdAcc when a successful connection is made on a VC. The is the ID by which IxAtmdAcc knows an active VC and should be used in IxAtmdAcc API calls to reference a specific VC.

#### A.3.8.7.2 IxAtmNpeRxVcId

ATM Rx VC identifier used by the ATM Npe.

This Id is generated by IxAtmdAcc when a successful data connection is made on a rx VC. This Id is typically used in conjunction with the IxFpathAcc component.

#### A.3.8.7.3 IxAtmSchedulerVcId

ATM VC scheduling connection identifier.

This id is generated and used by ATM Tx controller, generally the traffic shaper (e.g. IxAtmSch). The IxAtmdAcc component will request one of these Ids whenever a data connection on a Tx VC is requested. This ID will be used in callbacks to the ATM Transmission Ctrl s/w (e.g. IxAtmm) to reference a particular VC.

### A.3.8.8 Enumeration Type Documentation

#### A.3.8.8.1 enum IxAtmLogicalPort

Logical Port Definitions

Only 1 port is available in SPHY configuration 12 ports are enabled in MPHY configuration

**Enumeration values**

- IX_UTOPIA_PORT_0 Port 0.
- IX_UTOPIA_PORT_1 Port 1.
- IX_UTOPIA_PORT_2 Port 2.
- IX_UTOPIA_PORT_3 Port 3.
- IX_UTOPIA_PORT_4 Port 4.
- IX_UTOPIA_PORT_5 Port 5.
- IX_UTOPIA_PORT_6 Port 6.
- IX_UTOPIA_PORT_7 Port 7.
- IX_UTOPIA_PORT_8 Port 8.
- IX_UTOPIA_PORT_9 Port 9.
- IX_UTOPIA_PORT_10 Port 10.
- IX_UTOPIA_PORT_11 Port 11.
- IX_UTOPIA_MAX_PORTS Not a port - just a definition for the maximum possible ports.

#### A.3.8.8.2 enum IxAtmRxQueueId

Rx Queue Type for RX traffic.

IxAtmRxQueueId defines the queues involved for receiving data.

There are two queues to facilitate prioritization handling and processing the 2 queues with different algorithms and constraints.

For example, one queue can carry voice (or time-critical traffic), the other queue can carry non-voice traffic.

**Enumeration Values**

- IX_ATM_RX_A  RX queue A.
- IX_ATM_RX_B  RX queue B.
- IX_ATM_MAX_RX_STREAMS  Maximum number of RX streams.

#### A.3.8.8.3 enum IxAtmServiceCategory

Enumerated type representing available ATM service categories. For more information on these categories, see “Traffic Management Specification” v4.1, published by the ATM Forum - http://http://www.atmforum.com.

**Enumeration Values**

- IX_ATM_CBR  Constant Bit Rate.
- IX_ATM_RTVBR  Real Time Variable Bit Rate.
- IX_ATM_VBR  Variable Bit Rate.
- IX_ATM_UBR  Unspecified Bit Rate.
- IX_ATM_ABR  Available Bit Rate (not supported).

## A.3.9 Security (IxCryptoAcc) API

Intel® IXP42X product line Security component Public API.

### A.3.9.1 Data Structures

- struct IxCryptoAccAuthCtx
  Structure storing authentication configuration parameters required to perform security functionality.
- struct IxCryptoAccCipherCtx
  Structure storing cipher configuration parameters required to perform security functionality.
- struct IxCryptoAccCtx
  Structure storing configuration parameters required to perform security functionality.

### A.3.9.2 Defines

- #define IX_CRYPTO_ACC_MAX_CIPHER_KEY_LENGTH  32
  Max length (byte) of cipher key for DES (64 bit),  3DES (192 bit), AES (128, 192 & 256 bit).
- #define IX_CRYPTO_ACC_MAX_CIPHER_IV_LENGTH  16
  Max IV length in byte.

- #define IX_CRYPTO_ACC_MAX_AUTH_KEY_LENGTH 20
  Max length (byte) of authentication key for SHA1 (160 bit) and MD5 (128 bit).
- #define IX_CRYPTO_ACC_MAX_AUTH_IV_LENGTH 20
  Max length (byte) of initial chaining variable for SHA1 (160 bit) and MD5 (128 bit).
- #define IX_CRYPTO_ACC_MAX_QUEUE_DEPTH 64
  Max queue depth supported by the Queue Manager.
- #define IX_CRYPTO_ACC_MAX_ACTIVE_SA_TUNNELS 1000
  Maximum active tunnels supported could be changed by the client based on the application's requirements.
- #define IX_CRYPTO_ACC_DES_KEY_64 8
- #define IX_CRYPTO_ACC_DES_BLOCK_64 8
- #define IX_CRYPTO_ACC_DES_IV_64 8
- #define IX_CRYPTO_ACC_3DES_KEY_192 24
- #define IX_CRYPTO_ACC_AES_KEY_128 16
- #define IX_CRYPTO_ACC_AES_KEY_192 24
- #define IX_CRYPTO_ACC_AES_KEY_256 32
- #define IX_CRYPTO_ACC_AES_BLOCK_128 16
- #define IX_CRYPTO_ACC_AES_CBC_IV_128 16
- #define IX_CRYPTO_ACC_AES_CTR_IV_64 8
- #define IX_CRYPTO_ACC_SHA1_KEY_160 20
- #define IX_CRYPTO_ACC_SHA1_DIGEST_160 20
- #define IX_CRYPTO_ACC_MD5_KEY_128 16
- #define IX_CRYPTO_ACC_MD5_DIGEST_128 16

### A.3.9.3 Typedefs

- typedef void(* IxCryptoAccRegisterCompleteCallback )(UINT32 cryptoCtxId, IX_MBUF *pMbuf, IxCryptoAccStatus status)
  Cryptographic Context registration completed callback notification.
- typedef void(* IxCryptoAccPerformCompleteCallback )(UINT32 cryptoCtxId, IX_MBUF *pSrcMbuf, IX_MBUF *pDestMbuf, IxCryptoAccStatus status)
  Security process completed callback notification.

### A.3.9.4 Enumerations

- enum IxCryptoAccOperation { IX_CRYPTO_ACC_OP_ENCRYPT = 0, IX_CRYPTO_ACC_OP_DECRYPT, IX_CRYPTO_ACC_OP_AUTH_CALC, IX_CRYPTO_ACC_OP_AUTH_CHECK, IX_CRYPTO_ACC_OP_ENCRYPT_AUTH, IX_CRYPTO_ACC_OP_AUTH_DECRYPT, IX_CRYPTO_ACC_OP_TYPE_OF_OPERATION }
  Cryptographic Operation Definitions.

- enum IxCryptoAccCipherAlgo { IX_CRYPTO_ACC_CIPHER_NULL = 0, IX_CRYPTO_ACC_CIPHER_DES, IX_CRYPTO_ACC_CIPHER_3DES, IX_CRYPTO_ACC_CIPHER_AES, IX_CRYPTO_ACC_CIPHER_ALGO_TYPE }

  Cipher Algorithm Definitions.

- enum IxCryptoAccCipherMode { IX_CRYPTO_ACC_MODE_NULL = 0, IX_CRYPTO_ACC_MODE_ECB, IX_CRYPTO_ACC_MODE_CBC, IX_CRYPTO_ACC_MODE_CTR, IX_CRYPTO_ACC_MODE_TYPE }

  Cipher Mode Definitions.

- enum IxCryptoAccAuthAlgo { IX_CRYPTO_ACC_AUTH_NULL = 0, IX_CRYPTO_ACC_AUTH_SHA1, IX_CRYPTO_ACC_AUTH_MD5, IX_CRYPTO_ACC_AUTH_TYPE }

  Authentication Algorithm Definitions.

- enum IxCryptoAccStatus { IX_CRYPTO_ACC_STATUS_SUCCESS = IX_SUCCESS, IX_CRYPTO_ACC_STATUS_FAIL = IX_FAIL, IX_CRYPTO_ACC_STATUS_WAIT, IX_CRYPTO_ACC_STATUS_RETRY, IX_CRYPTO_ACC_STATUS_QUEUE_FULL, IX_CRYPTO_ACC_STATUS_OPERATION_NOT_SUPPORTED, IX_CRYPTO_ACC_STATUS_CIPHER_ALGO_NOT_SUPPORTED, IX_CRYPTO_ACC_STATUS_CIPHER_MODE_NOT_SUPPORTED, IX_CRYPTO_ACC_STATUS_CIPHER_INVALID_KEY_LEN, IX_CRYPTO_ACC_STATUS_CIPHER_INVALID_IV_LEN, IX_CRYPTO_ACC_STATUS_CIPHER_INVALID_BLOCK_LEN, IX_CRYPTO_ACC_STATUS_AUTH_ALGO_NOT_SUPPORTED, IX_CRYPTO_ACC_STATUS_AUTH_INVALID_DIGEST_LEN, IX_CRYPTO_ACC_STATUS_AUTH_INVALID_KEY_LEN, IX_CRYPTO_ACC_STATUS_AUTH_FAIL, IX_CRYPTO_ACC_STATUS_CRYPTO_CTX_NOT_VALID, IX_CRYPTO_ACC_STATUS_EXCEED_MAX_TUNNELS }

  Invalid Status Definitions.

### A.3.9.5 Functions

- PUBLIC IxCryptoAccStatus ixCryptoAccInit (void)

  Initialize the Security Access component.

- PUBLIC IxCryptoAccStatus ixCryptoAccCtxRegister (IxCryptoAccCtx *pAccCtx, IX_MBUF *pMbufPrimaryChainVar, IX_MBUF *pMbufSecondaryChainVar, IxCryptoAccRegisterCompleteCallback registerCallbackFn, IxCryptoAccPerformCompleteCallback performCallbackFn, UINT32 *pCryptoCtxId)

  Cryptographic Context ID, CryptoCtxId Allocation for the Client.

- PUBLIC IxCryptoAccStatus ixCryptoAccCtxUnregister (UINT32 cryptoCtxId)

  Unregister the CryptoCtxId from Cryptographic Context Database (CCD).

- PUBLIC IxCryptoAccStatus ixCryptoAccAuthCryptPerform (UINT32 cryptoCtxId, IX_MBUF *pSrcMbuf, IX_MBUF *pDestMbuf, UINT16 authStartOffset, UINT16 authDataLen, UINT16 cryptStartOffset, UINT16 cryptDataLen, UINT16 icvOffset, UINT8 *pIV)

  Perform Authentication and Decryption/Encryption functionality.

- PUBLIC void ixCryptoAccShow (void)

  API for printing statistic and status.

- PUBLIC void ixCryptoAccShowWithId (UINT32 cryptoCtxId)
  API for printing statistic and status.
- PUBLIC IxCryptoAccStatus ixCryptoAccCryptoServiceStop (void)
  API to stop the security hardware accelerator services.

### A.3.9.6 Detailed Description

Intel® IXP42X product line Security component Public API.

### A.3.9.7 Define Documentation

#### A.3.9.7.1 #define IX_CRYPTO_ACC_MAX_ACTIVE_SA_TUNNELS 1000

Maximum active tunnels supported could be changed by the client based on the application's requirements.

Number of active tunnels will not impact the performance but will have an impact on the memory needed to keep the crypto context information will depend on it. Overall memory requirement depends on the number of tunnels.

### A.3.9.8 Typedef Documentation

#### A.3.9.8.1 typedef void(..)

typedef void(* IxCryptoAccPerformCompleteCallback)( UINT32 cryptoCtxId, IX_MBUF *pSrcMbuf, IX_MBUF *pDestMbuf, IxCryptoAccStatus status)

Security process completed callback notification.

This function is called to notify a client that the cryptographic transaction has been completed. The CryptoCtxId is returned to the client to indicate which process that has been completed. The CryptoCtxId is created via ixCryptoAccCtxRegister, and this ID is unique to a particular IPSec tunnel.

**Parameters**

***cryptoCtxId*** — [in] The CryptoCtxId is provided when the callback was registered via ixCryptoAccCtxRegister for a particular Security Association (SA). Then the CryptoCtxId will be passed ixCryptoAccAuthCryptPerform. CryptoCtxId points to a struct into (CCD) consists of cryptographic parameters database required by the Network Processor Engine (NPE). It is returned to the client to indicate that the process for the particular CryptoCtxId is completed.

***pSrcMbuf*** — [in] Pointer to the source data mbuf descriptor, which is also processed data mbuf descriptor if UseDifferentSrcAndDestMbufs is FALSE.

***pDestMbuf*** — [in] Only used if UseDifferentSrcAndDestMbufs is TRUE. Pointer to the processed data mbuf descriptor.

***status*** — [in] Status reporting to the client via IxCryptoAccStatus.

### Returns

- IX_CRYPTO_ACC_STATUS_SUCCESS — Operation completed successfully.
- IX_CRYPTO_ACC_STATUS_FAIL — Operation fail.
- IX_CRYPTO_ACC_STATUS_AUTH_FAIL — Authentication is unsuccessful. Note that when the authentication fails, the content of the destination mbuf (which is the same as the source mbuf if a normal in-line operation is performed) will be indeterminate.

## A.3.9.8.2 typedef void(..)

typedef void(* IxCryptoAccRegisterCompleteCallback)( UINT32 cryptoCtxId, IX_MBUF *pMbuf, IxCryptoAccStatus status)

Cryptographic Context registration completed callback notification.

This function is called to notify a client that the Crypto Context has been registered. This function will return the status through the associated CryptoCtxId once the parameters are calculated by Network Processor Engine is (NPE) and stored in the struct, Cryptographic Context Database (CCD) which reference by the CryptoCtxId. The CryptoCtxId is valid until ixCryptoAccCtxUnregister is invoked.

If the callback function return the IX_CRYPTO_ACC_STATUS_WAIT status, it indicates that registration is not completed yet, but the mbuf pointer will need to be freed by client and the client need to wait for the next completion indication.Registration completed successfully only if status IX_CRYPTO_ACC_STATUS_SUCCESS is received.

### Parameters

***cryptoCtxId*** — [in] This CryptoCtxId is provided when the callback was registered for a particular Security Association (SA) via ixCryptoAccAuthCryptPerform. CryptoCtxId points to a struct (CCD) consists of cryptographic parameters database required by the Network Processor Engine (NPE).

***pMbuf*** — [in] Pointer to the unused mbuf (to be freed by client). The client should free any mbuf that is not NULL.

***status*** — [in] Status reporting to the client via IxCryptoAccStatus.

### Returns

- IX_CRYPTO_ACC_STATUS_SUCCESS — registration successful.
- IX_CRYPTO_ACC_STATUS_FAIL — registration fail.
- IX_CRYPTO_ACC_STATUS_WAIT — registration not completed yet, wait for the next completion indication. NPE is busy in calculating the initial variables needed for the registration.

## A.3.9.9 Enumeration Type Documentation

### A.3.9.9.1 enum IxCryptoAccAuthAlgo

Authentication Algorithm Definitions.

**Note**

Only two authentication algorithms are supported, SHA1 and MD5.

### A.3.9.9.2 enum IxCryptoAccCipherAlgo

Cipher Algorithm Definitions.

**Note**

3DES and AES will be supported if not violating import/export rules.

### A.3.9.9.3 enum IxCryptoAccCipherMode

Cipher Mode Definitions.

**Note**

CFB and OFB are not supported.

### A.3.9.9.4 enum IxCryptoAccOperation

Cryptographic Operation Definitions.

**Note**

Only one of these operation is supported.

### A.3.9.9.5 enum IxCryptoAccStatus

Invalid Status Definitions.

**Note**

These status will be used by the APIs to return to the client.

## A.3.9.10 Function Documentation

### A.3.9.10.1 ixCryptoAccAuthCryptPerform (..)

ixCryptoAccAuthCryptPerform (UINT32 cryptoCtxId, IX_MBUF * pSrcMbuf, IX_MBUF * pDestMbuf, UINT16 authStartOffset, UINT16 authDataLen, UINT16 cryptStartOffset, UINT16 cryptDataLen, UINT16 icvOffset, UINT8 * pIV)

Perform Authentication and Decryption/Encryption functionality.

This function is called for authentication and decryption/encryption functionality. Note that the restriction with respect to AuthStartOffset, AuthStartLen, CryptStartOffset and CryptStartLen is that the crypted data must be a subset of the authenticated data. The boundary relationship of

[(authStartOffset+authDataLen) >= (cryptDataLen+cryptStartOffset) >= (cryptStartOffset >= authStartOffset] MUST BE SATISFIED. There should not be any chained mbuf boundary with an ICV field.

### Parameters

***cryptoCtxId*** — [in] is the CCD pointer to be supplied by the client. This CryptoCtxId is allocated via ixCryptoAccCtxRegister. The CryptoCtxId must be a valid Id.

***pSrcMbuf*** — [in] is a pointer to the data to be processed. This mbuf structure is allocated by the client. The processed data will be stored in the location if UseDifferentSrcAndDestMbufs is FALSE. The same pointer is then returned to the client via callback function.

***pDestMbuf*** — [in] only used if UseDifferentSrcAndDestMbufs is TRUE. This is a pointer to the buffer that the result is to be written to. This mbuf structure is allocated by the client. The processed data will be stored in the location. The same pointer is then returned to the client via callback function IxCryptoAccPerformCompleteCallback registered during ixCryptoAccCtxRegister. The length of mbuf MUST be big enough to hold the result of operation. The result of operation COULD NOT span into two or more different mbufs, thus the mbuf supplied must be at least the length of expected result.

***authStartOffset*** — [in] supplied by the client to indicate the start of the payload to be authenticated.

***authDataLen*** — [in] supplied by the client to indicate the length of the payload to be authenticated in Bytes. The maximum data length must not exceed 65471

***icvOffset*** — [in] supplied by the client to indicate the start of the ICV (Initial Chaining Variable) used for the authentication. This ICV field should not cross to multiple mbufs, in a chained mbuf.

***cryptStartOffset*** — [in] supplied by the client to indicate the start of the payload to be decryption/encryption.

***cryptDataLen*** — [in] supplied by the client to indicate the length of the payload to be decryption/encryption in Bytes. The must be in the multiple of cipher block length.

***pIV*** — [in] Initialization Vector supplied by the client to be used for the decryption/encryption process.

### Returns

- IX_CRYPTO_ACC_STATUS_SUCCESS — Operation requested is successfully enqueued to hardware accelerator for processing.
- IX_CRYPTO_ACC_STATUS_FAIL — Cryptographic process failed for some unspecified internal reasons.
- IX_CRYPTO_ACC_STATUS_QUEUE_FULL — Cryptographic queue is full.
- IX_CRYPTO_ACC_STATUS_CIPHER_INVALID_BLOCK_LEN — Invalid plaintext / ciphertext block size passed in by the client.
- IX_CRYPTO_ACC_STATUS_CRYPTO_CTX_NOT_VALID — Crypto context is not registered.
  - Reentrant — no
  - ISR Callable — no

### A.3.9.10.2 ixCryptoAccCryptoServiceStop (void)

API to stop the security hardware accelerator services.

This function is called to stop THE security hardware accelerator services. All the requests pending in queue will be completed before the security hardware accelerator halt. Any new requests issued after this call will be rejected. All the Crypto Context will be unregistered in this function call.

#### Returns

- IX_CRYPTO_ACC_STATUS_SUCCESS — Operation requested is successful with all the pending requests completed and CCD cleared.
- IX_CRYPTO_ACC_STATUS_FAIL — Cryptographic stop request failed for some unspecified internal reasons.
  - Reentrant — no
  - ISR Callable — no

### A.3.9.10.3 ixCryptoAccCtxRegister (..)

ixCryptoAccCtxRegister (IxCryptoAccCtx * pAccCtx, IX_MBUF * pMbufPrimaryChainVar, IX_MBUF * pMbufSecondaryChainVar, IxCryptoAccRegisterCompleteCallback registerCallbackFn, IxCryptoAccPerformCompleteCallback performCallbackFn, UINT32 * pCryptoCtxId)

Cryptographic Context ID, CryptoCtxId Allocation for the Client.

This function is for the client to pass all the required information (for example, key, cipher algorithm, etc) for the hardware accelerator context. All the information will be converted into a cryptographic parameters structure to be shared with NPE. The structure will be associated with CryptoCtxId. (CryptoCtxId need to be passed back to client through callback). Besides, two empty mbuf are required to be passed in through interface for the use of access component and NPE to compute the primary and secondary initial chaining variables. pMbufPrimaryChainVar and pMbufSecondaryChainVar mbuf pointers must be NULL if authentication or combined service is not selected.

#### Parameters

***pAccCtx*** — [in] is a pointer to the Security Association Database (SAD) which will assigned a CryptoCtxId. The information inside SAD will be used for the creation of Cryptographic Context Database (CCD).

***pMbufPrimaryChainVar*** — [in] a pointer to an empty mbuf (must not be chained) for the use of access component to compute primary chaining variables for SHA1/MD5. This mbuf structure is allocated by the client (minimum size of the cluster required is 64 bytes), only if SHA1/MD5 is selected, and the mbuf pointer must be NULL if SHA1/MD5 is not selected. After the NPE complete the computation, the mbuf is returned separately through the client's registered callback.

***pMbufSecondaryChainVar*** — [in] a pointer to an empty mbuf (must not be chained) for the use of access component to compute secondary chaining variables for SHA1/MD5. This mbuf structure is allocated by the client (minimum size of the cluster required is 64 bytes), only if SHA1/MD5 is selected, and the mbuf pointer must be NULL if SHA1/MD5 is not selected. After the NPE complete the computation, the mbuf is returned separately through the client's registered callback.

***registerCallbackFn*** — [in] callback function pointer to return CryptoCtxId the client when the registration is completed. This cannot be NULL.

***performCallbackFn*** — [in] callback function pointer to return the processed buffer to the client with respect to the unique CryptoCtxId. This cannot be NULL.

***pCryptoCtxId*** — [inout] Crypto Context ID returned by access component for the crypto context registered.

**Returns**

- IX_CRYPTO_ACC_STATUS_SUCCESS — Registration parameters valid
- IX_CRYPTO_ACC_STATUS_FAIL — Registration failed for some unspecified internal reasons.
- IX_CRYPTO_ACC_STATUS_OPERATION_NOT_SUPPORTED — Invalid operation requested by the client.
- IX_CRYPTO_ACC_STATUS_CIPHER_ALGO_NOT_SUPPORTED — Invalid cipher algorithm requested by the client.
- IX_CRYPTO_ACC_STATUS_CIPHER_MODE_NOT_SUPPORTED — Invalid cipher mode requested by the client.
- IX_CRYPTO_ACC_STATUS_CIPHER_INVALID_KEY_LEN — Invalid cipher key length passed in by the client.
- IX_CRYPTO_ACC_STATUS_CIPHER_INVALID_BLOCK_LEN — Invalid cipher block length passed in by the client
- IX_CRYPTO_ACC_STATUS_CIPHER_INVALID_IV_LEN — Invalid IV length passed in by the client.
- IX_CRYPTO_ACC_STATUS_AUTH_ALGO_NOT_SUPPORTED — Invalid authentication requested by the client.
- IX_CRYPTO_ACC_STATUS_AUTH_INVALID_DIGEST_LEN — Invalid authentication digest length.
- IX_CRYPTO_ACC_STATUS_AUTH_INVALID_KEY_LEN — Invalid authentication key length.
- IX_CRYPTO_ACC_STATUS_EXCEED_MAX_TUNNELS — Exceed maximum tunnels permitted.
- IX_CRYPTO_ACC_STATUS_QUEUE_FULL — Queue full status returned and the registration request will be rejected.

  — Reentrant — no

  — ISR Callable — no

### A.3.9.10.4 ixCryptoAccCtxUnregister (..)

ixCryptoAccCtxUnregister (UINT32 cryptoCtxId)

Unregister the CryptoCtxId from Cryptographic Context Database (CCD).

This function is for freeing the particular CryptoCtxId from the Cryptographic Context Database.

#### Parameters

***cryptoCtxId*** — [in] is the CCD pointer that is required to be freed, which will be supplied by the client. The CryptoCtxId must be a valid Id.

#### Returns

- IX_CRYPTO_ACC_STATUS_SUCCESS — successfully unregister the CryptoCtxId.
- IX_CRYPTO_ACC_STATUS_FAIL — Unregistration failed for some unspecified internal reasons, e.g. uninitialized.
- IX_CRYPTO_ACC_STATUS_CRYPTO_CTX_NOT_VALID — invalid CryptoCtxId.
- IX_CRYPTO_ACC_STATUS_RETRY — retry the unregister operation as there are still some ixCryptoAccAuthCryptPerform requests associated with the CryptoCtxId pending on the NPE queue.
  - Reentrant — no
  - ISR Callable — no

### A.3.9.10.5 ixCryptoAccInit (void)

Initialize the Security Access component.

This API will initialize the Security Access component internals, such as Queue Manager, allocate memory and initialize all the variables. This API must be called before any other API call on this component can be made. This API is called once only.

#### Returns

- IX_CRYPTO_ACC_STATUS_SUCCESS — Successfully initialize the component.
- IX_CRYPTO_ACC_STATUS_FAIL — Initialization failed for some unspecified internal reason.
  - Reentrant — no
  - ISR Callable — no

### A.3.9.10.6 ixCryptoAccShow (void)

API for printing statistic and status.

This function is called by the client to print statistics, such as number of packets returned with operation fail, number of packets encrypted/ decrypted/authenticated while the function also print the current status of the queue, whether the queue is empty or full or current queue length.

- Reentrant — no
- ISR Callable — no

### A.3.9.10.7 ixCryptoAccShowWithId (UINT32 *cryptoCtxId*)

API for printing statistic and status.

This function prints all the statistics provided by ixCryptoAccShow. In addition the function will print out the contents of the Crypto Context corresponding to the CryptoCtxId supplied through the API.

#### Parameters

***cryptoCtxId*** — [in] Crypto Context ID which has been registered

- Reentrant — no
- ISR Callable — no

## A.3.10 DMA Types (IxDmaTypes)

The common set of types used in the DMA component.

### A.3.10.1 Enumerations

- enum IxDmaReturnStatus { IX_DMA_SUCCESS = IX_SUCCESS, IX_DMA_FAIL = IX_FAIL, IX_DMA_INVALID_TRANSFER_WIDTH, IX_DMA_INVALID_TRANSFER_LENGTH, IX_DMA_INVALID_TRANSFER_MODE, IX_DMA_INVALID_ADDRESS_MODE, IX_DMA_REQUEST_FIFO_FULL }

  Dma return status definitions.
- enum IxDmaTransferMode { IX_DMA_COPY_CLEAR = 0, IX_DMA_COPY, IX_DMA_COPY_BYTE_SWAP, IX_DMA_COPY_REVERSE, IX_DMA_TRANSFER_MODE_INVALID }

  Dma transfer mode definitions.
- enum IxDmaAddressingMode { IX_DMA_INC_SRC_INC_DST = 0, IX_DMA_INC_SRC_FIX_DST, IX_DMA_FIX_SRC_INC_DST, IX_DMA_FIX_SRC_FIX_DST, IX_DMA_ADDRESSING_MODE_INVALID }

  Dma addressing mode definitions.
- enum IxDmaTransferWidth { IX_DMA_32_SRC_32_DST = 0, IX_DMA_32_SRC_16_DST, IX_DMA_32_SRC_8_DST, IX_DMA_16_SRC_32_DST, IX_DMA_16_SRC_16_DST, IX_DMA_16_SRC_8_DST, IX_DMA_8_SRC_32_DST, IX_DMA_8_SRC_16_DST, IX_DMA_8_SRC_8_DST, IX_DMA_8_SRC_BURST_DST, IX_DMA_16_SRC_BURST_DST, IX_DMA_32_SRC_BURST_DST, IX_DMA_BURST_SRC_8_DST, IX_DMA_BURST_SRC_16_DST, IX_DMA_BURST_SRC_32_DST, IX_DMA_BURST_SRC_BURST_DST, IX_DMA_TRANSFER_WIDTH_INVALID }

  Dma transfer width definitions @Note Fixed addresses (either source or destination) do not support burst transfer width.
- enum IxDmaNpeId { IX_DMA_NPEID_NPEA = 0, IX_DMA_NPEID_NPEB, IX_DMA_NPEID_NPEC, IX_DMA_NPEID_MAX }

  NpeId numbers to identify NPE A, B or C.

### A.3.10.2 Detailed Description

The common set of types used in the DMA component.

## A.3.10.3 Enumeration Type Documentation

### A.3.10.3.1 enum IxDmaAddressingMode

Dma addressing mode definitions.

**Note**

Fixed source address to fixed destination address addressing mode is not supported.

**Enumeration Values**

- IX_DMA_INC_SRC_INC_DST Incremental source address to incremental destination address.
- IX_DMA_INC_SRC_FIX_DST Incremental source address to incremental destination address.
- IX_DMA_FIX_SRC_INC_DST Incremental source address to incremental destination address.
- IX_DMA_FIX_SRC_FIX_DST Incremental source address to incremental destination address.

### A.3.10.3.2 enum IxDmaNpeId

NpeId numbers to identify NPE A, B or C.

**Enumeration Values**

- IX_DMA_NPEID_NPEA Identifies NPE A.
- IX_DMA_NPEID_NPEB Identifies NPE B.
- IX_DMA_NPEID_NPEC Identifies NPE C.
- IX_DMA_NPEID_MAX Total Number of NPEs.

### A.3.10.3.3 enum IxDmaReturnStatus

DMA return status definitions.

**Enumeration values**

- IX_DMA_SUCCESS DMA Transfer Success.
- IX_DMA_FAIL DMA Transfer Fail.
- IX_DMA_INVALID_TRANSFER_WIDTH Invalid transfer width.
- IX_DMA_INVALID_TRANSFER_LENGTH Invalid transfer length.
- IX_DMA_INVALID_TRANSFER_MODE Invalid transfer mode.
- IX_DMA_INVALID_ADDRESS_MODE Invalid address mode.
- IX_DMA_REQUEST_FIFO_FULL DMA request queue is full.

### A.3.10.3.4 enum IxDmaTransferMode

Dma transfer mode definitions.

#### Note

Copy and byte swap, and copy and reverse modes only support multiples of word data length.

#### Enumeration values

- IX_DMA_COPY_CLEAR Copy and clear source.
- IX_DMA_COPY Copy.
- IX_DMA_COPY_BYTE_SWAP Copy and byte swap (endian).
- IX_DMA_COPY_REVERSE Copy and reverse.

### A.3.10.3.5 enum IxDmaTransferWidth

Dma transfer width definitions @Note Fixed addresses (either source or destination) do not support burst transfer width.

#### Enumeration values

- IX_DMA_32_SRC_32_DST 32-bit src to 32-bit dst.
- IX_DMA_32_SRC_16_DST 32-bit src to 16-bit dst.
- IX_DMA_32_SRC_8_DST 32-bit src to 8-bit dst.
- IX_DMA_16_SRC_32_DST 16-bit src to 32-bit dst.
- IX_DMA_16_SRC_16_DST 16-bit src to 16-bit dst.
- IX_DMA_16_SRC_8_DST 16-bit src to 8-bit dst.
- IX_DMA_8_SRC_32_DST 8-bit src to 32-bit dst.
- IX_DMA_8_SRC_16_DST 8-bit src to 16-bit dst.
- IX_DMA_8_SRC_8_DST 8-bit src to 8-bit dst.
- IX_DMA_8_SRC_BURST_DST 8-bit src to burst dst - Not supported for fixed destination address.
- IX_DMA_16_SRC_BURST_DST 16-bit src to burst dst - Not supported for fixed destination address.
- IX_DMA_32_SRC_BURST_DST 32-bit src to burst dst - Not supported for fixed destination address.
- IX_DMA_BURST_SRC_8_DST burst src to 8-bit dst - Not supported for fixed source address.
- IX_DMA_BURST_SRC_16_DST burst src to 16-bit dst - Not supported for fixed source address.
- IX_DMA_BURST_SRC_32_DST burst src to 32-bit dst - Not supported for fixed source address.
- IX_DMA_BURST_SRC_BURST_DST burst src to burst dst - Not supported for fixed source and destination address.

## A.3.11 DMA Access Driver (IxDmaAcc) API

The public API for the Intel® IXP42X product line IxDmaAcc component.

### A.3.11.1 Defines

- #define IX_DMA_REQUEST_FULL 16

  DMA request queue is full This constant is a return value used to tell the user that the IxDmaAcc queue is full.

### A.3.11.2 Typedefs

- typedef UINT32 IxDmaAccRequestId

  DMA Request Id type.

- typedef void(* IxDmaAccDmaCompleteCallback )(IxDmaReturnStatus status)

  DMA completion notification This function is called to notify a client that the DMA has been completed.

### A.3.11.3 Functions

- PUBLIC IX_STATUS ixDmaAccInit (IxNpeDlNpeId npeId)

  Initialize the DMA Access component This function will initialize the DMA Access component internals.

- PUBLIC IxDmaReturnStatus ixDmaAccDmaTransfer (IxDmaAccDmaCompleteCallback callback, UINT32 SourceAddr, UINT32 DestinationAddr, UINT16 TransferLength, IxDmaTransferMode TransferMode, IxDmaAddressingMode AddressingMode, IxDmaTransferWidth TransferWidth)

  Perform DMA transfer This function will perform DMA transfer between devices within the Intel® IXP42X product line memory map.

- PUBLIC IX_STATUS ixDmaAccShow (void)

  Display some component information for debug purposes Show some internal operation information relating to the DMA service. At a minimum the following will show. -the number of the DMA pend (in queue).

### A.3.11.4 Detailed Description

The public API for the Intel® IXP42X product line IxDmaAcc component.

### A.3.11.5 Define Documentation

#### A.3.11.5.1 #define IX_DMA_REQUEST_FULL 16

DMA request queue is full This constant is a return value used to tell the user that the IxDmaAcc queue is full.

## A.3.11.6 Typedef Documentation

### A.3.11.6.1 typedef void(..)

typedef void(* IxDmaAccDmaCompleteCallback)(IxDmaReturnStatus status)

DMA completion notification This function is called to notify a client that the DMA has been completed.

#### Parameters

***IxDmaReturnStatus*** — Status reporting to client.

## A.3.11.7 Function Documentation

### A.3.11.7.1 ixDmaAccDmaTransfer (..)

ixDmaAccDmaTransfer (IxDmaAccDmaCompleteCallback callback, UINT32 SourceAddr, UINT32 DestinationAddr, UINT16 TransferLength, IxDmaTransferMode TransferMode, IxDmaAddressingMode AddressingMode, IxDmaTransferWidth TransferWidth)

Perform DMA transfer This function will perform DMA transfer between devices within the Intel® IXP42X product line memory map.

#### Note

The following are restrictions for IxDmaAccDmaTransfer:

- The function is non re-entrant.
- The function assumes host devices are operating in big-endian mode.
- Fixed address does not support burst transfer width.
- Fixed source address to fixed destination address mode is not supported.
- The incrementing source address for expansion bus will not support a burst transfer width and copy and clear mode.

#### Parameters

***IxDmaAccDmaCompleteCallback*** —

***callback*** — function pointer to be stored and called when the DMA transfer is completed. This cannot be NULL.

***ixDmaSourceAddr*** — Starting address of DMA source. Must be a valid Intel® IXP42X product line memory map address.

***ixDmaDestinationAddr*** — Starting address of DMA destination. Must be a valid Intel® IXP42X product line memory map address.

***ixDmaTransferLength*** — Size of DMA data transfer. The range must be from 1-64Kbyte.

***ixDmaTransferMode*** — DMA transfer mode.

***ixDmaAddressingMode*** — DMA addressing mode.

***ixTransferWidth*** — DMA transfer width.

#### Returns

- IX_DMA_SUCCESS — Notification that the DMA request is successful.
- IX_DMA_FAIL — IxDmaAcc not yet initialized or some internal error has occurred.
- IX_DMA_INVALID_TRANSFER_WIDTH — Transfer width is not valid.
- IX_DMA_INVALID_TRANSFER_LENGTH — Transfer length outside of valid range.
- IX_DMA_INVALID_TRANSFER_MODE — Transfer Mode not valid.
- IX_DMA_REQUEST_FIFO_FULL — IxDmaAcc request queue is full.

### A.3.11.7.2 ixDmaAccInit (..)

ixDmaAccInit (IxNpeDlNpeId npeId)

Initialize the DMA Access component This function will initialize the DMA Access component internals.

#### Parameters

***npeId*** — NPE to use for Dma Transfer

#### Returns

- IX_SUCCESS — successfully initialized the component.
- IX_FAIL — Initialization failed for some unspecified internal reason.

### A.3.11.7.3 ixDmaAccShow (void)

Display some component information for debug purposes Show some internal operation information relating to the DMA service. At a minimum the following will show the number of the DMA pend (in queue).

#### Parameters

None

#### Returns

None

## A.3.12 Ethernet Access (IxEthAcc) API

ethAcc is a library that does provides access to the internal Intel® IXP42X product line 10/100Bt Ethernet MACs.

### A.3.12.1 Data Structures

- struct IxEthAccMacAddr

  The IEEE 802.3 Ethernet MAC address structure.

- struct IxEthAccMacAddr

  The IEEE 802.3 Ethernet MAC address structure.

- struct IxEthAccMacAddr

  The IEEE 802.3 Ethernet MAC address structure.

- struct IxEthEthObjStats

  This struct defines the statistics returned by this component. The component returns MIB2 EthObj variables which should are obtained from the hardware or maintained by this component.

- struct IxEthEthObjStats

  This struct defines the statistics returned by this component. The component returns MIB2 EthObj variables which should are obtained from the hardware or maintained by this component.

### A.3.12.2 Defines

- #define IX_ETH_ACC_NUMBER_OF_PORTS (2)
- #define IX_IEEE803_MAC_ADDRESS_SIZE (6)
- #define IX_ETH_ACC_NUM_TX_PRIORITIES (8)

  The number of transmit priorities.

- #define IX_ETHACC_RX_MBUF_MIN_SIZE (2048)

  This defines the required size of MBUF's submitted to the frame receive service.

- #define IXP425_ETH_ACC_MII_MAX_ADDR 32

  This defines the maximum MII address of any attached PHYs.

### A.3.12.3 Typedefs

- typedef void(* IxEthAccPortTxDoneCallback )(UINT32 callbackTag, IX_MBUF *buffer)

  Function prototype for Ethernet Tx Buffer Done callback. Registered via ixEthAccTxBufferDoneCallbackRegister.

- typedef void(* IxEthAccPortRxCallback )(UINT32 callbackTag, IX_MBUF *buffer, IxEthAccPortId portId)

  Function prototype for Ethernet Frame Rx callback. Registered via ixEthAccRxCallbackRegister.

### A.3.12.4 Enumerations

- enum IxEthAccStatus { IX_ETH_ACC_SUCCESS = IX_SUCCESS, IX_ETH_ACC_FAIL = IX_FAIL, IX_ETH_ACC_INVALID_PORT, IX_ETH_ACC_PORT_UNINITIALIZED, IX_ETH_ACC_MAC_UNINITIALIZED, IX_ETH_ACC_INVALID_ARG, IX_ETH_TX_Q_FULL, IX_ETH_ACC_NO_SUCH_ADDR }

  This is an enum to define the Ethernet Access status.

- enum IxEthAccPortId { IX_ETH_PORT_1 = 0, IX_ETH_PORT_2 = 1 }

  This is an emum to define the Intel® IXP42X product line MAC Ethernet device.

- enum IxEthAccTxPriority { IX_ETH_ACC_TX_PRIORITY_0 = 0, IX_ETH_ACC_TX_PRIORITY_1 = 1, IX_ETH_ACC_TX_PRIORITY_2 = 2, IX_ETH_ACC_TX_PRIORITY_3 = 3, IX_ETH_ACC_TX_PRIORITY_4 = 4, IX_ETH_ACC_TX_PRIORITY_5 = 5, IX_ETH_ACC_TX_PRIORITY_6 = 6, IX_ETH_ACC_TX_PRIORITY_7 = 7, IX_ETH_ACC_TX_DEFAULT_PRIORITY = IX_ETH_ACC_TX_PRIORITY_0 }

  enum to submit a frame with relative priority.

- enum IxEthAccDuplexMode { IX_ETH_ACC_FULL_DUPLEX, IX_ETH_ACC_HALF_DUPLEX }

- enum IxEthAccTxSchedulerDiscipline { FIFO_NO_PRIORITY, FIFO_PRIORITY }

  Definition for the port transmit scheduling discipline Definition for the port transmit scheduling discipline.

### A.3.12.5 Functions

- IxEthAccStatus ixEthAccInit (void)

  Initialize the Ethernet Access Service.

- IxEthAccStatus ixEthAccPortInit (IxEthAccPortId portId)

  Initialize an Ethernet MAC Port.

- IxEthAccStatus ixEthAccPortTxFrameSubmit (IxEthAccPortId portId, IX_MBUF *buffer, IxEthAccTxPriority priority)

  This function shall be used to submit MBUFs buffers for transmission on a particular MAC device.

- IxEthAccStatus ixEthAccPortTxDoneCallbackRegister (IxEthAccPortId portId, IxEthAccPortTxDoneCallback txCallbackFn, UINT32 callbackTag)

  This function registers a callback function to facilitate the return of transmit buffers to the user.

- IxEthAccStatus ixEthAccPortRxCallbackRegister (IxEthAccPortId portId, IxEthAccPortRxCallback rxCallbackFn, UINT32 callbackTag)

  The function registered through this function shall be called once per received Ethernet frame.

- IxEthAccStatus ixEthAccPortRxFreeReplenish (IxEthAccPortId portId, IX_MBUF *buffer)

  This function provides buffers for the Ethernet receive path.

- IxEthAccStatus ixEthAccPortEnable (IxEthAccPortId portId)

  Enable a port.

- IxEthAccStatus ixEthAccPortDisable (IxEthAccPortId portId)

  Disable a port.

- IxEthAccStatus ixEthAccPortEnabledQuery (IxEthAccPortId portId, BOOL *enabled)
  Get the enabled state of a port.
- IxEthAccStatus ixEthAccPortPromiscuousModeClear (IxEthAccPortId portId)
  Put the Ethernet MAC device in non-promiscuous mode.
- IxEthAccStatus ixEthAccPortPromiscuousModeSet (IxEthAccPortId portId)
  Put the MAC device in promiscuous mode.
- IxEthAccStatus ixEthAccPortUnicastMacAddressSet (IxEthAccPortId portId, IxEthAccMacAddr *macAddr)
  Configure unicast MAC address for a particular port.
- IxEthAccStatus ixEthAccPortUnicastMacAddressGet (IxEthAccPortId portId, IxEthAccMacAddr *macAddr)
  Get unicast MAC address for a particular MAC port.
- IxEthAccStatus ixEthAccPortMulticastAddressJoin (IxEthAccPortId portId, IxEthAccMacAddr *macAddr)
  ADD a multicast address to the MAC address table.
- IxEthAccStatus ixEthAccPortMulticastAddressLeave (IxEthAccPortId portId, IxEthAccMacAddr *macAddr)
  Remove a multicast address to the MAC address table.
- IxEthAccStatus ixEthAccPortUnicastAddressShow (IxEthAccPortId portId)
  Display unicast address has been configured using ixEthAccUnicastMacAddressSet Display unicast address has been configured using ixEthAccUnicastMacAddressSet.
- void ixEthAccPortMulticastAddressShow (IxEthAccPortId portId)
  Display multicast address which have been configured using ixEthAccMulticastAddressJoin Display multicast address which have been configured using ixEthAccMulticastAddressJoin.
- IxEthAccStatus ixEthAccPortDuplexModeSet (IxEthAccPortId portId, IxEthAccDuplexMode mode)
  Set the duplex mode for the MAC.
- IxEthAccStatus ixEthAccPortDuplexModeGet (IxEthAccPortId portId, IxEthAccDuplexMode *mode)
  Get the duplex mode for the MAC.
- IxEthAccStatus ixEthAccPortTxFrameAppendFCSEnable (IxEthAccPortId portId)
  Enable the appending of Ethernet FCS to all frames submitted to this port.
- IxEthAccStatus ixEthAccPortTxFrameAppendFCSDisable (IxEthAccPortId portId)
  Disable the appending of Ethernet FCS to all frames submitted to this port.
- IxEthAccStatus ixEthAccPortRxFrameAppendFCSEnable (IxEthAccPortId portId)
  Forward frames with FCS included in the receive buffer to the user.
- IxEthAccStatus ixEthAccPortRxFrameAppendFCSDisable (IxEthAccPortId portId)
  Disable the appending of Ethernet FCS to all frames submitted to this port.
- IxEthAccStatus ixEthAccTxSchedulingDisciplineSet (IxEthAccPortId portId, IxEthAccTxSchedulerDiscipline sched)
  Set the port scheduling to one of IxEthAccTxSchedulerDiscipline Set the port scheduling to one of IxEthAccTxSchedulerDiscipline.

- IxEthAccStatus ixEthAccMibIIStatsGet (IxEthAccPortId portId,IxEthEthObjStats *retStats)

  Return the statistics maintained for a port. Return the statistics maintained for a port.

- IxEthAccStatus ixEthAccMibIIStatsGetClear (IxEthAccPortId portId, IxEthEthObjStats *retStats)

  Return and clear the statistics maintained for a port. Return and clear the statistics maintained for a port.

- IxEthAccStatus ixEthAccMibIIStatsClear (IxEthAccPortId portId)

  Clear the statistics maintained for a port. Clear the statistics maintained for a port.

- void ixEthAccStatsShow (IxEthAccPortId portId)

  Display a ports statistics on the standard io console using printf. Display a ports statistics on the standard io console using printf.

- IxEthAccStatus ixEthAccMiiPhyScan (BOOL phyPresent[])

  Scan the MDIO bus for PHYs This function scans PHY addresses 0 through 31, and sets phyPresent[n] to TRUE if a phy is discovered at address n.

- IxEthAccStatus ixEthAccMiiPhyConfig (UINT32 phyAddr, BOOL speed100, BOOL fullDuplex, BOOL autonegotiate)

  Configure a PHY Configure a PHY's speed, duplex and autonegotiation status.

- IxEthAccStatus ixEthAccMiiPhyReset (UINT32 phyAddr)

  Reset a PHY Reset a PHY.

- IxEthAccStatus ixEthAccMiiLinkStatus (UINT32 phyAddr, BOOL *linkUp, BOOL *speed100, BOOL *fullDuplex, BOOL *autoneg)

  Retrieve the current status of a PHY Retrieve the link, speed, duplex and autonegotiation status of a PHY.

- IxEthAccStatus ixEthAccMiiShow (UINT32 phyAddr)

  Display information on a specified PHY Display link status, speed, duplex and Auto Negotiation status.

- IxEthAccStatus ixEthAccMiiStatsShow (UINT32 phyAddr)

  Display detailed information on a specified PHY Display the current values of the standard MII registers for a PHY, in addition to the status information supplied by ixEthAccMiiShow.

### A.3.12.6 Detailed Description

ethAcc is a library that does provides access to the internal Intel® IXP42X product line 10/100Bt Ethernet MACs.

### A.3.12.7 Define Documentation

#### A.3.12.7.1 #define IX_ETH_ACC_NUM_TX_PRIORITIES (8)

The number of transmit priorities.

#### A.3.12.7.2 #define IX_ETHACC_RX_MBUF_MIN_SIZE (2048)

This defines the required size of MBUF's submitted to the frame receive service.

#### A.3.12.7.3 #define IXP425_ETH_ACC_MII_MAX_ADDR 32

This defines the maximum MII address of any attached PHYs.

Max number for PHY address is 31, add on for range checking.

### A.3.12.8 Typedef Documentation

#### A.3.12.8.1 typedef void(..)

typedef void(* IxEthAccPortRxCallback)(UINT32 callbackTag, IX_MBUF *buffer, IxEthAccPortId portId)

Function prototype for Ethernet Frame Rx callback. Registered via ixEthAccRxCallbackRegister.

It is the responsibility of the user function to free any MBUFs which it receives.

- Reentrant — yes, user provided function should be reentrant.
- ISR Callable — yes, user provided function must be callable from an ISR.

**Calling Context**

This callback is called in the context of the queue manager dispatch loop ixQmgrgrDispatcherLoopRun within the Intel® IXP42X product line Queue Manager (IxQMgr) API component. The calling context may be from interrupt or high priority thread. The decision is system specific.

**Parameters**

***callbackTag*** — This tag is that provided when the callback was registered for a particular MAC via ixEthAccRxCallbackRegister. It allows the same callback to be used for multiple MACs.

***mbuf*** — Pointer to the Rx mbuf descriptor.

***portId*** — ID of the port which received the buffer (set to IX_ETH_NUMBER_OF_PORTS if not found)

**Returns**

Void

#### A.3.12.8.2 typedef void(..)

typedef void(* IxEthAccPortTxDoneCallback)( UINT32 callbackTag, IX_MBUF *buffer )

Function prototype for Ethernet Tx Buffer Done callback. Registered via ixEthAccTxBufferDoneCallbackRegister.

This function is called once the previously submitted buffer is no longer required by this service. It may be returned upon successful transmission of the frame or shutdown of port prior to submission. The calling of this registered function is not a guarantee of successful transmission of the buffer.

- Reentrant - yes, user provided function should be reentrant.
- ISR Callable - yes, user provided function must be callable from an ISR.

### Note

Calling Context: This callback is called in the context of the queue manager dispatch loop ixQmgrgrDispatcherLoopRun within the Intel® IXP42X product line Queue Manager (IxQMgr) API component. The calling context may be from interrupt or high priority thread. The decision is system specific.

### Parameters

***callbackTag*** — This tag is that provided when the callback was registered for a particular MAC via ixEthAccRxCallbackRegister. It allows the same callback to be used for multiple MACs.

***mbuf*** — Pointer to the Tx mbuf descriptor.

### Returns

void

## A.3.12.9 Enumeration Type Documentation

### A.3.12.9.1 enum IxEthAccDuplexMode

#### Enumeration values

- IX_ETH_ACC_FULL_DUPLEX  Full duplex operation of the MAC.
- IX_ETH_ACC_HALF_DUPLEX  Half duplex operation of the MAC.

### A.3.12.9.2 enum IxEthAccPortId

This is an emum to define the Intel® IXP42X product line MAC Ethernet device.

#### Enumeration values

- IX_ETH_PORT_1  Ethernet Port 1.
- IX_ETH_PORT_2  Ethernet port 2.

### A.3.12.9.3 enum IxEthAccTxPriority

enum to submit a frame with relative priority.

#### Enumeration values

- IX_ETH_ACC_TX_PRIORITY_0 Lowest Priority submission.
- IX_ETH_ACC_TX_PRIORITY_7 Highest priority submission.
- IX_ETH_ACC_TX_DEFAULT_PRIORITY  By default send all pkts with lowest priority.

### A.3.12.9.4 enum IxEthAccTxSchedulerDiscipline

Definition for the port transmit scheduling discipline Definition for the port transmit scheduling discipline.

Select the port transmit scheduling discipline

- FIFO: No Priority — In this configuration all frames submitted to the access component shall be submitted to the MAC hardware in the strict order in which it was received.

- FIFO: Priority — This shall be a very simple priority mechanism all submitted frames at a higher priority shall be forwarded to Ethernet MAC for transmission before lower priorities. There shall be no fairness mechanisms applied across different priorities. Higher priority frames could starve lower priority frames indefinitely.

## A.3.12.10 Function Documentation

### A.3.12.10.1 ixEthAccInit (void)

Initialize the Ethernet Access Service.

Initialize the Intel® IXP42X product line Ethernet Access Service.

- Reentrant - no
- ISR Callable - no

This should be called once per module initialization.

#### Precondition

The NPE must first be downloaded with the required microcode which supports all required features.

#### Returns

- IxEthAccStatus
- IX_ETH_ACC_SUCCESS
- IX_ETH_ACC_FAIL — Service has failed to initialize.

### A.3.12.10.2 ixEthAccMibIIStatsClear (..)

ixEthAccMibIIStatsClear (IxEthAccPortId portId)

Clear the statistics maintained for a port. Clear the statistics maintained for a port.

- Reentrant — yes
- ISR Callable — no

#### Precondition

#### Parameters

***portId*** — IxEthAccPortId

#### Returns

- IxEthAccStatus
- IX_ETH_ACC_SUCCESS
- IX_ETH_ACC_FAIL — Invalid arguments.
- IX_ETH_ACC_INVALID_PORT — portId is invalid.
- IX_ETH_ACC_PORT_UNINITIALIZED — portId is un-initialized

### A.3.12.10.3 ixEthAccMibIIStatsGet (..)

ixEthAccMibIIStatsGet (IxEthAccPortId portId, IxEthEthObjStats * retStats)

Return the statistics maintained for a port. Return the statistics maintained for a port.

- Reentrant - yes
- ISR Callable - no

#### Precondition

#### Parameters

***portId*** — IxEthAccPortId

***retStats*** — IxEthEthObjStats

#### Note

Please note the user is responsible for cache coherency of the retStat buffer. The data is actually populated via the NPEs. As such cache safe memory should be used in the retStats argument.

#### Returns

- IxEthAccStatus
- IX_ETH_ACC_SUCCESS
- IX_ETH_ACC_FAIL — Invalid arguments.
- IX_ETH_ACC_INVALID_PORT — portId is invalid.
- IX_ETH_ACC_PORT_UNINITIALIZED — portId is un-initialized

### A.3.12.10.4 ixEthAccMibIIStatsGetClear (IxEthAccPortId *portId*, IxEthEthObjStats * *retStats*)

Return and clear the statistics maintained for a port. Return and clear the statistics maintained for a port.

- Reentrant - yes
- ISR Callable - yes

#### Precondition

#### Parameters

***portId*** — IxEthAccPortId

***retStats*** — IxEthEthObjStats

#### Note

Please note the user is responsible for cache coherency of the retStats buffer. The data is actually populated via the NPEs. As such cache safe memory should be used in the retStats argument.

#### Returns

- IxEthAccStatus
- IX_ETH_ACC_SUCCESS
- IX_ETH_ACC_FAIL — invalid arguments.
- IX_ETH_ACC_INVALID_PORT — portId is invalid.
- IX_ETH_ACC_PORT_UNINITIALIZED — portId is un-initialized

### A.3.12.10.5 ixEthAccMiiLinkStatus (..)

ixEthAccMiiLinkStatus (UINT32 phyAddr, BOOL * linkUp, BOOL * speed100, BOOL * fullDuplex, BOOL * autoneg)

Retrieve the current status of a PHY Retrieve the link, speed, duplex and autonegotiation status of a PHY.

- Reentrant — no
- ISR Callable — no

#### Precondition

The MAC on Ethernet Port 2 (NPE C) must be initialized, and generating the MDIO clock.

#### Parameters

***phyAddr*** — the address of the Ethernet PHY (0-31)

***linkUp*** — set to TRUE if the link is up

***speed100*** — set to TRUE indicates 100Mbit/s, FALSE indicates 10Mbit/s

***fullDuplex*** — set to TRUE indicates Full Duplex, FALSE indicates Half Duplex

***autoneg*** — set to TRUE indicates autonegotiation is enabled, FALSE indicates autonegotiation is disabled

#### Returns

- IxEthAccStatus
- IX_ETH_ACC_SUCCESS
- IX_ETH_ACC_FAIL — invalid arguments.

### A.3.12.10.6 ixEthAccMiiPhyConfig (..)

ixEthAccMiiPhyConfig (UINT32 phyAddr, BOOL speed100, BOOL fullDuplex, BOOL autonegotiate)

Configure a PHY Configure a PHY's speed, duplex and autonegotiation status.

- Reentrant — no
- ISR Callable — no

#### Precondition

The MAC on Ethernet Port 2 (NPE C) must be initialized, and generating the MDIO clock.

#### Parameters

***phyAddr***

***speed100*** — set to TRUE for 100Mbit/s operation, FALSE for 10Mbit/s

***fullDuplex*** — set to TRUE for Full Duplex, FALSE for Half Duplex

***autonegotiate*** — set to TRUE to enable autonegotiation

#### Returns

- IxEthAccStatus
- IX_ETH_ACC_SUCCESS
- IX_ETH_ACC_FAIL — Invalid arguments.

### A.3.12.10.7 ixEthAccMiiPhyReset (..)

ixEthAccMiiPhyReset (UINT32 phyAddr)

Reset a PHY Reset a PHY.

- Reentrant — no
- ISR Callable — no

#### Precondition

The MAC on Ethernet Port 2 (NPE C) must be initialized, and generating the MDIO clock.

#### Parameters

***phyAddr*** — the address of the Ethernet PHY (0-31)

#### Returns

- IxEthAccStatus
- IX_ETH_ACC_SUCCESS
- IX_ETH_ACC_FAIL — Invalid arguments.

### A.3.12.10.8 ixEthAccMiiPhyScan (..)

ixEthAccMiiPhyScan (BOOL phyPresent[])

Scan the MDIO bus for PHYs This function scans PHY addresses 0 through 31, and sets phyPresent[n] to TRUE if a phy is discovered at address n.

- Reentrant — no
- ISR Callable — no

#### Precondition

The MAC on Ethernet Port 2 (NPE C) must be initialized, and generating the MDIO clock.

#### Parameters

***phyPresent***

#### Returns

- IxEthAccStatus
- IX_ETH_ACC_SUCCESS
- IX_ETH_ACC_FAIL — Invalid arguments.

### A.3.12.10.9 ixEthAccMiiShow (..)

ixEthAccMiiShow (UINT32 phyAddr)

Display information on a specified PHY Display link status, speed, duplex and Auto Negotiation status.

- Reentrant — no
- ISR Callable — no

#### Precondition

The MAC on Ethernet Port 2 (NPE C) must be initialized, and generating the MDIO clock.

#### Parameters

***phyAddr*** — the address of the Ethernet PHY (0-31)

#### Returns

- IxEthAccStatus
- IX_ETH_ACC_SUCCESS
- IX_ETH_ACC_FAIL — Invalid arguments.

### A.3.12.10.10 ixEthAccMiiStatsShow (..)

ixEthAccMiiStatsShow (UINT32 phyAddr)

Display detailed information on a specified PHY Display the current values of the standard MII registers for a PHY, in addition to the status information supplied by ixEthAccMiiShow.

- Reentrant — no
- ISR Callable — no

#### Precondition

The MAC on Ethernet Port 2 (NPE C) must be initialized, and generating the MDIO clock.

#### Parameters

***phyAddr*** — the address of the Ethernet PHY (0-31)

#### Returns

IxEthAccStatus

- IX_ETH_ACC_SUCCESS
- IX_ETH_ACC_FAIL — Invalid arguments.

### A.3.12.10.11 ixEthAccPortDisable (..)

ixEthAccPortDisable (IxEthAccPortId portId)

Disable a port.

This disables an Ethernet port for both Tx and Rx. Free MBufs are returned to the user via the registered callback when the port is disabled

- Reentrant — yes
- ISR Callable — yes

#### Precondition

The port must be enabled with ixEthAccPortEnable, otherwise this function has no effect

#### Parameters

***portId*** IxEthAccPortId — Port id to act upon.

#### Returns

- IxEthAccStatus
- IX_ETH_ACC_SUCCESS
- IX_ETH_ACC_INVALID_PORT — portId is invalid.
- IX_ETH_ACC_PORT_UNINITIALIZED — portId is not initialized
- IX_ETH_ACC_MAC_UNINITIALIZED — port MAC address is not initialized

### A.3.12.10.12 ixEthAccPortDuplexModeGet (..)

ixEthAccPortDuplexModeGet (IxEthAccPortId portId, IxEthAccDuplexMode * mode)

Get the duplex mode for the MAC. Returns the duplex configuration of the Intel® IXP42X product line MAC.

#### Note

The configuration should match that provisioned on the PHY. See ixEthAccDuplexModeSet

- Reentrant — no
- ISR Callable — no

#### Parameters

***portId*** — IxEthAccPortId

***mode*** — IxEthAccDuplexMode

#### Returns

IxEthAccStatus

- IX_ETH_ACC_SUCCESS
- IX_ETH_ACC_INVALID_PORT — portId is invalid.
- IX_ETH_ACC_PORT_UNINITIALIZED — portId is un-initialized

### A.3.12.10.13 ixEthAccPortDuplexModeSet (..)

ixEthAccPortDuplexModeSet (IxEthAccPortId portId, IxEthAccDuplexMode mode)

Set the duplex mode for the MAC. Configure the Intel® IXP42X product line MAC to either full or half duplex.

#### Note

The configuration should match that provisioned on the PHY.

- Reentrant — no
- ISR Callable — no

#### Parameters

***portId*** — IxEthAccPortId

***mode*** — IxEthAccDuplexMode

#### Returns

- IxEthAccStatus
- IX_ETH_ACC_SUCCESS
- IX_ETH_ACC_INVALID_PORT — portId is invalid.
- IX_ETH_ACC_PORT_UNINITIALIZED — portId is un-initialized

### A.3.12.10.14 ixEthAccPortEnable (..)

ixEthAccPortEnable (IxEthAccPortId portId)

Enable a port.

This enables an Ethernet port for both Tx and Rx.

- Reentrant — yes
- ISR Callable — yes

#### Precondition

The port must first be initialized via ixEthAccPortInit and the MAC address must be set using ixEthAccUnicastMacAddressSet before enabling it

#### Parameters

***portId*** — IxEthAccPortId — Port id to act upon.

#### Returns

- IxEthAccStatus
- IX_ETH_ACC_SUCCESS
- IX_ETH_ACC_INVALID_PORT — portId is invalid.
- IX_ETH_ACC_PORT_UNINITIALIZED — portId is not initialized.
- IX_ETH_ACC_MAC_UNINITIALIZED — port MAC address is not initialized.

### A.3.12.10.15 ixEthAccPortEnabledQuery (..)

ixEthAccPortEnabledQuery (IxEthAccPortId portId, BOOL * enabled)

Get the enabled state of a port. Return the enabled state of the port.

- Reentrant — yes
- ISR Callable — yes

#### Precondition

The port must first be initialized via ixEthAccPortInit

#### Parameters

***portId*** — IxEthAccPortId — Port id to act upon.

***enabled*** — BOOL — Location to store the state of the port.

#### Returns

- IxEthAccStatus
- IX_ETH_ACC_SUCCESS
- IX_ETH_ACC_INVALID_PORT — portId is invalid.

### A.3.12.10.16 ixEthAccPortInit (..)

ixEthAccPortInit (IxEthAccPortId portId)

Initialize an Ethernet MAC Port. Initialize the NPE/Ethernet MAC hardware. Verify NPE downloaded and operational. The NPE shall be available for usage once this API returns.

- Reentrant - no
- ISR Callable - no

#### Notes

This should be called once per mac device. The NPE/MAC shall be in disabled state after init.

Precondition — The component must be initialized via ixEthAccInit The NPE must first be downloaded with the required microcode which supports all required features.

Dependant on Services — (Must be initialized before using this service may be initialized) ixNPEmh - NPE Message handling service. ixQmgr - Queue Manager component.

#### Parameters

***portId*** — IxEthAccPortId

#### Returns

- IxEthAccStatus
- IX_ETH_ACC_SUCCESS
- IX_ETH_ACC_FAIL — NPE processor has failed to initialize.
- IX_ETH_ACC_INVALID_PORT — portId is invalid.

### A.3.12.10.17 ixEthAccPortMulticastAddressJoin (..)

ixEthAccPortMulticastAddressJoin (IxEthAccPortId portId, IxEthAccMacAddr * macAddr)

ADD a multicast address to the MAC address table. ADD a multicast address to the MAC address table.

#### Note

Due to the operation of the Ethernet MAC multicast filtering mechanism, frames which do not have a multicast destination address which were provisioned via this API may be forwarded to the NPEs. This is a result of the hardware comparison algorithm used in the destination mac address logic within the Ethernet MAC.

#### See also:

*Intel® IXP425 Network Processor Based on Intel® XScale™ Microarchitecture Component Specification* (11725)

- Reentrant — yes
- ISR Callable — no

#### Parameters

***portId*** — Ethernet port id.

***macAddr*** — Ethernet Mac address.

#### Returns

- IxEthAccStatus
- IX_ETH_ACC_SUCCESS
- IX_ETH_ACC_FAIL — Error writing to the MAC registers
- IX_ETH_ACC_INVALID_PORT — portId is invalid.
- IX_ETH_ACC_PORT_UNINITIALIZED — portId is un-initialized

### A.3.12.10.18 ixEthAccPortMulticastAddressLeave (..)

ixEthAccPortMulticastAddressLeave (IxEthAccPortId portId, IxEthAccMacAddr * macAddr)

Remove a multicast address to the MAC address table. Remove a multicast address to the MAC address table.

- Reentrant — yes
- ISR Callable — no

#### Parameters

***portId*** — Ethernet port id.

***macAddr*** — Ethernet Mac address.

#### Returns

- IxEthAccStatus
- IX_ETH_ACC_SUCCESS
- IX_ETH_ACC_NO_SUCH_ADDR — Failed if MAC address was not in the table.
- IX_ETH_ACC_INVALID_PORT — portId is invalid.
- IX_ETH_ACC_PORT_UNINITIALIZED — portId is un-initialized

### A.3.12.10.19 ixEthAccPortMulticastAddressShow (..)

ixEthAccPortMulticastAddressShow (IxEthAccPortId portId)

Display multicast address which have been configured using ixEthAccMulticastAddressJoin Display multicast address which have been configured using ixEthAccMulticastAddressJoin.

- Reentrant — yes
- ISR Callable — no

#### Parameters

***portId*** — Ethernet port id.

#### Returns

Void

### A.3.12.10.20 ixEthAccPortPromiscuousModeClear (..)

ixEthAccPortPromiscuousModeClear (IxEthAccPortId portId)

Put the Ethernet MAC device in non-promiscuous mode. In non-promiscuous mode the MAC will filter all frames other than destination MAC address which matches the following criteria:

- — Unicast address provisioned via ixEthAccUnicastMacAddressSet
- — All broadcast frames.
- — Multicast addresses provisioned via ixEthAccMulticastAddressJoin

- Reentrant — yes
- ISR Callable — no

**See also:**

ixEthAccPortPromiscuousModeSet

**Parameters**

***portId*** — Ethernet port id.

**Returns**

- IxEthAccStatus
- IX_ETH_ACC_SUCCESS
- IX_ETH_ACC_INVALID_PORT: portId is invalid.
- IX_ETH_ACC_PORT_UNINITIALIZED: portId is un-initialized

### A.3.12.10.21 ixEthAccPortPromiscuousModeSet (..)

ixEthAccPortPromiscuousModeSet (IxEthAccPortId portId

Put the MAC device in promiscuous mode. If the device is in promiscuous mode then all received frames shall be forwarded to the NPE for processing.

- Reentrant — yes
- ISR Callable — no

**See also:**

ixEthAccPortPromiscuousModeClear

**Parameters**

***portId*** — Ethernet port id.

**Returns**

- IxEthAccStatus
- IX_ETH_ACC_SUCCESS
- IX_ETH_ACC_INVALID_PORT: portId is invalid.
- IX_ETH_ACC_PORT_UNINITIALIZED: portId is un-initialized

### A.3.12.10.22 ixEthAccPortRxCallbackRegister (..)

ixEthAccPortRxCallbackRegister (IxEthAccPortId portId, IxEthAccPortRxCallback rxCallbackFn, UINT32 callbackTag)

The function registered through this function shall be called once per received Ethernet frame.

This function will dispatch a predefined number of frames to the user level via the provided function. The invocation shall be made for each frame dequeued from the Ethernet QM queue. The user is required to free any MBUFs supplied via this callback. In addition the registered callback must free up MBUF's from the receive free queue when the port is disabled

If called several times the latest callback shall be registered for a particular port.

- Reentrant — yes
- ISR Callable — yes

#### Precondition

The port must be enabled (see ixEthAccPortEnable) before this callback will be invoked

#### Parameters

***portId*** — Register callback for a particular MAC device.

***rxCallbackFn*** — IxEthAccRxCallbackFn - Function to be called when Ethernet frames are available.

***callbackTag*** — This tag shall be provided to the callback function.

#### Returns

- IxEthAccStatus
- IX_ETH_ACC_SUCCESS
- IX_ETH_ACC_INVALID_PORT — portId is invalid.
- IX_ETH_ACC_PORT_UNINITIALIZED — portId is un-initialized.
- IX_ETH_ACC_INVALID_ARG — Argument other than portId is invalid.

### A.3.12.10.23 ixEthAccPortRxFrameAppendFCSDisable (..)

ixEthAccPortRxFrameAppendFCSDisable (IxEthAccPortId portId)

Disable the appending of Ethernet FCS to all frames submitted to this port.

Do not forward the FCS portion of the received Ethernet frame to the user. The FCS is striped from the receive buffer. Frame FCS validity checks are still carried out on all received frames. This is the default behavior of the component. Do not change this behavior while the port is enabled.

- Reentrant — yes
- ISR Callable — no

#### Parameters

***portId*** — IxEthAccPortId

#### Returns

- IxEthAccStatus
- IX_ETH_ACC_SUCCESS
- IX_ETH_ACC_INVALID_PORT — portId is invalid.
- IX_ETH_ACC_PORT_UNINITIALIZED — portId is un-initialized

### A.3.12.10.24 ixEthAccPortRxFrameAppendFCSEnable (..)

ixEthAccPortRxFrameAppendFCSEnable (IxEthAccPortId portId)

Forward frames with FCS included in the receive buffer to the user.

Enable the appending of Ethernet FCS to all frames submitted to this port. This is the default behavior of the access component. The Frame length received will include the FCS (ie, a minimum sized ethernet frame shall have a length of 64bytes).

- Reentrant — yes
- ISR Callable — no

#### Parameters

***portId*** — IxEthAccPortId

#### Returns

- IxEthAccStatus
- IX_ETH_ACC_SUCCESS
- IX_ETH_ACC_INVALID_PORT — portId is invalid.
- IX_ETH_ACC_PORT_UNINITIALIZED — portId is un-initialized

### A.3.12.10.25 ixEthAccPortRxFreeReplenish (..)

ixEthAccPortRxFreeReplenish (IxEthAccPortId portId, IX_MBUF * buffer)

This function provides buffers for the Ethernet receive path.

This component does not have a buffer management mechanisms built in. All Rx buffers must be supplied to it via this interface.

- Reentrant — yes
- ISR Callable — yes

#### Note

If the buffer replenish operation fails it is the responsibility of the user to free the buffer. Sufficient buffers must be supplied to the component to maintain receive throughput and avoid rx buffer underflow conditions. To meet this goal, It is expected that the user preload the component with a sufficient number of buffers prior to enabling the NPE Ethernet receive path.

### Parameters

***portId*** — Provide buffers only to specific Rx MAC.

***mbuf*** — Provide an MBUF to the Ethernet receive mechanism.

### Returns

- IxEthAccStatus
- IX_ETH_ACC_SUCCESS
- IX_ETH_ACC_FAIL — Buffer has was not able to queue the buffer in the receive service.
- IX_ETH_ACC_INVALID_PORT — portId is invalid.
- IX_ETH_ACC_PORT_UNINITIALIZED — portId is un-initialized

## A.3.12.10.26 ixEthAccPortTxDoneCallbackRegister (..)

ixEthAccPortTxDoneCallbackRegister (IxEthAccPortId portId, IxEthAccPortTxDoneCallback txCallbackFn, UINT32 callbackTag)

This function registers a callback function to facilitate the return of transmit buffers to the user.

This function registers the transmit buffer done function callback for a particular port.

The registered callback function is called once the previously submitted buffer is no longer required by this service. It may be returned upon successful transmission of the frame or shutdown of port prior to submission. The calling of this registered function is not a guarantee of successful transmission of the buffer.

If called several times the latest callback shall be registered for a particular port.

- Reentrant — yes
- ISR Callable — yes

### Precondition

The port must be initialized via ixEthAccPortInit

### Parameters

***portId*** — Register callback for a particular MAC device.

***txCallbackFn*** — IxEthAccTxBufferDoneCallbackFn - Function to be called to return transmit buffers to the user.

***callbackTag*** — This tag shall be provided to the callback function.

### Returns

- IxEthAccStatus
- IX_ETH_ACC_SUCCESS
- IX_ETH_ACC_INVALID_PORT — portId is invalid.

- IX_ETH_ACC_PORT_UNINITIALIZED — portId is un-initialized
- IX_ETH_ACC_INVALID_ARG — Argument other than portId is invalid.

### A.3.12.10.27 ixEthAccPortTxFrameAppendFCSDisable (..)

ixEthAccPortTxFrameAppendFCSDisable (IxEthAccPortId portId)

Disable the appending of Ethernet FCS to all frames submitted to this port.

Disable the appending of Ethernet FCS to all frames submitted to this port. This is not the default behavior of the access component.

#### Note

Since the FCS is not appended to the frame it is expected that the frame submitted to the component includes a valid FCS at the end of the data, although this will not be validated. The component shall forward the frame to the Ethernet MAC WITHOUT modification. Do not change this behavior while the port is enabled.

- Reentrant — yes
- ISR Callable — no

#### Parameters

***portId*** — IxEthAccPortId

#### Returns

- IxEthAccStatus
- IX_ETH_ACC_SUCCESS
- IX_ETH_ACC_INVALID_PORT — portId is invalid.
- IX_ETH_ACC_PORT_UNINITIALIZED — portId is un-initialized.

### A.3.12.10.28 ixEthAccPortTxFrameAppendFCSEnable (..)

ixEthAccPortTxFrameAppendFCSEnable (IxEthAccPortId portId)

Enable the appending of Ethernet FCS to all frames submitted to this port.

Enable the appending of Ethernet FCS to all frames submitted to this port. This is the default behavior of the access component. Do not change this behavior while the port is enabled.

- Reentrant — yes
- ISR Callable — no

#### Parameters

***portI*** — IxEthAccPortId

#### Returns

- IxEthAccStatus
- IX_ETH_ACC_SUCCESS

- IX_ETH_ACC_INVALID_PORT — portId is invalid.
- IX_ETH_ACC_PORT_UNINITIALIZED — portId is un-initialized

### A.3.12.10.29 ixEthAccPortTxFrameSubmit (..)

ixEthAccPortTxFrameSubmit (IxEthAccPortId portId, IX_MBUF * buffer, IxEthAccTxPriority priority)

This function shall be used to submit MBUFs buffers for transmission on a particular MAC device.

This function shall be used to submit MBUFs buffers for transmission on a particular MAC device. The buffer shall be freed automatically once the frame has been transmitted. The only alterations made to the buffer are assocuated with the next packet chain pointer. This is used to internally queue frames in the service. Buffers shall be not queued for transmission if the port is disabled. The port can be enabled using ixEthAccPortEnable

- Reentrant — yes
- ISR Callable — yes

#### Precondition

ixEthAccPortTxDoneCallbackRegister must be called to register a function to allow this service to return the buffer to the calling service.

#### Note

If the buffer submit fails for any reason the user has retained ownership of the buffer.

#### Parameters

***portId*** — MAC port ID to transmit Ethernet frame on.

***mbuf*** — pointer to an MBUF formatted buffer. Chained buffers are supported for transmission.

***priority*** — IxEthAccTxPriority

#### Returns

- IxEthAccStatus
- IX_ETH_ACC_SUCCESS
- IX_ETH_ACC_FAIL — Failed to queue frame for transmission.
- IX_ETH_ACC_INVALID_PORT — portId is invalid.
- IX_ETH_ACC_PORT_UNINITIALIZED — portId is un-initialized.

### A.3.12.10.30 ixEthAccPortUnicastAddressShow (..)

ixEthAccPortUnicastAddressShow (IxEthAccPortId portId)

Display unicast address has been configured using ixEthAccUnicastMacAddressSet Display unicast address has been configured using ixEthAccUnicastMacAddressSet.

- Reentrant — yes
- ISR Callable — no

#### Parameters

***portId*** — Ethernet port ID.

#### Returns

void

### A.3.12.10.31 ixEthAccPortUnicastMacAddressGet (..)

ixEthAccPortUnicastMacAddressGet (IxEthAccPortId portId, IxEthAccMacAddr * macAddr)

Get unicast MAC address for a particular MAC port.

#### Precondition

The MAC address must first be set via ixEthAccMacPromiscuousModeSet If the MAC address has not been set, the function will return a IX_ETH_ACC_MAC_UNINITIALIZED status

- Reentrant — yes
- ISR Callable — no

#### Parameters

***portId*** — Ethernet port id.

***macAddr*** — Ethernet MAC address.

#### Returns

- IxEthAccStatus

### A.3.12.10.32 ixEthAccPortUnicastMacAddressSet (..)

ixEthAccPortUnicastMacAddressSet (IxEthAccPortId portId, IxEthAccMacAddr * macAddr)

Configure unicast MAC address for a particular port.

- Reentrant — yes
- ISR Callable — no

#### Parameters

***portId*** — Ethernet port id.

***macAddr*** — Ethernet Mac address.

#### Returns

- IxEthAccStatus
- IX_ETH_ACC_SUCCESS
- IX_ETH_ACC_INVALID_PORT — portId is invalid.
- IX_ETH_ACC_PORT_UNINITIALIZED — portId is un-initialized.

### A.3.12.10.33 ixEthAccStatsShow (..)

ixEthAccStatsShow (IxEthAccPortId portId)

Display a ports statistics on the standard io console using printf. Display a ports statistics on the standard io console using printf.

- Reentrant — no
- ISR Callable — no

#### Parameters

***portId*** — IxEthAccPortId

#### Returns

void

### A.3.12.10.34 ixEthAccTxSchedulingDisciplineSet (..)

ixEthAccTxSchedulingDisciplineSet (IxEthAccPortId portId, IxEthAccTxSchedulerDiscipline sched)

Set the port scheduling to one of IxEthAccTxSchedulerDiscipline Set the port scheduling to one of IxEthAccTxSchedulerDiscipline.

The default behavior of the component is FIFO_NO_PRIORITY.

- Reentrant — yes
- ISR Callable — yes

#### Parameters

***portId*** — IxEthAccPortId

***sched*** — IxEthAccTxSchedulerDiscipline

#### Returns

- IxEthAccStatus
- IX_ETH_ACC_SUCCESS — Set appropriate discipline.
- IX_ETH_ACC_FAIL — Invalid/unsupported discipline.
- IX_ETH_ACC_INVALID_PORT — portId is invalid.
- IX_ETH_ACC_PORT_UNINITIALIZED — portId is un-initialized

## A.3.13 Ethernet Access Fast Path (IxEthAccFpathDep) API

The dependencies common to the Intel® IXP42X product line Ethernet Access (IxEthAcc) API component and the IxFpath component.

### A.3.13.1 Defines

- #define IX_ETH_ACC_FPATH_AWARE 0

  This define indicates that the ethernet access service is used in a fast path enabled system.

### A.3.13.2 Detailed Description

The dependencies common to the Intel® IXP42X product line Ethernet Access (IxEthAcc) API component and the IxFpath component.

### A.3.13.3 Define Documentation

#### A.3.13.3.1 #define IX_ETH_ACC_FPATH_AWARE 0

This define indicates that the ethernet access service is used in a fast path enabled system.

When fast path ATM to Ethernet is enabled, the Ethernet transmit Q for Eth Tx port becomes shared between EthAcc and Npe-A. 64 entries of the transmit Q are reserved for use by NpeA and 64 for used by EthAcc. So when IX_ETH_ACC_FPATH_AWARE is set to TRUE, the TX queue depth available to EthAcc is reduced by half. This will obviously have a potentially major impact on the performance of the component and the timing dynamics of the passage of buffers through the system.

IMPLICATIONS - FpathAcc; The maximum number of buffers which can be provisioned for the fast path must be matched with number of entries to be reserved for the fast path on the Eth Tx queue. This is currently hard coded within FpathAcc.

IMPLICATIONS - EthAcc; The EthAcc Tx code must make a significant modification to the behavior such that enough room is maintained in the TX queue for Npe-A to write its entire quota of fast path buffers.

Furthermore, in normal operation EthAcc by default relies on the TX Q empty callback to trigger submissions of buffers previously queued on software queues. However when operating in a system in which Fast path is enabled, the TX Q may never become empty, as it may be kept non-empty by submissions from the NPE-A. Hence Tx Q empty trigger cannot be used. Instead the TX Done callback is used to trigger such submissions.

## A.3.14 Ethernet Database (IxEthDB) API

ethDB is a library that does provides a MAC address database learning/filtering capability.

### A.3.14.1 Data Structures

- struct IxEthDBMacAddr
  The IEEE 802.3 Ethernet MAC address structure.
- struct IxEthDBMacAddr
  The IEEE 802.3 Ethernet MAC address structure.

### A.3.14.2 Defines

- #define INLINE __inline__
- #define IX_ETH_DB_PRIVATE
- #define IX_ETH_DB_PUBLIC
- #define IX_ETH_DB_PORT_ID_TO_NPE(id) (id + 1)
  port ID => message handler NPE id conversion (0 => NPE_B, 1 => NPE_C).
- #define IX_ETH_DB_NPE_TO_PORT_ID(npe) (npe - 1)
  message handler NPE id => port ID conversion (NPE_B => 0, NPE_C => 1).
- #define IX_IEEE803_MAC_ADDRESS_SIZE (6)
  The size of the MAC address.
- #define IX_ETH_DB_MAINTENANCE_TIME (1 * 60)
  The ixEthDBDatabaseMaintenance must be called by the user at a frequency of IX_ETH_DB_MAINTENANCE_TIME.
- #define IX_ETH_DB_LEARNING_ENTRY_AGE_TIME (15 * 60 )
  The define specifies the filtering database age entry time. Static entries older than IX_ETH_DB_LEARNING_ENTRY_AGE_TIME +/- IX_ETH_DB_MAINTENANCE_TIME shall be removed.

### A.3.14.3 Typedefs

- typedef UINT32 IxEthDBPortId
  Definition of an Intel® IXP42X product line port.
- typedef UINT32 IxEthDBPortMap
  Port dependency map definition.

### A.3.14.4 Enumerations

- enum IxEthDBStatus { IX_ETH_DB_SUCCESS = IX_SUCCESS, IX_ETH_DB_FAIL = IX_FAIL, IX_ETH_DB_INVALID_PORT, IX_ETH_DB_PORT_UNINITIALIZED, IX_ETH_DB_MAC_UNINITIALIZED, IX_ETH_DB_INVALID_ARG, IX_ETH_DB_NO_SUCH_ADDR, IX_ETH_DB_NOMEM, IX_ETH_DB_BUSY, IX_ETH_DB_END }
  Ethernet database status.

### A.3.14.5 Functions

- IX_ETH_DB_PUBLIC IxEthDBStatus ixEthDBInit (void)

  Initializes the Ethernet learning/filtering database.

- IX_ETH_DB_PUBLIC void ixEthDBPortInit (IxEthDBPortId portID)

  Initializes a port.

- IX_ETH_DB_PUBLIC IxEthDBStatus ixEthDBPortEnable (IxEthDBPortId portID)

  Enable a port.

- IX_ETH_DB_PUBLIC IxEthDBStatus ixEthDBPortDisable (IxEthDBPortId portID)

  Disable processing on a port.

- IX_ETH_DB_PUBLIC IxEthDBStatus ixEthDBPortAddressSet (IxEthDBPortId portID, IxEthDBMacAddr *macAddr)

  Set a port MAC address.

- IxEthDBStatus ixEthDBFilteringStaticEntryProvision (IxEthDBPortId portId, IxEthDBMacAddr *macAddr)

  Statically Populate the Filtering database with a mac address. Statically populate the filtering database with a mac address. The entry shall not be subject to aging. If there is an entry (static or dynamic) with the corresponding MAC address on any port this entry shall take precedence. All other entries for the MAC address shall be removed.

- IxEthDBStatus ixEthDBFilteringDynamicEntryProvision (IxEthDBPortId portId, IxEthDBMacAddr *macAddr)

  Populate the filtering database with a dynamic mac address entry.

- IxEthDBStatus ixEthDBFilteringEntryDelete (IxEthDBMacAddr *macAddr)

  Remove a MAC address entry from the filtering database.

- IxEthDBStatus ixEthDBFilteringPortSearch (IxEthDBPortId portId, IxEthDBMacAddr *macAddr)

  Search the Filtering data base for portId for the macAddr.

- IxEthDBStatus ixEthDBFilteringDatabaseSearch (IxEthDBPortId *portId, IxEthDBMacAddr *macAddr)

  Search the filtering database for a MAC address.

- IxEthDBStatus ixEthDBFilteringPortUpdatingSearch (IxEthDBPortId *portID, IxEthDBMacAddr *macAddr)

  Search the filtering database for a MAC address, return the port and reset the age.

- IxEthDBStatus ixEthDBPortAgingDisable (IxEthDBPortId port)

  Disable the aging function for a specific port.

- IxEthDBStatus ixEthDBPortAgingEnable (IxEthDBPortId port)

  Enable the aging function for a specific port.

- void ixEthDBDatabaseMaintenance (void)

  Required in order to do maintenance operations on the filtering database.

- IxEthDBStatus ixEthDBFilteringDatabaseShow (IxEthDBPortId portId)

  This function displays the Mac Ethernet MAC address filtering tables. It displays items such as MAC address/port, dynamic/static, entry age for all Mac Ethernet ports. It uses printf to display the details.

- void ixEthDBFilteringDatabaseShowAll (void)

  Displays the MAC address recorded in the filtering database, grouped by port ID.

## A.3.14.6 Detailed Description

ethDB is a library that does provides a MAC address database learning/filtering capability.

## A.3.14.7 Define Documentation

### A.3.14.7.1 #define IX_ETH_DB_LEARNING_ENTRY_AGE_TIME (15 * 60 )

The define specifies the filtering database age entry time. Static entries older than IX_ETH_DB_LEARNING_ENTRY_AGE_TIME +/- IX_ETH_DB_MAINTENANCE_TIME shall be removed.

### A.3.14.7.2 #define IX_ETH_DB_MAINTENANCE_TIME (1 * 60)

The ixEthDBDatabaseMaintenance must be called by the user at a frequency of IX_ETH_DB_MAINTENANCE_TIME.

## A.3.14.8 Enumeration Type Documentation

### A.3.14.8.1 enum IxEthDBStatus

Ethernet database status.

**Enumeration values**

- IX_ETH_DB_NO_SUCH_ADDR Address not found for search or delete operations.
- IX_ETH_DB_NOMEM Learning database memory full.
- IX_ETH_DB_BUSY Learning database is busy.
- IX_ETH_DB_END Database browser passed the end.

## A.3.14.9 Function Documentation

### A.3.14.9.1 ixEthDBDatabaseMaintenance (void)

Required in order to do maintenance operations on the filtering database.

In order to do maintenance functions on the database this function must be called with a period of @IX_ETH_DB_MAINTENANCE_TIME seconds. It must be called regardless of whether learning function is enabled or not.

- Reentrant — no
- ISR Callable — no

#### Parameters

None

#### Returns

void

### A.3.14.9.2 ixEthDBFilteringDatabaseSearch (..)

ixEthDBFilteringDatabaseSearch (IxEthDBPortId * portId, IxEthDBMacAddr * macAddr)

Search the filtering database for a MAC address.

Search the filtering database for a MAC address. The function returns the portId for the MAC address. If no match is found the function returns IX_ETH_DB_FAIL. The portId returned is only valid if the function finds a match.

- Reentrant — yes
- ISR Callable — no

#### Parameters

***portId*** — IxEthDBPortId

***macAddr*** — IxEthDBMacAddr

#### Return

- IX_ETH_DB_SUCCESS  The MAC address was found.
- IX_ETH_DB_NO_SUCH_ADDR  The MAC address was not found.

### A.3.14.9.3 ixEthDBFilteringDatabaseShow (..)

ixEthDBFilteringDatabaseShow (IxEthDBPortId portId)

This function displays the Mac Ethernet MAC address filtering tables. It displays items such as MAC address/port, dynamic/static, entry age for all Mac Ethernet ports. It uses printf to display the details.

- Reentrant — yes
- ISR Callable — no

#### Parameters

***portId*** — IxEthDBPortId

#### Returns

- IxEthDBStatus
- IX_ETH_DB_SUCCESS
- IX_ETH_DB_INVALID_PORT — portId is invalid.
- IX_ETH_DB_PORT_UNINITIALIZED — portId is un-initialized

#### A.3.14.9.4 ixEthDBFilteringDatabaseShowAll (void)

Displays the MAC address recorded in the filtering database, grouped by port ID.

- Reentrant — yes
- ISR Callable — no

#### A.3.14.9.5 ixEthDBFilteringDynamicEntryProvision (..)

ixEthDBFilteringDynamicEntryProvision (IxEthDBPortId portId, IxEthDBMacAddr * macAddr)

Populate the filtering database with a dynamic mac address entry.

Populate the filtering database with a dynamic mac address entry. This entry will be subject to normal aging function. If there is an entry (static or dynamic) with the corresponding MAC address on any port this entry shall take precedence. All other entries for the MAC address shall be removed.

- Reentrant — yes
- ISR Callable — no

**Parameters**

***portId*** — IxEthDBPortId

***macAddr*** — IxEthDBMacAddr

**Returns**

- IxEthDBStatus
- IX_ETG_ACC_SUCCESS
- IX_ETH_DB_FAIL — Failed to populate the entry in the database.
- IX_ETH_DB_INVALID_PORT — portId is invalid.
- IX_ETH_DB_PORT_UNINITIALIZED — portId is un-initialized.

#### A.3.14.9.6 ixEthDBFilteringEntryDelete (..)

ixEthDBFilteringEntryDelete (IxEthDBMacAddr * macAddr)

Remove a MAC address entry from the filtering database.

- Reentrant — yes
- ISR Callable — no

**Parameters**

***macAddr*** — IxEthDBMacAddr

**Returns**

- IxEthDBStatus
- IX_ETH_DB_SUCCESS
- IX_ETH_DB_NO_SUCH_ADDR — Failed to remove the address as it was not in the table.

### A.3.14.9.7 ixEthDBFilteringPortSearch (..)

ixEthDBFilteringPortSearch (IxEthDBPortId portId, IxEthDBMacAddr * macAddr)

Search the Filtering data base for portId for the macAddr.

This functions Search the Filtering data base for a specific port. If the mac address is found on the port the function returns IX_ETH_DB_SUCCESS. If the port does not contain an entry for the port it returns IX_ETH_DB_FAIL

- Reentrant — yes
- ISR Callable — no

#### Parameters

***portId*** — IxEthDBPortId

***macAddr*** — IxEthDBMacAddr

#### Returns

- IxEthDBStatus
- IX_ETH_DB_SUCCESS — MAC address was found.
- IX_ETH_DB_NO_SUCH_ADDR — MAC address was not found.
- IX_ETH_DB_INVALID_PORT — portId is invalid.
- IX_ETH_DB_PORT_UNINITIALIZED — portId is un-initialized

### A.3.14.9.8 ixEthDBFilteringPortUpdatingSearch (..)

ixEthDBFilteringPortUpdatingSearch (IxEthDBPortId * portID, IxEthDBMacAddr * macAddr)

Search the filtering database for a MAC address, return the port and reset the age.

This function searches the filtering database for the given MAC address. If found, the given portID location is set to the portID found in the database and the MAC address age is reset to 0. The portID is valid only if the function has found a match.

- Reentrant — yes
- ISR Callable — no

#### Return

- IX_ETH_DB_SUCCESS — the MAC address was found
- IX_ETH_DB_NO_SUCH_ADDR — the MAC address was not found

### A.3.14.9.9 ixEthDBFilteringStaticEntryProvision (..)

ixEthDBFilteringStaticEntryProvision (IxEthDBPortId portId, IxEthDBMacAddr * macAddr)

Statically Populate the Filtering database with a mac address. Statically populate the filtering database with a mac address. The entry shall not be subject to aging. If there is an entry (static or dynamic) with the corresponding MAC address on any port this entry shall take precedence. All other entries for the MAC address shall be removed.

- Reentrant — yes
- ISR Callable — no

**Parameters**

***portId*** — IxEthDBPortId

***macAddr*** — IxEthDBMacAddr

**Returns**

- IxEthDBStatus
- IX_ETH_DB_SUCCESS
- IX_ETH_DB_FAIL — Failed to populate the database entry.
- IX_ETH_DB_INVALID_PORT — portId is invalid.
- IX_ETH_DB_PORT_UNINITIALIZED — portId is un-initialized

### A.3.14.9.10 IxEthDBStatus ixEthDBInit (void)

initializes the Ethernet learning/filtering database.

**Returns**

- IX_ETH_DB_SUCCESS — initialization was successful
- IX_ETH_DB_FAIL — initialization failed (OSSL error)

### A.3.14.9.11 ixEthDBPortAddressSet (..)

ixEthDBPortAddressSet (IxEthDBPortId portID, IxEthDBMacAddr * macAddr)

set a port MAC address.

**Parameters**

***portID*** — ID of the port whose MAC address is set

***macAddr*** — port MAC address

This function is to be called from the Ethernet Access component top-level ixEthDBUnicastAddressSet(). Event processing cannot be enabled for a port until its MAC address has been set.

**Returns**

- IX_ETH_DB_SUCCESS — MAC address was set successfully
- IX_ETH_DB_FAIL — MAC address was not set due to a message handler failure
- IX_ETH_DB_INVALID_PORT — if the port is not an Ethernet NPE

**See also:**

IxEthDBPortDefs.h for port definitions

#### A.3.14.9.12 ixEthDBPortAgingDisable (..)

ixEthDBPortAgingDisable (IxEthDBPortId port)

Disable the aging function for a specific port.

- Reentrant — yes
- ISR Callable — no

**Parameters**

***portId*** — IxEthDBPortId

**Returns**

- IxEthDBStatus
- IX_ETH_DB_SUCCESS
- IX_ETH_DB_INVALID_PORT — portId is invalid.
- IX_ETH_DB_PORT_UNINITIALIZED — portId is un-initialized.

#### A.3.14.9.13 ixEthDBPortAgingEnable (..)

ixEthDBPortAgingEnable (IxEthDBPortId port)

Enable the aging function for a specific port.

Enable the aging of dynamic source learned mac address entries in the filtering database.

**Note**

The aging function relies on the ixEthDBDatabaseMaintenance being called with a period of IX_ETH_DB_MAINTENANCE_TIME seconds.

- Reentrant — yes
- ISR Callable — no

**Parameters**

***portId*** — IxEthDBPortId

**Returns**

- IxEthDBStatus
- IX_ETH_DB_SUCCESS
- IX_ETH_DB_INVALID_PORT — portId is invalid.
- IX_ETH_DB_PORT_UNINITIALIZED — portId is un-initialized.

#### A.3.14.9.14 ixEthDBPortDisable (..)

ixEthDBPortDisable (IxEthDBPortId portID)

Disable processing on a port.

#### Parameters

***portID*** — ID of the port to disable processing on

This function is called automatically from the Ethernet Access component top-level portDisable() routine and should be manually called for any user-defined port (any port that is not one of the two Ethernet NPEs).

#### Returns

- IX_ETH_DB_SUCCESS — if disabling is successful
- IX_ETH_DB_FAIL — if the disabling was not successful due to a message handler error

### A.3.14.9.15 ixEthDBPortEnable (..)

ixEthDBPortEnable (IxEthDBPortId portID)

Enable a port.

#### Parameters

***portID*** — ID of the port to enable processing on

This function is called automatically from the Ethernet Access component top-level portEnable() routine and should be manually called for any user-defined port (any port that is not one of the two Ethernet NPEs).

#### Returns

- IX_ETH_DB_SUCCESS — if enabling is successful.
- IX_ETH_DB_FAIL — if the enabling was not successful due to a message handler error.
- IX_ETH_DB_MAC_UNINITIALIZED — the MAC address of this port was not initialized (only for Ethernet NPEs).

Precondition — ixEthDBPortAddressSet() needs to be called prior to enabling the port events for Ethernet NPEs

#### See also:

ixEthDBPortAddressSet()

IxEthDBPortDefs.h for port definitions

### A.3.14.9.16 ixEthDBPortInit (..)

ixEthDBPortInit (IxEthDBPortId portID)

initializes a port.

**Parameters**

***portID*** — ID of the port to be initialized

This function is called automatically by the Ethernet Access component top-level portInit() routine and should be manually called for any user-defined port (any port that is not one of the two Ethernet NPEs).

**See also:**

IxEthDBPortDefs.h for port definitions

## A.3.15 Ethernet Database Port Definitions (IxEthDBPortDefs)

Intel® IXP42X product line Ethernet Port Definitions for private MAC learning API.

### A.3.15.1 Data Structures

- struct IxEthDBPortDefinition
  Port Definition - a structure contains the Port type and capabilities.

### A.3.15.2 Defines

- #define IX_ETH_DB_NUMBER_OF_PORTS (3)
  Number of supported ports.
- #define COMPLETE_ETH_PORT_MAP ((1 << IX_ETH_DB_NUMBER_OF_PORTS) - 1)
  Complete set of ports in use.
- #define IX_ETH_DB_CHECK_PORT(portID)
  Safety checks to verify whether the port is invalid or uninitialized.

### A.3.15.3 Enumerations

- enum IxEthDBPortType { ETH_GENERIC = 0, ETH_NPE }
  Port types - currently only Ethernet NPEs are recognized as specific types.
- enum IxEthDBPortCapability { NO_CAPABILITIES = 0, ENTRY_AGING = 0x1 }
  Port capabilities - used by ixEthAccDatabaseMaintenance to decide whether it should manually age entries or not depending on the port capabilities.

### A.3.15.4 Detailed Description

Intel® IXP42X product line Ethernet Port Definitions for private MAC learning API.

### A.3.15.5 Define Documentation

#### A.3.15.5.1 #define COMPLETE_ETH_PORT_MAP ((1 << IX_ETH_DB_NUMBER_OF_PORTS) - 1)

complete set of ports in use.

only ports 0, 1 and 2 are in use - sets bit[n] to 1 if port[n] exists

#### A.3.15.5.2 #define IX_ETH_DB_CHECK_PORT(portID)

Value:

```
{ \
    if ((portID) < 0 || (portID) >= IX_ETH_DB_NUMBER_OF_PORTS) \
    { \
        return IX_ETH_DB_INVALID_PORT; \
    } \
    \
    if (!ixEthDBPortInfo[(portID)].enabled) \
    { \
        return IX_ETH_DB_PORT_UNINITIALIZED; \
    } \
}
```

safety checks to verify whether the port is invalid or uninitialized.

### A.3.15.6 Enumeration Type Documentation

#### A.3.15.6.1 enum IxEthDBPortCapability

Port capabilities — used by ixEthAccDatabaseMaintenance to decide whether it should manually age entries or not depending on the port capabilities.

Ethernet NPEs have aging capabilities, meaning that they will age the entries automatically (by themselves).

## A.3.16 Ethernet NPE (IxEthNpe) API

Contains the API for Ethernet NPE.

### A.3.16.1 Defines

- #define IX_ETHNPE_X2P_NPE_HALT 0x00

  Request from the Intel XScale core client for the NPE to immediately halt all execution and flush any mbufs in its possession.

- #define IX_ETHNPE_X2P_ELT_SETPORTADDRESS 0x10

  Indication from the Intel XScale core client that the attached Ethernet port's MAC address is equal to the specified value and that the port ID of attached Ethernet port should be set to the specified value.

- #define IX_ETHNPE_X2P_ELT_ACCESSREQUEST 0x11

  Request from the Intel XScale core client for the NPE to relinquish control of the Ethernet Learning Tree and write it back to external memory (at the location specified in the last X2P_ELT_AccessRelease message).

- #define IX_ETHNPE_X2P_ELT_ACCESSRELEASE 0x12

  Indication from the Intel XScale core client that it has relinquished control of the Ethernet Learning Tree and has written an updated version of it, with its base node at the specified address (the base node is the empty node immediately preceding the true root node).

- #define IX_ETHNPE_X2P_ELT_INSERTADDRESS 0x13

  Indication from the Intel XScale core client that the NPE should insert the specified MAC address/Port ID into internal tree.

- #define IX_ETHNPE_X2P_FP_SETETHERNETTYPE 0x20

  Indication from the Intel XScale core client that the universal FastPath Ethernet type should be set to the specified value.

- #define IX_ETHNPE_X2P_FP_WRITETEMPMDF 0x21

  Indication from the Intel XScale core client that the specified modification template should be loaded into the NPE modification template array at the specified index.

- #define IX_ETHNPE_X2P_STATS_SHOW 0x30

  Request from the Intel XScale core client for the current MAC port statistics data to be written to the (empty) statistics structure and the specified location in external memory.

- #define IX_ETHNPE_X2P_STATS_RESET 0x31

  Request from the Intel XScale core client for the NPE to reset all of its internal MAC port statistics state variables.

- #define IX_ETHNPE_P2X_NPE_STATUS 0x00

  Indication from the NPE of its current status.

- #define IX_ETHNPE_P2X_ELT_ACKPORTADDRESS 0x10

  Indication from the NPE that is has finished processing the previous X2P_ELT_SetPortAddress message.

- #define IX_ETHNPE_P2X_ELT_ACCESSGRANT 0x11

  Indication from the NPE that it relinquished control of the Ethernet Learning Tree and has written it back to external memory at the specified base address.

- #define IX_ETHNPE_P2X_ELT_BALANCEREQUEST 0x12

  Request from the NPE for the Intel XScale core client to insert the specified MAC address into the Ethernet Learning Tree and rebalance it (the NPE has run out of depth while attempting to insert the source MAC address itself).

- #define IX_ETHNPE_P2X_ELT_NEWADDRESS 0x13

  Indication from the NPE that it has just learned (i.e. inserted into its internal tree) the specified new MAC address.

- #define IX_ETHNPE_P2X_ELT_INSERTADDRESSACK 0x14

  Indication from the NPE that it has successfully enqueued (to the learning process) the MAC address from the previous X2P_ELT_Insert_Address message.

- #define IX_ETHNPE_P2X_ELT_INSERTADDRESSNACK 0x15

  Indication from the NPE that it is unable to enqueue (to the learning process) the MAC address from the previous X2P_ELT_Insert_Address message.

- #define IX_ETHNPE_P2X_FP_WRITETEMPMDFACK 0x20
  Indication from the NPE that it is finished copying the FastPath modification template for the specified VC at the specified address.
- #define IX_ETHNPE_P2X_FP_SHUTDOWNVCACK 0x21
  Indication from the NPE that it has processed a special VC shutdown description for the VC with the specified FastPath index.
- #define IX_ETHNPE_P2X_STATS_REPORT 0x30
  Indication from the NPE that the current MAC port statistics are available in the specified buffer.
- #define IX_ETHNPE_P2X_STATS_CLEAR_REPORT 0x31
  Indication from the NPE that the current MAC port statistics are cleared.
- #define MASK(hi, lo) ((1 << ((hi) + 1)) - (1 << (lo)))
  Macro for mask.
- #define BITS(x, hi, lo) (((x) & MASK(hi,lo)) >> (lo))
  Macro for bits.
- #define IX_ETHNPE_QM_Q_FIELD_NPEID_L 31
  QMgr Queue NPE ID field left boundary.
- #define IX_ETHNPE_QM_Q_FIELD_NPEID_R 31
  QMgr Queue NPE ID field right boundary.
- #define IX_ETHNPE_QM_Q_FIELD_FPBIT_L 31
  QMgr Queue Fast Path bit field left boundary.
- #define IX_ETHNPE_QM_Q_FIELD_FPBIT_R 31
  QMgr Queue Fast Path bit field right boundary.
- #define IX_ETHNPE_QM_Q_FIELD_PORTID_L 30
  QMgr Queue Port ID field left boundary.
- #define IX_ETHNPE_QM_Q_FIELD_PORTID_R 28
  QMgr Queue Port ID field right boundary.
- #define IX_ETHNPE_QM_Q_FIELD_PRIOR_L 30
  QMgr Queue Priority field left boundary.
- #define IX_ETHNPE_QM_Q_FIELD_PRIOR_R 28
  QMgr Queue Priority field right boundary.
- #define IX_ETHNPE_QM_Q_FIELD_ADDR_L 27
  QMgr Queue Address field left boundary.
- #define IX_ETHNPE_QM_Q_FIELD_ADDR_R 2
  QMgr Queue Address field right boundary.
- #define IX_ETHNPE_QM_Q_FIELD_FPINDEX_L 15
  QMgr Queue Fast Path index field left boundary.
- #define IX_ETHNPE_QM_Q_FIELD_FPINDEX_R 1
  QMgr Queue Fast Path index field right boundary.

- #define IX_ETHNPE_QM_Q_FIELD_FPSHTDN_R 0
  QMgr Queue Fast Path shutdown field right boundary.
- #define IX_ETHNPE_QM_Q_FIELD_FPSHTDN_L 0
- #define IX_ETHNPE_QM_Q_FREEENET_ADDR_MASK
  Macro to mask the Address field of the FreeEnet Queue Manager Entry.
- #define IX_ETHNPE_QM_Q_RXENET_NPEID_MASK
  Macro to mask the NPE ID field of the RxEnet Queue Manager Entry.
- #define IX_ETHNPE_QM_Q_RXENET_PORTID_MASK
  Macro to mask the Port ID field of the Queue Manager RxEnet Queue entry.
- #define IX_ETHNPE_QM_Q_RXENET_ADDR_MASK
  Macro to mask the Mbuf Address field of the RxEnet Queue Manager Entry.
- #define IX_ETHNPE_QM_Q_TXENET_FPBIT_MASK
  Macro to mask the FastPath Flag field of the TxEnet Queue Manager Entry.
- #define IX_ETHNPE_QM_Q_TXENET_PRIOR_MASK
  Macro to mask the Priority field of the TxEnet Queue Manager Entry.
- #define IX_ETHNPE_QM_Q_TXENET_ADDR_MASK
  Macro to mask the FP Descriptor Address field of the TxEnet Queue Manager Entry.
- #define IX_ETHNPE_QM_Q_TXENET_FPINDEX_MASK
  Macro to mask the FastPath Index field of the TxEnet Shutdown Indicator Queue Manager Entry.
- #define IX_ETHNPE_QM_Q_TXENET_FPSHTDN_MASK
  Macro to mask the FastPath VC Shutdown indicator field of the TxEnet Shutdown Indicator Queue Manager Entry.
- #define IX_ETHNPE_QM_Q_TXENETDONE_NPEID_MASK
  Macro to mask the NPE ID field of the TxEnetDone Queue Manager Entry.
- #define IX_ETHNPE_QM_Q_TXENETDONE_ADDR_MASK
  Macro to mask the Mbuf Address field of the TxEnetDone Queue Manager Entry.
- #define IX_ETHNPE_QM_Q_FREEFP_ADDR_MASK
  Macro to mask the FP Descriptor Address field of the Fast Path FreeFP Queue Manager Entry.
- #define IX_ETHNPE_QM_Q_FREEENET_ADDR_VAL(x) ((x) & IX_ETHNPE_QM_Q_FREEENET_ADDR_MASK)
  Extraction macro for Address field of FreeNet Queue Manager Entry.
- #define IX_ETHNPE_QM_Q_RXENET_NPEID_VAL(x)
  Extraction macro for NPE ID field of RxEnet Queue Manager Entry.
- #define IX_ETHNPE_QM_Q_RXENET_PORTID_VAL(x)
  Extraction macro for Port ID field of RxEnet Queue Manager Entry.
- #define IX_ETHNPE_QM_Q_RXENET_ADDR_VAL(x) ((x) & IX_ETHNPE_QM_Q_RXENET_ADDR_MASK)
  Extraction macro for Address field of RxEnet Queue Manager Entry.

- #define IX_ETHNPE_QM_Q_TXENET_FPBIT_VAL(x)
- #define IX_ETHNPE_QM_Q_TXENET_PRIOR_VAL(x)

  Extraction macro for Priority field of TxEnet Queue Manager Entry.
- #define IX_ETHNPE_QM_Q_TXENET_ADDR_VAL(x) ((x) & IX_ETHNPE_QM_Q_TXENET_ADDR_MASK)

  Extraction macro for Address field of Queue Manager SlowPath TxEnet Queue Manager Entry.
- #define IX_ETHNPE_QM_Q_TXENET_FPINDEX_VAL(x)

  Extraction macro for Fast Path Index field of FastPath TxEnet Queue Manager Entry.
- #define IX_ETHNPE_QM_Q_TXENET_FPSHTDN_VAL(x)

  Extraction macro for FastPath Shutdown field of TxEnet/RxFP VC Shutdown Indicator Queue Manager Entry.
- #define IX_ETHNPE_QM_Q_TXENETDONE_NPEID_VAL(x)

  Extraction macro for NPE ID field of TxEnetDone Queue Manager Entry.
- #define IX_ETHNPE_QM_Q_TXENETDONE_ADDR_VAL(x) ((x) & IX_ETHNPE_QM_Q_TXENETDONE_ADDR_MASK)

  Extraction macro for Address field of TxEnetDone Queue Manager Entry.
- #define IX_ETHNPE_QM_Q_FREEFP_ADDR_VAL(x) ((x) & IX_ETHNPE_QM_Q_FREEFP_ADDR_MASK)

  Extraction macro for Address field of FastPath FreeFP Queue Manager Entry.
- #define IX_ETHNPE_QM_Q_FREEENET_ENTRY(addr) (addr)

  Queue entry construction macros for FreeNet Queue Manager.
- #define IX_ETHNPE_QM_Q_RXENET_ENTRY(addr, id, prt)

  Queue entry construction macros for RxEnet Queue Manager.
- #define IX_ETHNPE_QM_Q_TXENET_ENTRY(addr, fp, pri)

  Queue entry construction macros for TxEnet Queue Manager.
- #define IX_ETHNPE_QM_Q_TXENETDONE_ENTRY(addr, id) ((id) << 30 | (addr))

  Queue entry construction macros for TxEnetDone Queue Manager.
- #define IX_ETHNPE_QM_Q_VCSHUTDOWN_ENTRY(fpidx) ((fpidx) << 1 | 0x80000001)

  Queue entry construction macros for VC Shutdown Queue Manager.
- #define IX_ETHNPE_QM_Q_FREEFP_ENTRY(addr) (addr)

  Queue entry construction macros for FreeFP Queue Manager.

### A.3.16.2 Detailed Description

Contains the API for Ethernet NPE.

## A.3.16.3 Define Documentation

### A.3.16.3.1 #define IX_ETHNPE_P2X_ELT_BALANCEREQUEST 0x12

Request from the NPE for the Intel XScale core client to insert the specified MAC address into the Ethernet Learning Tree and rebalance it (the NPE has run out of depth while attempting to insert the source MAC address itself).

A P2X_ELT_AccessGrant message is implied (i.e. the NPE will have relinquished control of the tree and written it back to external memory prior to issuing this message).

### A.3.16.3.2 #define IX_ETHNPE_QM_Q_FREEENET_ADDR_MASK

Value:

```
MASK (IX_ETHNPE_QM_Q_FIELD_ADDR_L, \
                  IX_ETHNPE_QM_Q_FIELD_ADDR_R)
```

Macro to mask the Address field of the FreeEnet Queue Manager Entry.

### A.3.16.3.3 #define IX_ETHNPE_QM_Q_FREEENET_ADDR_VAL(x) ((x) & IX_ETHNPE_QM_Q_FREEENET_ADDR_MASK)

Extraction macro for Address field of FreeNet Queue Manager Entry.

Pointer to an mbuf buffer descriptor

### A.3.16.3.4 #define IX_ETHNPE_QM_Q_FREEFP_ADDR_MASK

Value:

```
MASK (IX_ETHNPE_QM_Q_FIELD_ADDR_L, \
                  IX_ETHNPE_QM_Q_FIELD_ADDR_R)
```

Macro to mask the FP Descriptor Address field of the Fast Path FreeFP Queue Manager Entry.

### A.3.16.3.5 #define IX_ETHNPE_QM_Q_FREEFP_ADDR_VAL(x) ((x) & IX_ETHNPE_QM_Q_FREEFP_ADDR_MASK)

Extraction macro for Address field of FastPath FreeFP Queue Manager Entry.

Pointer to a FastPath descriptor

### A.3.16.3.6 #define IX_ETHNPE_QM_Q_RXENET_ADDR_MASK

Value:

```
MASK (IX_ETHNPE_QM_Q_FIELD_ADDR_L, \
                  IX_ETHNPE_QM_Q_FIELD_ADDR_R)
```

Macro to mask the Mbuf Address field of the RxEnet Queue Manager Entry.

#### A.3.16.3.7 #define IX_ETHNPE_QM_Q_RXENET_ADDR_VAL(x) ((x) & IX_ETHNPE_QM_Q_RXENET_ADDR_MASK)

Extraction macro for Address field of RxEnet Queue Manager Entry.

Pointer to an mbuf buffer descriptor

#### A.3.16.3.8 #define IX_ETHNPE_QM_Q_RXENET_ENTRY(addr, id, prt)

Value:

```
((id)  << 30 | \
                                                    (prt) << 28 | (addr))
```

Queue entry construction macros for RxEnet Queue Manager.

#### A.3.16.3.9 #define IX_ETHNPE_QM_Q_RXENET_NPEID_MASK

Value:

```
MASK (IX_ETHNPE_QM_Q_FIELD_NPEID_L, \
                  IX_ETHNPE_QM_Q_FIELD_NPEID_R)
```

Macro to mask the NPE ID field of the RxEnet Queue Manager Entry.

#### A.3.16.3.10 #define IX_ETHNPE_QM_Q_RXENET_NPEID_VAL(x)

Value:

```
BITS (x, IX_ETHNPE_QM_Q_FIELD_NPEID_L, \
                     IX_ETHNPE_QM_Q_FIELD_NPEID_R)
```

Extraction macro for NPE ID field of RxEnet Queue Manager Entry.

Set to 0 for entries originating from the Eth0 NPE; Set to 1 for entries originating from the Eth1 NPE.

#### A.3.16.3.11 #define IX_ETHNPE_QM_Q_RXENET_PORTID_MASK

Value:

```
MASK (IX_ETHNPE_QM_Q_FIELD_PORTID_L, \
                  IX_ETHNPE_QM_Q_FIELD_PORTID_R)
```

Macro to mask the Port ID field of the Queue Manager RxEnet Queue entry.

#### A.3.16.3.12 #define IX_ETHNPE_QM_Q_RXENET_PORTID_VAL(x)

Value:

```
BITS (x, IX_ETHNPE_QM_Q_FIELD_PORTID_L, \
                     IX_ETHNPE_QM_Q_Field_PortID_R)
```

Extraction macro for Port ID field of RxEnet Queue Manager Entry.

- 0-5 — Assignable (by the Intel XScale core client) to any of the physical ports.
- 6 — Reserved.
- 7 — Indication that the NPE did not find the associated frame's destination MAC address within its internal filtering database.

#### A.3.16.3.13 #define IX_ETHNPE_QM_Q_TXENET_ADDR_MASK

Value:

```
MASK (IX_ETHNPE_QM_Q_FIELD_ADDR_L, \
                  IX_ETHNPE_QM_Q_FIELD_ADDR_R)
```

Macro to mask the FP Descriptor Address field of the TxEnet Queue Manager Entry.

#### A.3.16.3.14 #define IX_ETHNPE_QM_Q_TXENET_ADDR_VAL(x) ((x) & IX_ETHNPE_QM_Q_TXENET_ADDR_MASK)

Extraction macro for Address field of Queue Manager SlowPath TxEnet Queue Manager Entry.

Pointer to an mbuf buffer descriptor

#### A.3.16.3.15 #define IX_ETHNPE_QM_Q_TXENET_ENTRY(addr, fp, pri)

Value:

```
((fp)  << 30 | \
                                                    (pri) << 28 | (addr))
```

Queue entry construction macros for TxEnet Queue Manager.

#### A.3.16.3.16 #define IX_ETHNPE_QM_Q_TXENET_FPBIT_MASK

Value:

```
MASK (IX_ETHNPE_QM_Q_FIELD_FPBIT_L, \
                  IX_ETHNPE_QM_Q_FIELD_FPBIT_R)
```

Macro to mask the FastPath Flag field of the TxEnet Queue Manager Entry.

#### A.3.16.3.17 #define IX_ETHNPE_QM_Q_TXENET_FPBIT_VAL(x)

Value:

```
BITS (x, IX_ETHNPE_QM_Q_FIELD_FPBIT_L, \
                     IX_ETHNPE_QM_Q_FIELD_FPBIT_R)
```

#### A.3.16.3.18 #define IX_ETHNPE_QM_Q_TXENET_FPINDEX_MASK

Value:

```
MASK (IX_ETHNPE_QM_Q_FIELD_FPINDEX_L, \
                  IX_ETHNPE_QM_Q_FIELD_FPINDEX_R)
```

Macro to mask the FastPath Index field of the TxEnet Shutdown Indicator Queue Manager Entry.

#### A.3.16.3.19 #define IX_ETHNPE_QM_Q_TXENET_FPINDEX_VAL(x)

Value:

```
BITS (x, IX_ETHNPE_QM_Q_FIELD_FPINDEX_L, \
                    IX_ETHNPE_QM_Q_FIELD_FPINDEX_R)
```

Extraction macro for Fast Path Index field of FastPath TxEnet Queue Manager Entry.

Pointer to a FastPath descriptor

#### A.3.16.3.20 #define IX_ETHNPE_QM_Q_TXENET_FPSHTDN_MASK

Value:

```
MASK (IX_ETHNPE_QM_Q_FIELD_FPSHTDN_L, \
                  IX_ETHNPE_QM_Q_FIELD_FPSHTDN_R)
```

Macro to mask the FastPath VC Shutdown indicator field of the TxEnet Shutdown Indicator Queue Manager Entry.

#### A.3.16.3.21 #define IX_ETHNPE_QM_Q_TXENET_FPSHTDN_VAL(x)

Value:

```
BITS (x, IX_ETHNPE_QM_Q_FIELD_FPSHTDN_L, \
                    IX_ETHNPE_QM_Q_FIELD_FPSHTDN_R)
```

Extraction macro for FastPath Shutdown field of TxEnet/RxFP VC Shutdown Indicator Queue Manager Entry.

#### A.3.16.3.22 #define IX_ETHNPE_QM_Q_TXENET_PRIOR_MASK

Value:

```
MASK (IX_ETHNPE_QM_Q_FIELD_PRIOR_L, \
                  IX_ETHNPE_QM_Q_FIELD_PRIOR_R)
```

Macro to mask the Priority field of the TxEnet Queue Manager Entry.

#### A.3.16.3.23 #define IX_ETHNPE_QM_Q_TXENET_PRIOR_VAL(x)

Value:

```
BITS (x, IX_ETHNPE_QM_Q_FIELD_PRIOR_L, \
                    IX_ETHNPE_QM_Q_FIELD_PRIOR_R)
```

Extraction macro for Priority field of TxEnet Queue Manager Entry.

Priority of the packet (as described in IEEE 802.1D) must be 0 for FastPath VC shutdown indicators. This field is cleared upon return from the Ethernet NPE (to either the TxEnetDone or FreeFP queue).

#### A.3.16.3.24 #define IX_ETHNPE_QM_Q_TXENETDONE_ADDR_MASK

Value:

```
MASK (IX_ETHNPE_QM_Q_FIELD_ADDR_L, \
                  IX_ETHNPE_QM_Q_FIELD_ADDR_R)
```

Macro to mask the Mbuf Address field of the TxEnetDone Queue Manager Entry.

#### A.3.16.3.25 #define IX_ETHNPE_QM_Q_TXENETDONE_ADDR_VAL(x) ((x) & IX_ETHNPE_QM_Q_TXENETDONE_ADDR_MASK)

Extraction macro for Address field of TxEnetDone Queue Manager Entry.

Pointer to an mbuf buffer descriptor

#### A.3.16.3.26 #define IX_ETHNPE_QM_Q_TXENETDONE_NPEID_MASK

Value:

```
MASK (IX_ETHNPE_QM_Q_FIELD_NPEID_L, \
                  IX_ETHNPE_QM_Q_FIELD_NPEID_R)
```

Macro to mask the NPE ID field of the TxEnetDone Queue Manager Entry.

#### A.3.16.3.27 #define IX_ETHNPE_QM_Q_TXENETDONE_NPEID_VAL(x)

Value:

```
BITS (x, IX_ETHNPE_QM_Q_FIELD_NPEID_L, \
                     IX_ETHNPE_QM_Q_FIELD_NPEID_R)
```

Extraction macro for NPE ID field of TxEnetDone Queue Manager Entry.

Set to 0 for entries originating from the Eth0 NPE; set to 1 for entries originating from the Eth1 NPE.

#### A.3.16.3.28 #define IX_ETHNPE_X2P_ELT_ACCESSRELEASE 0x12

Indication from the Intel XScale core client that it has relinquished control of the Ethernet Learning Tree and has written an updated version of it, with its base node at the specified address (the base node is the empty node immediately preceding the true root node).

The tree will remain at the same location until the next X2P_ELT_AccessRelease message.

#### A.3.16.3.29 #define IX_ETHNPE_X2P_FP_SETETHERNETTYPE 0x20

Indication from the Intel XScale core client that the universal FastPath Ethernet type should be set to the specified value.

This message is useful only in a “single stack” (IPv4 or IPv6) application. The Ethernet type will be template-specific in future Ethernet NPE firmware releases, in order to support “mixed stack” (IPv4 and IPv6) applications.

#### A.3.16.3.30 #define IX_ETHNPE_X2P_NPE_HALT 0x00

Request from the Intel XScale core client for the NPE to immediately halt all execution and flush any mbufs in its possession.

Any free mbuf held by the NPE receive process is flushed to the RxEnet queue. Transmit path mbufs and FastPath descriptors (those in the Priority Queue and any one currently in the process of transmission) are immediately flushed to either the TxEnetDone queue or the FreeFP queue, depending on their point of origin. When an X2P_NPE_Halt message is issued, there must be no pending FastPath VC shutdowns. Following a halt operation, a NPE may be brought into the "ready" state only by downloading the firmware again.

#### A.3.16.3.31 #define IX_ETHNPE_X2P_STATS_RESET 0x31

Request from the Intel XScale core client for the NPE to reset all of its internal MAC port statistics state variables.

As a side effect, this message entails an implicit request that the NPE write the current MAC port statistics into the MAC statistics structure at the specified location in external memory.

## A.3.17 Fast Path Access (IxFpathAcc) API

The public API for the Intel® IXP42X product line IxFpathAcc Component.

### A.3.17.1 Defines

- #define IX_FPATH_POOL_FULL 2
  Buffer pool is full.
- #define IX_FPATHACC_CLS_OPCODE_END 0x00
  Classifier Opcode Definitions:
  - #define IX_FPATHACC_CLS_OPCODE_UPPER_LIMIT 0x01
  - #define IX_FPATHACC_CLS_OPCODE_LOWER_LIMIT 0x02
  - #define IX_FPATHACC_CLS_OPCODE_MATCH_BYTES 0x03
  - #define IX_FPATHACC_CLS_OPCODE_CHECKSUM 0x04
  - #define IX_FPATHACC_CLS_OPCODE_1_OUT_OF_4_MATCH 0x05
  - #define IX_FPATHACC_CLS_OPCODE_A_OR_B_MATCH 0x06
  - #define IX_FPATHACC_CLS_OPCODE_WAN_MAC_LEARN 0x07
  - #define IX_FPATHACC_CLS_OPCODE_CONDITIONAL_BRANCH 0x08
  - #define IX_FPATHACC_CLS_OPCODE_COPY16 0x09
  - #define IX_FPATHACC_CLS_OPCODE_NOP_PAD2 0x0A
  - #define IX_FPATHACC_CLS_OPCODE_NOP_PAD1 0x0B
  - #define IX_FPATHACC_CLS_OPCODE_NOP 0x0C
  - #define IX_FPATHACC_CLS_DUMMY_BYTE 0xFF

### A.3.17.2 Typedefs

- typedef unsigned int IxFpathAccClientId

  Client Id type.

- typedef void(* IxFpathAccNewWanMacAddrCallback )(IxFpathAccClientId clientId, unsigned char *macAddress)

  New WAN MAC address notification.

- typedef void(* IxFpathAccBufPoolUnderRunCallback )(IxFpathAccClientId clientId)

  The fast path buffer pool has suffered an under-run event.

### A.3.17.3 Enumerations

- enum IxFpathEthernetPort { IX_FPATHACC_ETHERNET_PORT0 = 0, IX_FPATHACC_ETHERNET_MAX_PORTS }

  The Ethernet port identifiers.

### A.3.17.4 Functions

- PUBLIC IX_STATUS ixFpathAccInit (void)

  Initialize the Fpath Access component.

- PUBLIC IX_STATUS ixFpathAccEtherTypeSet (IxFpathEthernetPort port, UINT16 etherType)

  Set the Ethernet type for frames on the specified port.

- PUBLIC IX_STATUS ixFpathAccBufferSupply (IX_MBUF *mBuf, int *unpopulatedEntryCnt)

  Populate the Fast Path buffer pool.

- PUBLIC IX_STATUS ixFpathAccBufferPoolInfoGet (int *maxEntries, int *populatedEntryCnt)

  Get the fast path buffer pool info.

- PUBLIC IX_STATUS ixFpathAccBufferPoolUnderRunCallbackSet (IxFpathAccBufPoolUnderRunCallback callback, IxFpathAccClientId clientId)

  Set the buffer pool under-run notification callback.

- PUBLIC IX_STATUS ixFpathAccTemplatePairSet (IxAtmNpeRxVcId npeVcId, unsigned char *classifierTemplate, int classifierTemplateLength, unsigned char *modifierTemplate, int modifierTemplateLength)

  Set the classifier/modifier template for a specific VC.

- PUBLIC IX_STATUS ixFpathAccTemplatePairClear (IxAtmNpeRxVcId npeVcId)

  Disable a template pair for a VC.

- PUBLIC IX_STATUS ixFpathAccWanMacAddrCallbackSet (IxFpathAccNewWanMacAddrCallback macAddrCallback, IxAtmNpeRxVcId npeVcId, IxFpathAccClientId clientId)

  Set the Mac Address callback for a VC.

- PUBLIC IX_STATUS ixFpathAccStatisticsGet (IxAtmNpeRxVcId npeVcId, int *packetCnt)

  Retrieve the fast path statistics for a VC.

- PUBLIC IX_STATUS ixFpathAccShow (void)
  Display some component information for debug purposes.

## A.3.17.5 Detailed Description

The public API for the Intel® IXP42X product line IxFpathAcc Component.

## A.3.17.6 Define Documentation

### A.3.17.6.1 #define IX_FPATH_POOL_FULL 2

Buffer pool is full.

This constant is a return value used to tell the user that the IxFpathAcc buffer pool is full.

### A.3.17.6.2 #define IX_FPATHACC_CLS_OPCODE_END 0x00

Classifier Opcode Definitions — Opcodes used in classifier templates - NPE-A interprets a sequences of these opcodes known as a template.

**Classifier Instruction Format**

1 byte 1 bytes 1 byte Np bytes depending on Opcode, Opcode Offset OperandCount (Nd) Parameters

Opcode — Classifier operation Offset — Position of the first byte to check, unsigned char, 16 is the first byte of the current cell; positions 0-15 indicate bytes in the previous cell and are therefore only valid after a Save16Bytes operation. OperandCount — Number of bytes of PDU data needed to perform the checking.

Parameters — Some opcodes require opcode specific parameters. The number of which depends on the opcode and the OperandCount.

Op name(value) Nd Np Definition of the operation and parameter* End(0) N/A 0 value No more checking is needed UpperLimit(1) 2,4 Nd Nd byte data as an unsigned integer must be <= Nd byte of Pt as an unsigned integer. LowerLimit(2) N/A 1 One byte data as an unsigned integer must be >= One byte of Pt as an unsigned integer. Match(3) 1,2,3, Nd Nd byte data must be == Nd byte of Pt. 4,8,10 Checksum(4) 10 0 Nd many short (i.e. 2-byte) data 16-bit checksum must be == 0xFFFF. 1-out-of-4-match(5) 2,4,12 Nd byte data must be == one of the 4 Nd -byte 16,48 data in Pt. A-or-B match(6) N/A 2 One byte data must be either one of the two bytes in Pt. MacWanLearn(7) N/A 0 Inspect and report WAN MAC src addresses. Conditional (8) Special 1 If one byte match, skip next SpecialNd many bytes Branch Nd of TempCls. Else go to the next test. Save16Bytes(9) N/A 0 D0 must be 48 for this instruction to work as intended. This instruction copies the last 16 bytes of the current cell's payload to a VC-specific temporary buffer and pauses the execution of the classifier until the next AAL5 cell is received on this VC. When the next cell is received, the previous cell's data is automatically prepended to the current cell and the classifier is restarted on the operation following Save16Bytes in the template. This instruction allows the classifier to cross cell boundaries subject to the restriction of a 16-byte overlap. NOPpad3(10) N/A 0 This instruction is used only to align the Pt field of a subsequent 1-out-of-4-match instruction. When the classifier encounters this instruction it will advance the TempCls pointer by three, i.e. past this NOPpad3 instruction and the following two bytes. NOPpad2(11) N/A 0 This instruction is used only to align the Pt field of a subsequent 1-out-of-4-match instruction. When the classifier encounters this instruction it will advance the TempCls pointer by two, i.e. past this NOPpad2 instruction and the following byte. NOPpad1(12) N/A 0 This instruction is used only to align the Pt field of a

subsequent 1-out-of-4-match instruction. When the classifier encounters this instruction it will advance the Classifier pointer by one, i.e. past this NOPpad1 instruction. reserved(13) N/A N/A THIS IS AN ILLEGAL OPCODE. It is used internally by the PSM to efficiently implement the NOP instructions above.

## A.3.17.7 Typedef Documentation

### A.3.17.7.1 typedef void(..)

typedef void(* IxFpathAccBufPoolUnderRunCallback)(IxFpathAccClientId clientId)

The fast path buffer pool has suffered an under-run event.

This function is called to notify a client that the fast path NPEs have requested more buffers from the pool than were placed in the pool at initialization. The possible result is that packets are sent on the slow path instead of the fast path.

This function will called exactly once every time the pool goes from empty to under-run.

#### Parameters

***clientId*** — ID provided at registration of this callback.

#### See also:

ixFpathAccBufferPoolUnderRunCallbackSet, ixFpathAccBufferSupply

### A.3.17.7.2 typedef void(..)

typedef void(* IxFpathAccNewWanMacAddrCallback)( IxFpathAccClientId clientId, unsigned char *macAddress)

New WAN MAC address notification.

This function is called to notify a client that the fast path has identified a change in src MAC address of the incoming packet stream on a VC.

This function is only called if WanMacLearning instruction is enabled in the classifier template set for the VC.

This function will be called every time a new WAN MAC address is learned, and is also guaranteed to be called every 500ms when the WAN MAC address is constant.

#### Note

If the WAN MAC learning instruction is the first instruction in a classifier template all packets on the VC will be examined for MAC address changes. If it is the last instruction then onlt packets qualifying for the fast path will be examined.

#### Parameters

***clientId*** — ID provided at registration of this callback.

***macAddress*** — A pointer to the latest observed Mac Address for the VC.

## A.3.17.8 Enumeration Type Documentation

### A.3.17.8.1 enum IxFpathEthernetPort

The Ethernet port identifiers.

#### Enumeration values

IX_FPATHACC_ETHERNET_PORT0 — Ethernet port 0 - attached to NPE-B >.

## A.3.17.9 Function Documentation

### A.3.17.9.1 ixFpathAccBufferPoolInfoGet (..)

ixFpathAccBufferPoolInfoGet (int * maxEntries, int * populatedEntryCnt)

Get the fast path buffer pool info.

This function gets the value for the maximum number of buffers which the fast path buffer pool can accommodate, it also retrieves the number of fast path buffers which have been supplied to the pool so far.

This function will fail if a template is set for any VC.

#### Parameters

***maxEntries*** — points to a variable which will be filled with the maximum number of buffers which the fast path free pool can contain.

***populatedEntryCnt*** — points to a variable which will filled a value representing number of fast path buffers currently supplied to the fast path pool.

#### See also:

ixFpathAccBufferPoolUnderRunCallbackSet, ixFpathAccBufferSupply

#### Returns

- IX_SUCCESS — successful add a buffer chain to the pool.
- IX_FAIL — Invalid parameter, a template is set for a VC or internal error.

### A.3.17.9.2 ixFpathAccBufferPoolUnderRunCallbackSet (..)

ixFpathAccBufferPoolUnderRunCallbackSet (IxFpathAccBufPoolUnderRunCallback callback, IxFpathAccClientId clientId)

Set the buffer pool under-run notification callback.

Set the buffer under-run notification function. If one has already been set then it is supersede. Hence only one notification callback is supported.

#### Note

The callback will be executed in the context of a Qmgr dispatch notification. The under-run callback can only be set when there is no active template

### See also:

IxFpathAccBufPoolUnderRunCallback

### Parameters

***callback*** — function pointer to be stored. This can be NULL.

***clientId*** — Id supplied as a parameter to the callback function when invoked. this value is ignored if the callback is NULL.

### Returns

- IX_SUCCESS — Notification callback has been set.
- IX_FAIL — IxFpathAcc not yet initialized or some internal error has occurred.

## A.3.17.9.3 ixFpathAccBufferSupply (..)

ixFpathAccBufferSupply (IX_MBUF * mBuf, int * unpopulatedEntryCnt)

Populate the Fast Path buffer pool.

This function supplies a single mbuf or chain of mbufs to the fast path service. An mbuf chain is broken into single mbufs and placed in the pool.

All mbufs must be the same size and will be used to transport packets between the NPEs. The buffer pool is common so any buffer can be used for any fast path Vc.

This function should be called before any templates are supplied to the service otherwise packets will be sent on the slow path. Usually this pool should be filled completely.

If the number of elements in a chain of buffers supplied to this call exceeds the capacity of the pool the call will fail.

### Note

Ensure that the buffer size is sufficient for the largest packet to be sent on the fast path for any VC. Otherwise the packet will be sent on the slow path.

### Parameters

***mbuf*** — single buffer or chain of buffers to be added to the pool. This parameter can be null. The first buffer supplied will set the acceptable size for all subsequent buffers.

***unpopulatedEntryCnt*** — points to a variable which will filled a value representing the space in entries remaining unfilled in the pool.

### See also:

ixFpathAccBufferPoolUnderRunCallbackSet, ixFpathAccBufferPoolInfoGet

### Returns

- IX_SUCCESS - successful add a buffer chain to the pool.
- IX_FPATH_POOL_FULL - the pool has reach its capacity.
- IX_FAIL - Invalid parameter or internal error.

### A.3.17.9.4 ixFpathAccEtherTypeSet (..)

ixFpathAccEtherTypeSet (IxFpathEthernetPort port, UINT16 etherType)

Set the Ethernet type for frames on the specified port.

This function sets the Ethernet type value to be used in fastpath frames on the specified Ethernet port where the modifier engine must build Ethernet frame header from scratch.

Hence, this function must be called before any fastpath templates are supplied to the service for VCs which build Ethernet frame headers, however IxFpathAcc will not prevent the Ether Type being set at any time, even while templates are set.

Furthermore the current fast path implementation only supports forwarding of IPv4 packets which requires a value of 0x0800 for the Ethernet Type. Other values can be supplied but the modifier engine operation only supports IPv4 right now.

#### See also:

ixFpathAccTemplatePairSet

#### Parameters

***port*** — which Ethernet port to configure.

***EtherType*** — the Ethernet type.

#### Returns

- IX_SUCCESS — successful configured the ether type for the specified port.
- IX_FAIL — Invalid port or the NPE failed to accept the request.

### A.3.17.9.5 ixFpathAccInit (void)

Initialize the Fpath Access component.

This function will initialize the Fpath Access component internals and must be called before any other API call on this component can be made.

#### Note

Do not call this function if IX_ETH_ACC_FPATH_AWARE IS NOT SET see IxEthAcc.h.

#### Returns

- IX_SUCCESS — successfully initialized the component
- IX_FAIL — Initialization failed for some unspecified internal reason.

### A.3.17.9.6 ixFpathAccShow (void)

Display some component information for debug purposes.

Show some internal operation information relating to the fast path service.

At a minimum the following will be show.

- the active fast path VCs

- the number of times a template has been set for particular VC.
- the last 5 MAC addresses for a particular VC
- a count of the number of mac address callbacks invoked for a VC.
- the number of under-run events which has occurred for the fast path buffer pool.
- the current level of the buffer pool.
- the number of fast path buffers returned via IxAtmdAcc.
- the physical address of the templates currently active.

#### Parameters

None

#### Returns

None

### A.3.17.9.7 ixFpathAccStatisticsGet (..)

ixFpathAccStatisticsGet (IxAtmNpeRxVcId npeVcId, int * packetCnt)

Retrieve the fast path statistics for a VC.

The fast path service keeps a count of all packets which are transferred on the fast path.

This value is cumulative for the same VC id. If the npeVcId changes the statistics count is cleared.

#### Parameters

***npeVcId*** — VC identifier as provided to ixFpathAccTemplatePairSet.

***packetCnt*** — current packet count for the specified VC.

#### Returns

- IX_SUCCESS — successfully retrieved the stats count.
- IX_FAIL — invalid parameter or some unspecified internal error has occurred.

### A.3.17.9.8 ixFpathAccTemplatePairClear (..)

ixFpathAccTemplatePairClear (IxAtmNpeRxVcId npeVcId)

Disable a template pair for a VC.

This function disables fast path processing for a particular VC by clearing the previously provisioned template pair for that VC. It also has the side effect of clearing the statistics for the npeVcId.

#### See also:

ixFpathAccTemplatePairSet

### Parameters

***npeVcId*** — The identifier for a VC supplied previously in a call to ixFpathAccTemplatePairSet.

### Returns

- IX_SUCCESS — Successfully completed the clear operation
- IX_FAIL — Invalid parameter or some unspecified internal error has occurred. The state of the Npe(s) is indeterminate.

## A.3.17.9.9 ixFpathAccTemplatePairSet (..)

ixFpathAccTemplatePairSet (IxAtmNpeRxVcId npeVcId, unsigned char * classifierTemplate, int classifierTemplateLength, unsigned char * modifierTemplate, int modifierTemplateLength)

Set the classifier/modifier template for a specific VC.

This function provisions the NPEs with the supplied classifier/modifier templates. The desired VC is indicated by the npeVcId and must have been previously obtained from IxAtmdAcc.

Packets will begin to flow on the fast path once the template pair has been installed.

The fast path requires a valid npeVcId at all times so if a VC configured via IxAtmdAcc is removed or changed such that it's npeVcId is different then the npeVcId must be refreshed by calling this function. Otherwise the fast path will not be enabled on the VC. In general the fast path service for a VC before it is disabled on IxAtmdAcc.

The integrity of the templates supplied to the service are not validated. Hence the it is the callers responsibility to ensure that the template contains a valid set of template instructions.

If this function is called twice or more in succession for a particular VC, the template values of the most recent call are provisioned on the NPEs. Hence this function can be used to periodically update the template pair with having to call ixFpathAccTemplatePairClear.

### Notes

- No fast path processing occurs on a VC while template are being set. A VC which has templates already set and hence has fast path enabled will have fast path processing disabled while the template update is executed.
- Template data MUST be in main memory when calling this API. The NPE engines copy the data from main memory only and not from cache. Failure to ensure this could cause the NPEs to be provisioned with erroneous template contents and cause unpredictable behavior.

### See also:

ixFpathAccTemplatePairClear

### Parameters

***npeVcId*** — Identifier for a the VC obtained from IxAtmdAcc

***classifierTemplate*** — Pointer to byte array containing a combination of classifier instructions. This parameter cannot be NULL. The template must be constructed in network byte order (i.e. big endian). The template is internally copied by the fpath component.This template applies to a single VC and will be applied to all packets incoming on the VC once this function returns.

***classifierTemplateLength*** — The length in bytes of the classifier array. This value cannot be 0.

***modifierTemplate*** — Pointer to a byte array containing a combination of modifier instructions. The template is internally copied by the fpath component The pointer cannot be NULL.The template must be constructed in network byte order (i.e. big endian). This combination of modifier instructions will be applied to every packet which is passed by the classifier.

***modifierTemplateLength*** — The length in bytes of the modifier array. This value cannot be 0.

#### Returns

- IX_SUCCESS — Successfully provision the fast path with the supplied templates
- IX_FAIL — Invalid parameter or some unspecified internal failure has occurred.

### A.3.17.9.10 ixFpathAccWanMacAddrCallbackSet (..)

ixFpathAccWanMacAddrCallbackSet (IxFpathAccNewWanMacAddrCallback macAddrCallback, IxAtmNpeRxVcId npeVcId, IxFpathAccClientId clientId)

Set the Mac Address callback for a VC.

This function registers a callback to be called every time a new mac address is observed on the specified VC.

All Mac address transitions may not be notified in the case of high traffic loads.

A WAN MAC address detect instruction must be presented in the classifier template for MAC address observation service to be performed on a packet. Furthermore if the instruction appears as the first instruction in the template, MAC addresses for all packets on the VC will be observed, regardless of whether the packet ultimately goes fast path or slow path. On the other hand if it appears as the last instruction in the template only MAC addresses of packets which qualify for the past path will be observed.

Mac address observation is disabled if a template pair is cleared or when a template pair is being update via the ixFpathAccTemplatePairSet call.

Only one callback can be set per VC. Successive calls to this function will replace existing callbacks with newly supplied callback.

#### Notes

the callback function is execute in the context of the IxPsmMh message notification.

The WAN MAC address notification must be set before a template is set for a particular VC. Once a template is set this function will fail for that VC.

#### See also:

ixFpathAccTemplatePairSet

#### Parameters

***macAddrCallback*** — new Mac address callback. This parameter can be set to NULL to clear a previously submitted callback.

***npeVcId*** — VC identifier matching the npeVcId supplied in the call to ixFpathAccTemplatePairSet.

***clientId*** — Identifier to be used as a parameter to the callback.

**Returns**

- IX_SUCCESS — successfully stored the callback function.
- IX_FAIL — Invalid parameter or some unspecified internal error has occurred. The state of the Npe(s) is indeterminate.

## A.3.18 HSS Access (IxHssAcc) API

The public API for the Intel® IXP42X product line HssAccess component.

### A.3.18.1 Data Structures

- struct IxHssAccConfigParams
  Structure containing HSS configuration parameters.
- struct IxHssAccPktHdlcFraming
  This structure contains information required by the NPE to configure the HDLC co-processor.
- struct IxHssAccPortConfig
  Structure containing HSS port configuration parameters.

### A.3.18.2 Defines

- #define IX_HSSACC_TSLOTS_PER_HSS_PORT 128
  The max number of TDM timeslots supported per HSS port - 4E1's = 32x4 = 128.
- #define IX_HSSACC_PARAM_ERR 2
  HssAccess function return value for a parameter error.
- #define IX_HSSACC_RESOURCE_ERR 3
  HssAccess function return value for a resource error.
- #define IX_HSSACC_PKT_DISCONNECTING 4
  Indicates that a disconnect call is progressing and will disconnect soon.
- #define IX_HSSACC_Q_WRITE_OVERFLOW 5
  Indicates that an attempt to Tx or to replenish an RxFree Q failed due to Q overflow.
- #define IX_HSSACC_NO_ERROR 0
  HSS port no error present.
- #define IX_HSSACC_TX_FRM_SYNC_ERR 1
  HSS port TX Frame Sync error.
- #define IX_HSSACC_TX_OVER_RUN_ERR 2
  HSS port TX over-run error.
- #define IX_HSSACC_CHANNELISED_SW_TX_ERR 3
  NPE software error in channelised TX.

- #define IX_HSSACC_PACKETISED_SW_TX_ERR 4
  NPE software error in packetised TX.
- #define IX_HSSACC_RX_FRM_SYNC_ERR 5
  HSS port RX Frame Sync error.
- #define IX_HSSACC_RX_OVER_RUN_ERR 6
  HSS port RX over-run error.
- #define IX_HSSACC_CHANNELISED_SW_RX_ERR 7
  NPE software error in channelised RX.
- #define IX_HSSACC_PACKETISED_SW_RX_ERR 8
  NPE software error in packetised TX.
- #define IX_HSSACC_PKT_MIN_RX_MBUF_SIZE 64
  Minimum size of the Rx mbuf in bytes which the client must supply to the component.

### A.3.18.3 Typedefs

- typedef UINT32 IxHssAccPktUserId
  The client supplied value which will be supplied as a parameter with a given callback.
- typedef void(* IxHssAccLastErrorCallback )(unsigned lastHssError, unsigned servicePort)
  Prototype of the clients function to accept notification of the last error.
- typedef void(* IxHssAccPktRxCallback )(IX_MBUF *buffer, unsigned numHssErrs, IxHssAccPktStatus pktStatus, IxHssAccPktUserId rxUserId)
  Prototype of the clients function to accept notification of packetised rx.
- typedef void(* IxHssAccPktRxFreeLowCallback )(IxHssAccPktUserId rxFreeLowUserId)
  Prototype of the clients function to accept notification of requirement of more Rx Free buffers.
- typedef void(* IxHssAccPktTxDoneCallback )(IX_MBUF *buffer, unsigned numHssErrs, IxHssAccPktStatus pktStatus, IxHssAccPktUserId txDoneUserId)
  Prototype of the clients function to accept notification of completion with Tx buffers.
- typedef void(* IxHssAccChanRxCallback )(IxHssAccHssPort hssPortId, unsigned rxOffset, unsigned txOffset, unsigned numHssErrs)
  Prototype of the clients function to accept notification of channelised rx.

### A.3.18.4 Enumerations

- enum IxHssAccHssPort { IX_HSSACC_HSS_PORT_0, IX_HSSACC_HSS_PORT_MAX }
  The HSS port ID - There are two identical ports (0-1). Only port 0 will be supported in CSR1.0.
- enum IxHssAccHdlcPort { IX_HSSACC_HDLC_PORT_0, IX_HSSACC_HDLC_PORT_1, IX_HSSACC_HDLC_PORT_2, IX_HSSACC_HDLC_PORT_3, IX_HSSACC_HDLC_PORT_MAX }
  The HDLC port ID - There are four identical HDLC ports (0-3) per HSS port and they correspond to the 4 E1/T1 trunks.

- enum IxHssAccTdmSlotUsage { IX_HSSACC_TDMMAP_UNASSIGNED, IX_HSSACC_TDMMAP_HDLC, IX_HSSACC_TDMMAP_VOICE56K, IX_HSSACC_TDMMAP_VOICE64K, IX_HSSACC_TDMMAP_MAX }

  The HSS TDM stream timeslot assignment types.

- enum IxHssAccFrmSyncType { IX_HSSACC_FRM_SYNC_ACTIVE_LOW, IX_HSSACC_FRM_SYNC_ACTIVE_HIGH,IX_HSSACC_FRM_SYNC_FALLINGEDGE, IX_HSSACC_FRM_SYNC_RISINGEDGE, IX_HSSACC_FRM_SYNC_TYPE_MAX }

  The HSS frame sync pulse type.

- enum IxHssAccFrmSyncEnable { IX_HSSACC_FRM_SYNC_INPUT, IX_HSSACC_FRM_SYNC_INVALID_VALUE, IX_HSSACC_FRM_SYNC_OUTPUT_FALLING, IX_HSSACC_FRM_SYNC_OUTPUT_RISING, IX_HSSACC_FRM_SYNC_ENABLE_MAX }

  The IxHssAccFrmSyncEnable determines how the frame sync pulse is used.

- enum IxHssAccClkEdge { IX_HSSACC_CLK_EDGE_FALLING, IX_HSSACC_CLK_EDGE_RISING, IX_HSSACC_CLK_EDGE_MAX }

  IxHssAccClkEdge is used to determine the clk edge to use for framing and data.

- enum IxHssAccClkDir { IX_HSSACC_SYNC_CLK_DIR_INPUT, IX_HSSACC_SYNC_CLK_DIR_OUTPUT, IX_HSSACC_SYNC_CLK_DIR_MAX }

  The HSS clock direction.

- enum IxHssAccFrmPulseUsage { IX_HSSACC_FRM_PULSE_ENABLED, IX_HSSACC_FRM_PULSE_DISABLED, IX_HSSACC_FRM_PULSE_MAX }

  The HSS frame pulse usage.

- enum IxHssAccDataRate { IX_HSSACC_CLK_RATE, IX_HSSACC_HALF_CLK_RATE, IX_HSSACC_DATA_RATE_MAX }

  The HSS Data rate in relation to the clock.

- enum IxHssAccDataPolarity { IX_HSSACC_DATA_POLARITY_SAME, IX_HSSACC_DATA_POLARITY_INVERT, IX_HSSACC_DATA_POLARITY_MAX }

  The HSS data polarity type.

- enum IxHssAccBitEndian { IX_HSSACC_LSB_ENDIAN, IX_HSSACC_MSB_ENDIAN, IX_HSSACC_ENDIAN_MAX }

  HSS Data endianness.

- enum IxHssAccDrainMode { IX_HSSACC_TX_PINS_NORMAL, IX_HSSACC_TX_PINS_OPEN_DRAIN, IX_HSSACC_TX_PINS_MAX }

  Tx pin open drain mode.

- enum IxHssAccSOFType { IX_HSSACC_SOF_FBIT, IX_HSSACC_SOF_DATA, IX_HSSACC_SOF_MAX }

  HSS start of frame types.

- enum IxHssAccDataEnable { IX_HSSACC_DE_TRI_STATE, IX_HSSACC_DE_DATA, IX_HSSACC_DE_MAX }

  IxHssAccDataEnable is used to determine whether or not to drive the data pins.

- enum IxHssAccTxSigType { IX_HSSACC_TXSIG_LOW, IX_HSSACC_TXSIG_HIGH, IX_HSSACC_TXSIG_HIGH_IMP, IX_HSSACC_TXSIG_MAX }

  IxHssAccTxSigType is used to determine how to drive the data pins.

- enum IxHssAccFbType { IX_HSSACC_FB_FIFO, IX_HSSACC_FB_HIGH_IMP, IX_HSSACC_FB_MAX }

  IxHssAccFbType determines how to drive the Fbit.

- enum IxHssAcc56kEndianness { IX_HSSACC_56KE_BIT_7_UNUSED, IX_HSSACC_56KE_BIT_0_UNUSED, IX_HSSACC_56KE_MAX }

  56k data endianness when using the 56k type.

- enum IxHssAcc56kSel { IX_HSSACC_56KS_32_8_DATA, IX_HSSACC_56KS_56K_DATA, IX_HSSACC_56KS_MAX }

  56k data transmission type when using the 56k type.

- enum IxHssAccClkSpeed { IX_HSSACC_CLK_SPEED_512KHZ, IX_HSSACC_CLK_SPEED_1536KHZ, IX_HSSACC_CLK_SPEED_1544KHZ, IX_HSSACC_CLK_SPEED_1568KHZ, IX_HSSACC_CLK_SPEED_2048KHZ, IX_HSSACC_CLK_SPEED_4096KHZ, IX_HSSACC_CLK_SPEED_8192KHZ, IX_HSSACC_CLK_SPEED_MAX }

  IxHssAccClkSpeed represents the HSS clock speeds available.

- enum IxHssAccPktStatus { IX_HSSACC_PKT_OK, IX_HSSACC_STOP_SHUTDOWN_ERROR, IX_HSSACC_HDLC_ALN_ERROR, IX_HSSACC_HDLC_FCS_ERROR, IX_HSSACC_RXFREE_Q_EMPTY_ERROR, IX_HSSACC_HDLC_MAX_FRAME_SIZE_EXCEEDED, IX_HSSACC_HDLC_ABORT_ERROR, IX_HSSACC_DISCONNECT_IN_PROGRESS }

  Indicates the status of packets passed to the client.

- enum IxHssAccPktCrcType { IX_HSSACC_PKT_16_BIT_CRC = 16, IX_HSSACC_PKT_32_BIT_CRC = 32 }

  HDLC CRC type.

- enum IxHssAccPktHdlcIdleType { IX_HSSACC_HDLC_IDLE_ONES, IX_HSSACC_HDLC_IDLE_FLAGS }

  HDLC idle transmission type.

### A.3.18.5 Functions

- PUBLIC IX_STATUS ixHssAccPortInit (IxHssAccHssPort hssPortId, IxHssAccConfigParams *configParams, IxHssAccTdmSlotUsage *tdmMap, IxHssAccLastErrorCallback lastHssErrorCallback)

  Initialize a HSS port. No channelised or packetised connections should exist in the HssAccess layer while this interface is being called.

- PUBLIC IX_STATUS ixHssAccLastErrorRetrievalInitiate (IxHssAccHssPort hssPortId)

  Initiate the retrieval of the last HSS error. The HSS port should be configured before attempting to call this interface.

- PUBLIC IX_STATUS ixHssAccInit (void)

  This function is responsible for initializing resources for use by the packetised and channelised clients. It should be called before any other HssAccess interface is called. No other HssAccPacketised interface should be called while this interface is being processed.

- PUBLIC IX_STATUS ixHssAccPktPortConnect (IxHssAccHssPort hssPortId, IxHssAccHdlcPort hdlcPortId, BOOL hdlcFraming, unsigned blockSizeInWords, UINT32 rawIdleBlockPattern, IxHssAccPktHdlcFraming hdlcTxFraming, IxHssAccPktHdlcFraming hdlcRxFraming, unsigned frmFlagStart, IxHssAccPktRxCallback rxCallback,

IxHssAccPktUserId rxUserId, IxHssAccPktRxFreeLowCallback rxFreeLowCallback, IxHssAccPktUserId rxFreeLowUserId, IxHssAccPktTxDoneCallback txDoneCallback, IxHssAccPktUserId txDoneUserId)

This function is responsible for connecting a client to one of the 4 available HDLC ports. The HSS port should be configured before attempting a connect. No other HssAccPacketised interface should be called while this connect is being processed.

- PUBLIC IX_STATUS ixHssAccPktPortEnable (IxHssAccHssPort hssPortId, IxHssAccHdlcPort hdlcPortId)

  This function is responsible for enabling a packetised service for the specified HSS/HDLC port combination. It enables the RX flow. The client must have already connected to a packetised service and is responsible for ensuring an adequate amount of RX mbufs have been supplied to the access component before enabling the packetised service. This function must be called on a given port before any call to ixHssAccPktPortTx on the same port. No other HssAccPacketised interface should be called while this interface is being processed.

- PUBLIC IX_STATUS ixHssAccPktPortDisable (IxHssAccHssPort hssPortId, IxHssAccHdlcPort hdlcPortId)

  This function is responsible for disabling a packetised service for the specified HSS/HDLC port combination. It disables the RX flow. The client must have already connected to and enabled a packetised service for the specified HDLC port. This disable interface can be called before a disconnect, but is not required to.

- PUBLIC IX_STATUS ixHssAccPktPortDisconnect (IxHssAccHssPort hssPortId, IxHssAccHdlcPort hdlcPortId)

  This function is responsible for disconnecting a client from one of the 4 available HDLC ports. It is not required that the Rx Flow has been disabled before calling this function. If the RX Flow has not been disabled, the disconnect will disable it before proceeding with the disconnect. No other HssAccPacketised interface should be called while this interface is being processed.

- PUBLIC BOOL ixHssAccPktPortIsDisconnectComplete (IxHssAccHssPort hssPortId, IxHssAccHdlcPort hdlcPortId)

  This function is called to check if a given HSS/HDLC port combination is in a connected state or not. This function may be called at any time to determine a ports state. No other HssAccPacketised interface should be called while this interface is being processed.

- PUBLIC IX_STATUS ixHssAccPktPortRxFreeReplenish (IxHssAccHssPort hssPortId, IxHssAccHdlcPort hdlcPortId, IX_MBUF *buffer)

  Function which the client calls at regular intervals to provide mbufs to the access component for RX. A connection should exist for the specified hssPortId/hdlcPortId combination before attempting to call this interface. Also, the connection should not be in a disconnecting state.

- PUBLIC IX_STATUS ixHssAccPktPortTx (IxHssAccHssPort hssPortId, IxHssAccHdlcPort hdlcPortId, IX_MBUF *buffer)

  Function which the client calls when it wants to transmit packetised data. An enabled connection should exist on the specified hssPortId/hdlcPortId combination before attempting to call this interface. No other HssAccPacketised interface should be called while this interface is being processed.

- PUBLIC IX_STATUS ixHssAccChanConnect (IxHssAccHssPort hssPortId, unsigned bytesPerTSTrigger, UINT8 *rxCircular, unsigned numRxBytesPerTS, UINT32 *txPtrList, unsigned numTxPtrLists, unsigned numTxBytesPerBlk, IxHssAccChanRxCallback rxCallback)

  This function allows the client to connect to the Tx/Rx NPE Channelised Service. There can only be one client per HSS port. The client is responsible for ensuring that the HSS port is

configured appropriately before its connect request. No other HssAccChannelised interface should be called while this interface is being processed.

- PUBLIC IX_STATUS ixHssAccChanPortEnable (IxHssAccHssPort hssPortId)

  This function is responsible for enabling a channelised service for the specified HSS port. It enables the NPE RX flow. The client must have already connected to a channelised service before enabling the channelised service. No other HssAccChannelised interface should be called while this interface is being processed.

- PUBLIC IX_STATUS ixHssAccChanPortDisable (IxHssAccHssPort hssPortId)

  This function is responsible for disabling a channelised service for the specified HSS port. It disables the NPE RX flow. The client must have already connected to and enabled a channelised service for the specified HSS port. This disable interface can be called before a disconnect, but is not required to. No other HssAccChannelised interface should be called while this interface is being processed.

- PUBLIC IX_STATUS ixHssAccChanDisconnect (IxHssAccHssPort hssPortId)

  This function allows the client to Disconnect from a channelised service. If the NPE RX Flow has not been disabled, the disconnect will disable it before proceeding with other disconnect functionality. No other HssAccChannelised interface should be called while this interface is being processed.

- PUBLIC IX_STATUS ixHssAccChanStatusQuery (IxHssAccHssPort hssPortId, BOOL *dataRecvd, unsigned *rxOffset, unsigned *txOffset, unsigned *numHssErrs)

  This function is called by the client to query whether or not channelised data has been received. If there is, hssChanAcc will return the details in the output parameters. An enabled connection should exist on the specified hssPortId before attempting to call this interface. No other HssAccChannelised interface should be called while this interface is being processed.

- PUBLIC void ixHssAccShow (void)

  This function will display the current state of the IxHssAcc component. The output is sent to stdout.

- PUBLIC void ixHssAccStatsInit (void)

  This function will reset the IxHssAcc statistics.

## A.3.18.6 Detailed Description

The public API for the Intel® IXP42X product line HssAccess component.

IxHssAcc is the access layer to the HSS packetised and channelised services

## A.3.18.7 Typedef Documentation

### A.3.18.7.1 IxHssAccChanRxCallback

Prototype of the clients function to accept notification of channelised rx.

This callback, if defined by the client in the connect, will get called in the context of an IRQ. The IRQ will be triggered when the hssSyncQMQ is not empty. The queued entry will be dequeued and this function will be executed.

#### Parameters

***IxHssAccHssPort*** — hssPortId — The HSS port Id. There are two identical ports (0-1). Only port 0 will be supported in CSR1.0.

***unsigned txOffset*** (in) — an offset indicating from where within the txPtrList the NPE is currently transmitting from.

***unsigned rxOffset*** (in) — an offset indicating where within the receive buffers the NPE has just written the received data to.

***unsigned numHssErrs*** (in) — This is the number of hssErrors the Npe has received

#### Returns

void

### A.3.18.7.2 IxHssAccLastErrorCallback

Prototype of the clients function to accept notification of the last error.

This function is registered through the config. The client will initiate the last error retrieval. The HssAccess component will send a message to the NPE through the NPE Message Handler. When a response to the read is received, the NPE Message Handler will callback the HssAccess component which will execute this function in the same IxNpeMh context. The client will be passed the last error and the related service port (packetised 0-3, channelised 0)

#### Parameters

***unsigned lastHssError*** (in) — The last Hss error registered that has been registered.

***unsigned servicePort*** (in) — This is the service port number. (packetised 0-3, channelised 0)

#### Returns

void

### A.3.18.7.3 IxHssAccPktRxCallback

Prototype of the clients function to accept notification of packetised rx.

This function is registered through the ixHssAccPktPortConnect. hssPktAcc will pass received data in the form of mbufs to the client. The mbuf passed back to the client could contain a chain of buffers, depending on the packet size received.

#### Parameters

***IX_MBUF*** — buffer (in) - This is the mbuf which contains the payload received.

***unsigned numHssErrs*** (in) — This is the number of hssErrors the Npe has received

***IxHssAccPktStatus pktStatus*** (in) — This is the status of the mbuf that has been received.

***IxHssAccPktUserId rxUserId*** (in) — This is the client supplied value passed in at ixHssAccPktPortConnect time which is now returned to the client.

#### Returns

void

### A.3.18.7.4 IxHssAccPktRxFreeLowCallback

Prototype of the clients function to accept notification of requirement of more Rx Free buffers.

The client can choose to register a callback of this type when calling a connecting. This function is registered through the ixHssAccPktPortConnect. If defined, the access layer will provide the trigger for this callback. The callback will be responsible for supplying mbufs to the access layer for use on the receive path from the HSS using ixHssPktAccFreeBufReplenish.

#### Returns

void

### A.3.18.7.5 IxHssAccPktTxDoneCallback

Prototype of the clients function to accept notification of completion with Tx buffers.

This function is registered through the ixHssAccPktPortConnect. It enables the hssPktAcc to pass buffers back to the client when transmission is complete.

#### Parameters

***IX_MBUF *buffer*** (in) — This is the mbuf which contained the payload that was for Tx.

***unsigned numHssErrs*** (in) — This is the number of hssErrors the Npe has received

***IxHssAccPktStatus pktStatus*** (in) — This is the status of the mbuf that has been transmitted.

***IxHssAccPktUserId txDoneUserId*** (in) — This is the client supplied value passed in at ixHssAccPktPortConnect time which is now returned to the client.

#### Returns

void

### A.3.18.7.6 UINT32 IxHssAccPktUserId

The client supplied value which will be supplied as a parameter with a given callback.

This value will be passed into the ixHssAccPktPortConnect function once each with given callbacks. This value will then be passed back to the client as one of the parameters to each of these callbacks, when these callbacks are called.

## A.3.18.8 Enumeration Type Documentation

### A.3.18.8.1 enum IxHssAcc56kEndianness

56k data endianness when using the 56k type.

#### Enumeration values

- IX_HSSACC_56KE_BIT_7_UNUSED — High bit is unused.

- IX_HSSACC_56KE_BIT_0_UNUSED — Low bit is unused.
- IX_HSSACC_56KE_MAX — Delimiter for error checks.

#### A.3.18.8.2 enum IxHssAcc56kSel

56k data transmission type when using the 56k type.

**Enumeration values**

- IX_HSSACC_56KS_32_8_DATA — 32/8 bit data.
- IX_HSSACC_56KS_56K_DATA — 56K data.
- IX_HSSACC_56KS_MAX — Delimiter for error checks.

#### A.3.18.8.3 enum IxHssAccBitEndian

HSS Data endianness.

**Enumeration values**

- IX_HSSACC_LSB_ENDIAN TX/RX — Least Significant Bit first.
- IX_HSSACC_MSB_ENDIAN TX/RX — Most Significant Bit first.
- IX_HSSACC_ENDIAN_MAX — Delimiter for the purposes of error checks.

#### A.3.18.8.4 enum IxHssAccClkDir

The HSS clock direction.

**Enumeration values**

- IX_HSSACC_SYNC_CLK_DIR_INPUT — Clock is an input.
- IX_HSSACC_SYNC_CLK_DIR_OUTPUT — Clock is an output.
- IX_HSSACC_SYNC_CLK_DIR_MAX — Delimiter for error checks.

#### A.3.18.8.5 enum IxHssAccClkEdge

IxHssAccClkEdge is used to determine the clk edge to use for framing and data.

**Enumeration values**

- IX_HSSACC_CLK_EDGE_FALLING — Clock sampled off a falling edge.
- IX_HSSACC_CLK_EDGE_RISING — Clock sampled off a rising edge.
- IX_HSSACC_CLK_EDGE_MAX — Delimiter for error checks.

#### A.3.18.8.6 enum IxHssAccClkSpeed

IxHssAccClkSpeed represents the HSS clock speeds available.

**Enumeration values**

- IX_HSSACC_CLK_SPEED_512KHZ — 512 KHz.
- IX_HSSACC_CLK_SPEED_1536KHZ — 1.536 MHz.
- IX_HSSACC_CLK_SPEED_1544KHZ — 1.544 MHz.

- IX_HSSACC_CLK_SPEED_1568KHZ — 1.568 MHz.
- IX_HSSACC_CLK_SPEED_2048KHZ — 2.048 MHz.
- IX_HSSACC_CLK_SPEED_4096KHZ — 4.096 MHz.
- IX_HSSACC_CLK_SPEED_8192KHZ — 8.192 MHz.
- IX_HSSACC_CLK_SPEED_MAX — Delimiter for error checking.

#### A.3.18.8.7 enum IxHssAccDataEnable

IxHssAccDataEnable is used to determine whether or not to drive the data pins.

**Enumeration values**

- IX_HSSACC_DE_TRI_STATE — Tri-state the data pins.
- IX_HSSACC_DE_DATA — Push data out the data pins.
- IX_HSSACC_DE_MAX — Delimiter for error checks.

#### A.3.18.8.8 enum IxHssAccDataPolarity

The HSS data polarity type.

**Enumeration values**

- IX_HSSACC_DATA_POLARITY_SAME — Don't invert data between NPE and HSS FIFOs.
- IX_HSSACC_DATA_POLARITY_INVERT — Invert data between NPE and HSS FIFOs.
- IX_HSSACC_DATA_POLARITY_MAX — Delimiter for error checks.

#### A.3.18.8.9 enum IxHssAccDataRate

The HSS Data rate in relation to the clock.

**Enumeration values**

- IX_HSSACC_CLK_RATE — Data rate is at the configured clk speed.
- IX_HSSACC_HALF_CLK_RATE — Data rate is half the configured clk speed.
- IX_HSSACC_DATA_RATE_MAX — Delimiter for error checks.

#### A.3.18.8.10 enum IxHssAccDrainMode

Tx pin open drain mode.

**Enumeration values**

- IX_HSSACC_TX_PINS_NORMAL — Normal mode.
- IX_HSSACC_TX_PINS_OPEN_DRAIN — Open Drain mode.
- IX_HSSACC_TX_PINS_MAX — Delimiter for error checks.

#### A.3.18.8.11 enum IxHssAccFbType

IxHssAccFbType determines how to drive the Fbit.

***Warning:*** This will only be used for T1 @ 1.544MHz

**Enumeration values**

- IX_HSSACC_FB_FIFO — Fbit is dictated in FIFO.
- IX_HSSACC_FB_HIGH_IMP — Fbit is high impedance.
- IX_HSSACC_FB_MAX — Delimiter for error checks.

#### A.3.18.8.12 enum IxHssAccFrmPulseUsage

The HSS frame pulse usage.

**Enumeration values**

- IX_HSSACC_FRM_PULSE_ENABLED — Generate/Receive frame pulses.
- IX_HSSACC_FRM_PULSE_DISABLED — Disregard frame pulses.
- IX_HSSACC_FRM_PULSE_MAX — Delimiter for error checks.

#### A.3.18.8.13 enum IxHssAccFrmSyncEnable

The IxHssAccFrmSyncEnable determines how the frame sync pulse is used.

**Enumeration values**

- IX_HSSACC_FRM_SYNC_INPUT — Frame sync is sampled as an input.
- IX_HSSACC_FRM_SYNC_INVALID_VALUE — 1 is not used.
- IX_HSSACC_FRM_SYNC_OUTPUT_FALLING — Frame sync is an output generated off a falling clock edge.
- IX_HSSACC_FRM_SYNC_OUTPUT_RISING — Frame sync is an output generated off a rising clock edge.
- IX_HSSACC_FRM_SYNC_ENABLE_MAX — Delimiter for error checks.

#### A.3.18.8.14 enum IxHssAccFrmSyncType

The HSS frame sync pulse type.

**Enumeration values**

- IX_HSSACC_FRM_SYNC_ACTIVE_LOW — Frame sync is sampled low.
- IX_HSSACC_FRM_SYNC_ACTIVE_HIGH — sampled high.
- IX_HSSACC_FRM_SYNC_FALLINGEDGE — sampled on a falling edge.
- IX_HSSACC_FRM_SYNC_RISINGEDGE — sampled on a rising edge.
- IX_HSSACC_FRM_SYNC_TYPE_MAX — Delimiter for error checks.

#### A.3.18.8.15 enum IxHssAccHdlcPort

The HDLC port ID - There are four identical HDLC ports (0-3) per HSS port and they correspond to the 4 E1/T1 trunks.

**Enumeration values**

- IX_HSSACC_HDLC_PORT_0 — HDLC Port 0.
- IX_HSSACC_HDLC_PORT_1 — HDLC Port 1.
- IX_HSSACC_HDLC_PORT_2 — HDLC Port 2.
- IX_HSSACC_HDLC_PORT_3 — HDLC Port 3.
- IX_HSSACC_HDLC_PORT_MAX — Delimiter for error checks.

#### A.3.18.8.16 enum IxHssAccHssPort

The HSS port ID - There are two identical ports (0-1). Only port 0 will be supported in CSR1.0.

**Enumeration values**

- IX_HSSACC_HSS_PORT_0 — HSS Port 0.
- IX_HSSACC_HSS_PORT_MAX — Delimiter for error checks.

#### A.3.18.8.17 enum IxHssAccPktCrcType

HDLC CRC type.

**Enumeration values**

- IX_HSSACC_PKT_16_BIT_CRC — 16 bit CRC is being used.
- IX_HSSACC_PKT_32_BIT_CRC — 32 bit CRC is being used.

#### A.3.18.8.18 enum IxHssAccPktHdlcIdleType

HDLC idle transmission type.

**Enumeration values**

- IX_HSSACC_HDLC_IDLE_ONES — idle tx/rx will be a succession of ones.
- IX_HSSACC_HDLC_IDLE_FLAGS — idle tx/rx will be repeated flags.

#### A.3.18.8.19 enum IxHssAccPktStatus

Indicates the status of packets passed to the client.

**Enumeration values**

- IX_HSSACC_PKT_OK — Error free.
- IX_HSSACC_STOP_SHUTDOWN_ERROR — Errored due to stop or shutdown occurrence.
- IX_HSSACC_HDLC_ALN_ERROR — HDLC alignment error.
- IX_HSSACC_HDLC_FCS_ERROR — HDLC Frame Check Sum error.
- IX_HSSACC_RXFREE_Q_EMPTY_ERROR — RxFree Q became empty while receiving this packet.

- IX_HSSACC_HDLC_MAX_FRAME_SIZE_EXCEEDED — HDLC frame size received is greater than max specified at connect.
- IX_HSSACC_HDLC_ABORT_ERROR — HDLC frame received is invalid due to an abort sequence received.
- IX_HSSACC_DISCONNECT_IN_PROGRESS — Packet returned because a disconnect is in progress.

#### A.3.18.8.20 enum IxHssAccSOFType

HSS start of frame types.

**Enumeration values**

- IX_HSSACC_SOF_FBIT — Framing bit transmitted and expected on rx.
- IX_HSSACC_SOF_DATA — Framing bit not transmitted nor expected on rx.
- IX_HSSACC_SOF_MAX — Delimiter for error checks.

#### A.3.18.8.21 enum IxHssAccTdmSlotUsage

The HSS TDM stream timeslot assignment types.

**Enumeration values**

- IX_HSSACC_TDMMAP_UNASSIGNED — Unassigned.
- IX_HSSACC_TDMMAP_HDLC — HDLC packetised.
- IX_HSSACC_TDMMAP_VOICE56K — Voice56K channelised.
- IX_HSSACC_TDMMAP_VOICE64K — Voice64K channelised.
- IX_HSSACC_TDMMAP_MAX — Delimiter for error checks.

#### A.3.18.8.22 enum IxHssAccTxSigType

IxHssAccTxSigType is used to determine how to drive the data pins.

**Enumeration values**

- IX_HSSACC_TXSIG_LOW — Drive the data pins low.
- IX_HSSACC_TXSIG_HIGH — Drive the data pins high.
- IX_HSSACC_TXSIG_HIGH_IMP — Drive the data pins with high impedance.
- IX_HSSACC_TXSIG_MAX — Delimiter for error checks.

### A.3.18.9 Function Documentation

#### A.3.18.9.1 IX_STATUS ixHssAccChanConnect (..)

IX_STATUS ixHssAccChanConnect (IxHssAccHssPort hssPortId, unsigned bytesPerTSTrigger, UINT8 * rxCircular, unsigned numRxBytesPerTS, UINT32 * txPtrList, unsigned numTxPtrLists, unsigned numTxBytesPerBlk, IxHssAccChanRxCallback rxCallback)

This function allows the client to connect to the Tx/Rx NPE Channelised Service. There can only be one client per HSS port. The client is responsible for ensuring that the HSS port is configured appropriately before its connect request. No other HssAccChannelised interface should be called while this interface is being processed.

### Parameters

***IxHssAccHssPort*** hssPortId (in) — The HSS port Id. There are two identical ports (0-1). Only port 0 will be supported in CSR1.0.

***unsigned*** bytesPerTSTrigger (in) — The NPE will trigger the access component after bytesPerTSTrigger have been received for all trunk timeslots. This figure is a multiple of 8 e.g. 8 for 1ms trigger, 16 for 2ms trigger.

***UINT8*** *rxCircular (in) — A pointer to memory allocated by the client to be filled by data received. The buffer at this address is part of a pool of buffers to be accessed in a circular fashion. This address will be written to by the NPE. Therefore, it needs to be a physical address.

***unsigned*** numRxBytesPerTS (in) — The number of bytes allocated per timeslot within the receive memory. This figure will depend on the latency of the system. It needs to be deep enough for data to be read by the client before the NPE re-writes over that memory e.g. if the client samples at a rate of 40bytes per timeslot, numRxBytesPerTS may need to be 40bytes * 3. This would give the client 3 * 5ms of time before received data is over-written.

***UINT32*** *txPtrList (in) — The address of an area of contiguous memory allocated by the client to be populated with pointers to data for transmission. Each pointer list contains a pointer per active channel. The txPtrs will point to data to be transmitted by the NPE. Therefore, they must point to physical addresses.

***unsigned*** numTxPtrLists (in) — The number of pointer lists in txPtrList. This figure is dependent on jitter.

***unsigned*** numTxBytesPerBlk (in) — The size of the Tx data, in bytes, that each pointer within the PtrList points to.

***IxHssAccChanRxCallback*** rxCallback (in) — A client function pointer to be called back to handle the actual tx/rx of channelised data. If this is not NULL, an ISR will call this function. If this pointer is NULL, it implies that the client will use a polling mechanism to detect when the tx and rx of channelised data is to occur. The client will use hssChanAccStatus for this.

### Returns

- IX_SUCCESS — The function executed successfully
- IX_FAIL — The function did not execute successfully
- IX_HSSACC_PARAM_ERR — The function did not execute successfully due to a parameter error

## A.3.18.9.2 IX_STATUS ixHssAccChanDisconnect (..)

IX_STATUS ixHssAccChanDisconnect (IxHssAccHssPort hssPortId)

This function allows the client to Disconnect from a channelised service. If the NPE RX Flow has not been disabled, the disconnect will disable it before proceeding with other disconnect functionality. No other HssAccChannelised interface should be called while this interface is being processed.

#### Parameters

***IxHssAccHssPort*** hssPortId(in) — The HSS port Id. There are two identical ports (0-1). Only port 0 will be supported in CSR1.0.

#### Returns

- IX_SUCCESS — function executed successfully
- IX_FAIL — function did not execute successfully
- IX_HSSACC_PARAM_ERR — function did not execute successfully due to a parameter error

### A.3.18.9.3 IX_STATUS ixHssAccChanPortDisable (..)

IX_STATUS ixHssAccChanPortDisable (IxHssAccHssPort hssPortId)

This function is responsible for disabling a channelised service for the specified HSS port. It disables the NPE RX flow. The client must have already connected to and enabled a channelised service for the specified HSS port. This disable interface can be called before a disconnect, but is not required to. No other HssAccChannelised interface should be called while this interface is being processed.

#### Parameters

***IxHssAccHssPort*** hssPortId (in) — The HSS port Id. There are two identical ports (0-1). Only port 0 will be supported in CSR1.0.

#### Returns

- IX_SUCCESS — function executed successfully
- IX_FAIL — function did not execute successfully
- IX_HSSACC_PARAM_ERR — function did not execute successfully due to a parameter error

### A.3.18.9.4 IX_STATUS ixHssAccChanPortEnable (..)

IX_STATUS ixHssAccChanPortEnable (IxHssAccHssPort hssPortId)

This function is responsible for enabling a channelised service for the specified HSS port. It enables the NPE RX flow. The client must have already connected to a channelised service before enabling the channelised service. No other HssAccChannelised interface should be called while this interface is being processed.

#### Parameters

***IxHssAccHssPort*** hssPortId (in) — The HSS port Id. There are two identical ports (0-1). Only port 0 will be supported in CSR1.0.

#### Returns

- IX_SUCCESS — function executed successfully
- IX_FAIL — function did not execute successfully
- IX_HSSACC_PARAM_ERR — function did not execute successfully due to a parameter error

### A.3.18.9.5 IX_STATUS ixHssAccChanStatusQuery (..)

IX_STATUS ixHssAccChanStatusQuery (IxHssAccHssPort hssPortId, BOOL * dataRecvd, unsigned * rxOffset, unsigned * txOffset, unsigned * numHssErrs)

This function is called by the client to query whether or not channelised data has been received. If there is, hssChanAcc will return the details in the output parameters. An enabled connection should exist on the specified hssPortId before attempting to call this interface. No other HssAccChannelised interface should be called while this interface is being processed.

#### Parameters

***IxHssAccHssPort*** hssPortId (in) — The HSS port Id. There are two identical ports (0-1). Only port 0 will be supported in CSR1.0.

***BOOL*** *dataRecvd (out) — This BOOL indicates to the client whether or not the access component has read any data for the client. If FALSE, the other output parameters will not have been written to.

***unsigned*** *rxOffset (out) — An offset to indicate to the client where within the receive buffers the NPE has just written the received data to.

***unsigned*** *txOffset (out) — An offset to indicate to the client from where within the txPtrList the NPE is currently transmitting from

***unsigned*** *numHssErrs (out) — The total number of HSS port errors since initial port configuration

#### Returns

- IX_SUCCESS — function executed successfully
- IX_FAIL — function did not execute successfully
- IX_HSSACC_PARAM_ERR — function did not execute successfully due to a parameter error

### A.3.18.9.6 IX_STATUS ixHssAccInit (void)

This function is responsible for initializing resources for use by the packetised and channelised clients. It should be called before any other HssAccess interface is called. No other HssAccPacketised interface should be called while this interface is being processed.

#### Returns

- IX_SUCCESS — The function executed successfully
- IX_FAIL — The function did not execute successfully
- IX_HSSACC_RESOURCE_ERR — The function did not execute successfully due to a resource error

### A.3.18.9.7 IX_STATUS ixHssAccLastErrorRetrievalInitiate (..)

IX_STATUS ixHssAccLastErrorRetrievalInitiate (IxHssAccHssPort hssPortId)

Initiate the retrieval of the last HSS error. The HSS port should be configured before attempting to call this interface.

### Parameters

***IxHssAccHssPort hssPortId*** (in) — the HSS port ID

### Returns

- IX_SUCCESS — function executed successfully
- IX_FAIL — function did not execute successfully
- IX_HSSACC_PARAM_ERR — function did not execute successfully due to a parameter error

## A.3.18.9.8 ixHssAccPktPortConnect (..)

ixHssAccPktPortConnect (IxHssAccHssPort hssPortId, IxHssAccHdlcPort hdlcPortId, BOOL hdlcFraming, unsigned blockSizeInWords, UINT32 rawIdleBlockPattern, IxHssAccPktHdlcFraming hdlcTxFraming, IxHssAccPktHdlcFraming hdlcRxFraming, unsigned frmFlagStart, IxHssAccPktRxCallback rxCallback, IxHssAccPktUserId rxUserId, IxHssAccPktRxFreeLowCallback rxFreeLowCallback, IxHssAccPktUserId rxFreeLowUserId, IxHssAccPktTxDoneCallback txDoneCallback, IxHssAccPktUserId txDoneUserId)

This function is responsible for connecting a client to one of the 4 available HDLC ports. The HSS port should be configured before attempting a connect. No other HssAccPacketised interface should be called while this connect is being processed.

### Parameters

***IxHssAccHssPort*** hssPortId (in) — The HSS port Id. There are two identical ports (0-1). Only port 0 will be supported in CSR1.0.

***IxHssAccHdlcPort*** hdlcPortId (in) — This is the number of the HDLC port and it corresponds to the physical E1/T1 trunk i.e. 0, 1, 2, 3.

***BOOL*** hdlcFraming (in) — Value determines whether the service will use HDLC data or the debug, raw data type i.e. no HDLC processing.

***unsigned*** blockSizeInWords (in) — ax tx/rx block size.

***UINT32*** rawIdleBlockPattern (in) — Tx idle pattern in raw mode.

***IxHssAccHdlcFraming*** hdlcTxFraming (in) — This structure contains the following information required by the NPE to configure the HDLC co-processor for TX

***IxHssAccHdlcFraming*** hdlcRxFraming (in) — This structure contains the following information required by the NPE to configure the HDLC co-processor for RX

***unsigned*** frmFlagStart — Number of flags to precede to transmitted flags (0-2).

***IxHssAccPktRxCallback*** rxCallback (in) — Pointer to the clients packet receive function.

***IxHssAccPktUserId*** rxUserId (in) — The client supplied rx value to be passed back as an argument to the supplied rxCallback

***IxHssAccPktRxFreeLowCallback*** rxFreeLowCallback (in) — Pointer to the clients Rx free buffer request function. If NULL, assume client will trigger independently.

***IxHssAccPktUserId*** rxFreeLowUserId (in) — The client supplied RxFreeLow value to be passed back as an argument to the supplied rxFreeLowCallback

***IxHssAccPktTxDoneCallback*** txDoneCallback (in) — Pointer to the clients Tx done callback function

***IxHssAccPktUserId*** txDoneUserId (in) — The client supplied txDone value to be passed back as an argument to the supplied txDoneCallback

#### Returns

- IX_SUCCESS — The function executed successfully
- IX_FAIL — The function did not execute successfully
- IX_HSSACC_PARAM_ERR — The function did not execute successfully due to a parameter error
- IX_HSSACC_RESOURCE_ERR — The function did not execute successfully due to a resource error

### A.3.18.9.9 IX_STATUS ixHssAccPktPortDisable (..)

IX_STATUS ixHssAccPktPortDisable (IxHssAccHssPort hssPortId, IxHssAccHdlcPort hdlcPortId)

This function is responsible for disabling a packetised service for the specified HSS/HDLC port combination. It disables the RX flow. The client must have already connected to and enabled a packetised service for the specified HDLC port. This disable interface can be called before a disconnect, but is not required to.

#### Parameters

***IxHssAccHssPort*** hssPortId (in) — The HSS port Id. There are two identical ports (0-1). Only port 0 will be supported in CSR1.0.

***IxHssAccHdlcPort*** hdlcPortId (in) — The port id (0,1,2,3) to disable the service on.

#### Returns

- IX_SUCCESS — The function executed successfully
- IX_FAIL — The function did not execute successfully
- IX_HSSACC_PARAM_ERR — The function did not execute successfully due to a parameter error

### A.3.18.9.10 IX_STATUS ixHssAccPktPortDisconnect (..)

IX_STATUS ixHssAccPktPortDisconnect (IxHssAccHssPort hssPortId, IxHssAccHdlcPort hdlcPortId)

This function is responsible for disconnecting a client from one of the 4 available HDLC ports. It is not required that the Rx Flow has been disabled before calling this function. If the RX Flow has not been disabled, the disconnect will disable it before proceeding with the disconnect. No other HssAccPacketised interface should be called while this interface is being processed.

#### Parameters

***IxHssAccHssPort*** hssPortId (in) — The HSS port Id. There are two identical ports (0-1). Only port 0 will be supported in CSR1.0.

***IxHssAccHdlcPort*** hdlcPortId (in) — This is the number of the HDLC port to disconnect and it corresponds to the physical E1/T1 trunk i.e. 0, 1, 2, 3

#### Returns

- IX_SUCCESS — The function executed successfully
- IX_FAIL — The function did not execute successfully
- IX_HSSACC_PKT_DISCONNECTING — The function has initiated the disconnecting procedure but it has not completed yet.

### A.3.18.9.11 IX_STATUS ixHssAccPktPortEnable (..)

IX_STATUS ixHssAccPktPortEnable (IxHssAccHssPort hssPortId, IxHssAccHdlcPort hdlcPortId)

This function is responsible for enabling a packetised service for the specified HSS/HDLC port combination. It enables the RX flow. The client must have already connected to a packetised service and is responsible for ensuring an adequate amount of RX mbufs have been supplied to the access component before enabling the packetised service. This function must be called on a given port before any call to ixHssAccPktPortTx on the same port. No other HssAccPacketised interface should be called while this interface is being processed.

#### Parameters

***IxHssAccHssPort*** hssPortId (in) — The HSS port Id. There are two identical ports (0-1). Only port 0 will be supported in CSR1.0.

***IxHssAccHdlcPort*** hdlcPortId (in) — The port id (0,1,2,3) to enable the service on.

#### Returns

- IX_SUCCESS — The function executed successfully
- IX_FAIL — The function did not execute successfully
- IX_HSSACC_PARAM_ERR — The function did not execute successfully due to a parameter error

### A.3.18.9.12 BOOL ixHssAccPktPortIsDisconnectComplete (..)

BOOL ixHssAccPktPortIsDisconnectComplete (IxHssAccHssPort hssPortId, IxHssAccHdlcPort hdlcPortId)

This function is called to check if a given HSS/HDLC port combination is in a connected state or not. This function may be called at any time to determine a ports state. No other HssAccPacketised interface should be called while this interface is being processed.

#### Parameters

***IxHssAccHssPort*** hssPortId (in) — The HSS port Id. There are two identical ports (0-1). Only port 0 will be supported in CSR1.0.

***IxHssAccHdlcPort*** hdlcPortId (in) — This is the number of the HDLC port to disconnect and it corresponds to the physical E1/T1 trunk i.e. 0, 1, 2, 3

#### Returns

- TRUE — The state of this HSS/HDLC port combination is disconnected, so if a disconnect was called, it is now completed.
- FALSE — The state of this HSS/HDLC port combination is connected, so if a disconnect was called, it is not yet completed.

### A.3.18.9.13 IX_STATUS ixHssAccPktPortRxFreeReplenish (..)

IX_STATUS ixHssAccPktPortRxFreeReplenish (IxHssAccHssPort hssPortId, IxHssAccHdlcPort hdlcPortId, IX_MBUF * buffer)

Function which the client calls at regular intervals to provide mbufs to the access component for RX. A connection should exist for the specified hssPortId/hdlcPortId combination before attempting to call this interface. Also, the connection should not be in a disconnecting state.

#### Parameters

***IxHssAccHssPort*** hssPortId (in) — The HSS port Id. There are two identical ports (0-1). Only port 0 will be supported in CSR1.0.

***IxHssAccHdlcPort*** hdlcPortId (in) — This is the number of the HDLC port and it corresponds to the physical E1/T1 trunk i.e. 0, 1, 2, 3

***IX_MBUF*** *buffer (in) — A pointer to a free mbuf to filled with payload.

#### Returns

- IX_SUCCESS — The function executed successfully
- IX_FAIL — The function did not execute successfully
- IX_HSSACC_PARAM_ERR — The function did not execute successfully due to a parameter error
- IX_HSSACC_RESOURCE_ERR — The function did not execute successfully due to a resource error
- IX_HSSACC_Q_WRITE_OVERFLOW — The function did not succeed due to a Q overflow

### A.3.18.9.14 IX_STATUS ixHssAccPktPortTx (..)

IX_STATUS ixHssAccPktPortTx (IxHssAccHssPort hssPortId, IxHssAccHdlcPort hdlcPortId, IX_MBUF * buffer)

Function which the client calls when it wants to transmit packetised data. An enabled connection should exist on the specified hssPortId/hdlcPortId combination before attempting to call this interface. No other HssAccPacketised interface should be called while this interface is being processed.

#### Parameters

***IxHssAccHssPort*** hssPortId (in) — The HSS port Id. There are two identical ports (0-1). Only port 0 will be supported in CSR1.0.

***IxHssAccHdlcPort*** hdlcPortId (in) — This is the number of the HDLC port and it corresponds to the physical E1/T1 trunk i.e. 0, 1, 2, 3

***IX_MBUF*** *buffer (in) — A pointer to a chain of mbufs which the client has filled with the payload

#### Ret

- IX_SUCCESS — The function executed successfully
- IX_FAIL — The function did not execute successfully
- IX_HSSACC_PARAM_ERR — The function did not execute successfully due to a parameter error
- IX_HSSACC_RESOURCE_ERR — The function did not execute successfully due to a resource error. See note.
- IX_HSSACC_Q_WRITE_OVERFLOW — The function did not succeed due to a Q overflow

#### Note

IX_HSSACC_RESOURCE_ERR is returned when a free descriptor cannot be obtained to send the chain of mbufs to the NPE. This is a normal scenario. HssAcc has a pool of descriptors and this error means that they are currently all in use. The recommended approach to this is to retry until a descriptor becomes free and the packet is successfully transmitted. Alternatively, the user could wait until the next IxHssAccPktTxDoneCallback callback is triggered, and then retry, as it is this event that causes a transmit descriptor to be freed.

### A.3.18.9.15 IX_STATUS ixHssAccPortInit (..)

IX_STATUS ixHssAccPortInit (IxHssAccHssPort hssPortId, IxHssAccConfigParams * configParams, IxHssAccTdmSlotUsage * tdmMap, IxHssAccLastErrorCallback lastHssErrorCallback)

Initialize a HSS port. No channelised or packetised connections should exist in the HssAccess layer while this interface is being called.

#### Parameters

***IxHssAccHssPort*** hssPortId (in) — The HSS port Id. There are two identical ports (0-1). Only port 0 will be supported in CSR1.0

IxHssAccConfigParams *configParams (in) — Pointer to the HSS configuration structure

***IxHssAccTdmSlotUsage*** *tdmMap (in) — A pointer to an array of size IX_HSSACC_TSLOTS_PER_HSS_PORT, defining the slot usage over the HSS port

***IxHssAccLastErrorCallback*** lastHssErrorCallback (in) — Client callback to report last error

#### Returns

- IX_SUCCESS — The function executed successfully
- IX_FAIL — The function did not execute successfully
- IX_HSSACC_PARAM_ERR — The function did not execute successfully due to a parameter error

#### A.3.18.9.16 void ixHssAccShow (void)

This function will display the current state of the IxHssAcc component. The output is sent to stdout.

**Returns**

void

#### A.3.18.9.17 void ixHssAccStatsInit (void)

This function will reset the IxHssAcc statistics.

**Returns**

void

## A.3.19 NPE-A (IxNpeA) API

The Public API for the Intel® IXP42X product line NPE-A.

### A.3.19.1 Data Structures

- struct IxNpeA_AtmCellHeader
  Atm cell header definition.
- struct IxNpeA_AtmVcFp
  Atm Descriptor structure used for fpath.
- struct IxNpeA_NpePacketDescriptor
  HSS Packetised NpePacket Descriptor Structure.
- struct IxNpeA_RxAtmVc
  Rx Descriptor definition.
- struct IxNpeA_TxAtmVc
  Tx Descriptor definition.

### A.3.19.2 Defines

- #define IX_NPE_A_MSSG_ATM_UTOPIA_CONFIG_WRITE 0x20
  ATM Message ID command to write the data to the offset in the UTOPIA Configuration Table.
- #define IX_NPE_A_MSSG_ATM_UTOPIA_CONFIG_LOAD 0x21
  ATM Message ID command triggers the NPE to copy the UTOPIA Configuration Table to the UTOPIA coprocessor.
- #define IX_NPE_A_MSSG_ATM_UTOPIA_STATUS_UPLOAD 0x22
  ATM Message ID command triggers the NPE to read-back the UTOPIA status registers and update the UTOPIA Status Table.
- #define IX_NPE_A_MSSG_ATM_UTOPIA_STATUS_READ 0x23
  ATM Message ID command to read the UTOPIA Status Table at the specified offset.

- #define IX_NPE_A_MSSG_ATM_FP_TEMPLATE_WRITE 0x24

  ATM Message ID command to write the FastPath Template for fpIndex a word at a time.

- #define IX_NPE_A_MSSG_ATM_TX_ENABLE 0x25

  ATM Message ID command triggers the NPE to re-enable processing of any entries on the TxVcQ for this port.

- #define IX_NPE_A_MSSG_ATM_TX_DISABLE 0x26

  ATM Message ID command triggers the NPE to disable processing on this port.

- #define IX_NPE_A_MSSG_ATM_RX_ENABLE 0x27

  ATM Message ID command triggers the NPE to process any received cells for this VC according to the VC Lookup Table.

- #define IX_NPE_A_MSSG_ATM_RX_DISABLE 0x28

  ATM Message ID command triggers the NPE to disable processing for this VC.

- #define IX_NPE_A_MSSG_ATM_STATUS_READ 0x29

  ATM Message ID command to read the ATM status. The data is returned via a response message.

- #define IX_NPE_A_MSSG_ATM_FP_ENABLE 0x2a

  ATM Message ID command triggers the NPE to begin FastPath processing for this VC at the first cell of the next PDU.

- #define IX_NPE_A_MSSG_ATM_FP_DISABLE 0x2b

  ATM Message ID command triggers the NPE to stop FastPath processing with the completion of any current PDU.

- #define IX_NPE_A_MSSG_ATM_FP_STATISTICS_READ 0x2c

  ATM Message ID command to read the FastPath status word for the FastPath template identified by fpIndex.

- #define IX_NPE_A_MSSG_ATM_FP_WAN_MAC 0x2d

  ATM Message ID command generated by the NPE whenever a new MAC address is detected based upon monitoring of the 6 bytes in a PDU identified by the MacWanLearning operation.

- #define IX_NPE_A_MSSG_ATM_FP_STATISTICS_CLEAR 0x2e

  ATM Message ID command to clear the FastPath status word for the FastPath template.

- #define IX_NPE_A_MSSG_HSS_PORT_CONFIG_WRITE 0x40

  HSS Message ID command writes the ConfigWord value to the location in the HSS_CONFIG_TABLE specified by offset for HSS port hPort.

- #define IX_NPE_A_MSSG_HSS_PORT_CONFIG_LOAD 0x41

  HSS Message ID command triggers the NPE to copy the contents of the HSS Configuration Table to the appropriate configuration registers in the HSS coprocessor for the port specified by hPort.

- #define IX_NPE_A_MSSG_HSS_PORT_ERROR_READ 0x42

  HSS Message ID command triggers the NPE to return an HssErrorReadResponse message for HSS port hPort.

- #define IX_NPE_A_MSSG_HSS_CHAN_FLOW_ENABLE 0x43

  HSS Message ID command triggers the NPE to reset internal status and enable the HssChannelised operation on the HSS port specified by hPort.

- #define IX_NPE_A_MSSG_HSS_CHAN_FLOW_DISABLE 0x44

  HSS Message ID command triggers the NPE to disable the HssChannelised operation on the HSS port specified by hPort.

- #define IX_NPE_A_MSSG_HSS_CHAN_IDLE_PATTERN_WRITE 0x45

  HSS Message ID command writes the HSSnC_IDLE_PATTERN value for HSS port hPort. (n=hPort).

- #define IX_NPE_A_MSSG_HSS_CHAN_NUM_CHANS_WRITE 0x46

  HSS Message ID command writes the HSSnC_NUM_CHANNELS value for HSS port hPort. (n=hPort).

- #define IX_NPE_A_MSSG_HSS_CHAN_RX_BUF_ADDR_WRITE 0x47

  HSS Message ID command writes the HSSnC_RX_BUF_ADDR value for HSS port hPort. (n=hPort).

- #define IX_NPE_A_MSSG_HSS_CHAN_RX_BUF_CFG_WRITE 0x48

  HSS Message ID command writes the HSSnC_RX_BUF_SIZEB and HSSnC_RX_TRIG_PERIOD values for HSS port hPort. (n=hPort).

- #define IX_NPE_A_MSSG_HSS_CHAN_TX_BLK_CFG_WRITE 0x49

  HSS Message ID command writes the HSSnC_TX_BLK1_SIZEB, HSSnC_TX_BLK1_SIZEW, HSSnC_TX_BLK2_SIZEB, and HSSnC_TX_BLK2_SIZEW values for HSS port hPort. (n=hPort).

- #define IX_NPE_A_MSSG_HSS_CHAN_TX_BUF_ADDR_WRITE 0x4A

  HSS Message ID command writes the HSSnC_TX_BUF_ADDR value for HSS port hPort. (n=hPort).

- #define IX_NPE_A_MSSG_HSS_CHAN_TX_BUF_SIZE_WRITE 0x4B

  HSS Message ID command writes the HSSnC_TX_BUF_SIZEN value for HSS port hPort. (n=hPort).

- #define IX_NPE_A_MSSG_HSS_PKT_PIPE_FLOW_ENABLE 0x50

  HSS Message ID command triggers the NPE to reset internal status and enable the HssPacketised operation for the flow specified by pPipe on the HSS port specified by hPort.

- #define IX_NPE_A_MSSG_HSS_PKT_PIPE_FLOW_DISABLE 0x51

  HSS Message ID command triggers the NPE to disable the HssPacketised operation for the flow specified by pPipe on the HSS port specified by hPort.

- #define IX_NPE_A_MSSG_HSS_PKT_NUM_PIPES_WRITE 0x52

  HSS Message ID command writes the HSSnP_NUM_PIPES value for HSS port hPort.(n=hPort).

- #define IX_NPE_A_MSSG_HSS_PKT_PIPE_FIFO_SIZEW_WRITE 0x53

  HSS Message ID command writes the HSSnP_PIPEp_FIFOSIZEW value for packet-pipe pPipe on HSS port hPort. (n=hPort, p=pPipe).

- #define IX_NPE_A_MSSG_HSS_PKT_PIPE_HDLC_CFG_WRITE 0x54

  HSS Message ID command writes the HSSnP_PIPEp_HDLC_RXCFG and HSSnP_PIPEp_HDLC_TXCFG values for packet-pipe pPipe on HSS port hPort. (n=hPort, p=pPipe).

- #define IX_NPE_A_MSSG_HSS_PKT_PIPE_IDLE_PATTERN_WRITE 0x55

  HSS Message ID command writes the HSSnP_PIPEp_IDLE_PATTERN value for packet-pipe pPipe on HSS port hPort. (n=hPort, p=pPipe).

- #define IX_NPE_A_MSSG_HSS_PKT_PIPE_RX_SIZE_WRITE 0x56
  HSS Message ID command writes the HSSnP_PIPEp_RXSIZEB value for packet-pipe pPipe on HSS port hPort. (n=hPort, p=pPipe).
- #define IX_NPE_A_MSSG_HSS_PKT_PIPE_MODE_WRITE 0x57
  HSS Message ID command writes the HSSnP_PIPEp_MODE value for packet-pipe pPipe on HSS port hPort. (n=hPort, p=pPipe).
- #define IX_NPE_A_RXDESCRIPTOR_STATUS_OFFSET 0
  ATM Descriptor structure offset for Receive Descriptor Status field.
- #define IX_NPE_A_RXDESCRIPTOR_VCID_OFFSET 1
  ATM Descriptor structure offset for Receive Descriptor VC ID field.
- #define IX_NPE_A_RXDESCRIPTOR_CURRMBUFSIZE_OFFSET 2
  ATM Descriptor structure offset for Receive Descriptor Current Mbuf Size field.
- #define IX_NPE_A_RXDESCRIPTOR_ATMHEADER_OFFSET 4
  ATM Descriptor structure offset for Receive Descriptor ATM Header.
- #define IX_NPE_A_RXDESCRIPTOR_PSLOWPATH_OFFSET 8
  ATM Descriptor structure offset for Receive Descriptor Slow Path Pointer.
- #define IX_NPE_A_RXDESCRIPTOR_CURRMBUFLEN_OFFSET 12
  ATM Descriptor structure offset for Receive Descriptor Current MBuf length.
- #define IX_NPE_A_RXDESCRIPTOR_FPINDEX_OFFSET 14
  ATM Descriptor structure offset for Receive Descriptor Fast Path Index.
- #define IX_NPE_A_RXDESCRIPTOR_FPMATCHINDEX_OFFSET 15
  ATM Descriptor structure offset for Receive Descriptor Fast Path Match Index.
- #define IX_NPE_A_RXDESCRIPTOR_FPMBUFDATA_OFFSET 16
  ATM Descriptor structure offset for Receive Descriptor Fast MBuf Data.
- #define IX_NPE_A_RXDESCRIPTOR_PCURRMBUFF_OFFSET 20
  ATM Descriptor structure offset for Receive Descriptor Current MBuf Pointer.
- #define IX_NPE_A_RXDESCRIPTOR_PCURRMBUFDATA_OFFSET 24
  ATM Descriptor structure offset for Receive Descriptor Current MBuf Pointer.
- #define IX_NPE_A_RXDESCRIPTOR_PNEXTMBUF_OFFSET 28
  ATM Descriptor structure offset for Receive Descriptor Next MBuf Pointer.
- #define IX_NPE_A_RXDESCRIPTOR_TOTALLENGTH_OFFSET 32
  ATM Descriptor structure offset for Receive Descriptor Total Length.
- #define IX_NPE_A_RXDESCRIPTOR_AAL5CRCRESIDUE_OFFSET 36
  ATM Descriptor structure offset for Receive Descriptor AAL5 CRC Residue.
- #define IX_NPE_A_RXDESCRIPTOR_SIZE 40
  ATM Descriptor structure offset for Receive Descriptor Size.
- #define IX_NPE_A_TXDESCRIPTOR_PORT_OFFSET 0
  ATM Descriptor structure offset for Transmit Descriptor Port.
- #define IX_NPE_A_TXDESCRIPTOR_RSVD_OFFSET 1
  ATM Descriptor structure offset for Transmit Descriptor RSVD.

- #define IX_NPE_A_TXDESCRIPTOR_CURRMBUFLEN_OFFSET 2
  ATM Descriptor structure offset for Transmit Descriptor Current MBuf Length.
- #define IX_NPE_A_TXDESCRIPTOR_ATMHEADER_OFFSET 4
  ATM Descriptor structure offset for Transmit Descriptor ATM Header.
- #define IX_NPE_A_TXDESCRIPTOR_PCURRMBUFF_OFFSET 8
  ATM Descriptor structure offset for Transmit Descriptor Pointer to the current MBuf chain.
- #define IX_NPE_A_TXDESCRIPTOR_PCURRMBUFDATA_OFFSET 12
  ATM Descriptor structure offset for Transmit Descriptor Pointer to the current MBuf Data.
- #define IX_NPE_A_TXDESCRIPTOR_PNEXTMBUF_OFFSET 16
  ATM Descriptor structure offset for Transmit Descriptor Pointer to the Next MBuf chain.
- #define IX_NPE_A_TXDESCRIPTOR_TOTALLENGTH_OFFSET 20
  ATM Descriptor structure offset for Transmit Descriptor Total Length.
- #define IX_NPE_A_TXDESCRIPTOR_AAL5CRCRESIDUE_OFFSET 24
  ATM Descriptor structure offset for Transmit Descriptor AAL5 CRC Residue.
- #define IX_NPE_A_TXDESCRIPTOR_SIZE 28
  ATM Descriptor structure offset for Transmit Descriptor Size.
- #define IX_NPE_MPHY 1
  Define this macro to enable MPHY mode.
- #define IX_NPE_A_TXDONE_QUEUE_HIGHWATERMARK 2
  The NPE reserves the High Watermark for its operation. But it must be set by the Intel XScale core.
- #define IX_NPE_A_QMQ_FPATH_FREE_Q 0
  Queue ID for FastPath Free Queue.
- #define IX_NPE_A_QMQ_ATM_TX_DONE 1
  Queue ID for ATM Transmit Done queue.
- #define IX_NPE_A_QMQ_ATM_TX0 2
  Queue ID for ATM transmit Queue in a single phy configuration.
- #define IX_NPE_A_QMQ_ATM_TXID_MIN IX_NPE_A_QMQ_ATM_TX0
  Queue Manager Queue ID for ATM transmit Queue with minimum number of queue.
- #define IX_NPE_A_QMQ_ATM_TXID_MAX IX_NPE_A_QMQ_ATM_TX11
  Queue Manager Queue ID for ATM transmit Queue with maximum number of queue.
- #define IX_NPE_A_QMQ_ATM_RX_HI 21
  Queue Manager Queue ID for ATM Receive high Queue.
- #define IX_NPE_A_QMQ_ATM_RX_LO 22
  Queue Manager Queue ID for ATM Receive low Queue.
- #define IX_NPE_A_QMQ_ATM_TX1 IX_NPE_A_QMQ_ATM_TX0+1
  Queue ID for ATM transmit Queue Multiphy from 1 to 11.
- #define IX_NPE_A_QMQ_ATM_TX2 IX_NPE_A_QMQ_ATM_TX1+1
- #define IX_NPE_A_QMQ_ATM_TX3 IX_NPE_A_QMQ_ATM_TX2+1

- #define IX_NPE_A_QMQ_ATM_TX4 IX_NPE_A_QMQ_ATM_TX3+1
- #define IX_NPE_A_QMQ_ATM_TX5 IX_NPE_A_QMQ_ATM_TX4+1
- #define IX_NPE_A_QMQ_ATM_TX6 IX_NPE_A_QMQ_ATM_TX5+1
- #define IX_NPE_A_QMQ_ATM_TX7 IX_NPE_A_QMQ_ATM_TX6+1
- #define IX_NPE_A_QMQ_ATM_TX8 IX_NPE_A_QMQ_ATM_TX7+1
- #define IX_NPE_A_QMQ_ATM_TX9 IX_NPE_A_QMQ_ATM_TX8+1
- #define IX_NPE_A_QMQ_ATM_TX10 IX_NPE_A_QMQ_ATM_TX9+1
- #define IX_NPE_A_QMQ_ATM_TX11 IX_NPE_A_QMQ_ATM_TX10+1
- #define IX_NPE_A_QMQ_HSS_CHL_RX_TRIG 12
  Hardware QMgr Queue ID for HSS Channelised Receive trigger.
- #define IX_NPE_A_QMQ_HSS_PKT_RX 13
  Hardware QMgr Queue ID for HSS Packetised Receive.
- #define IX_NPE_A_QMQ_HSS_PKT_TX0 14
  Hardware QMgr Queue ID for HSS Packetised Transmit queue 0.
- #define IX_NPE_A_QMQ_HSS_PKT_TX1 15
  Hardware QMgr Queue ID for HSS Packetised Transmit queue 1.
- #define IX_NPE_A_QMQ_HSS_PKT_TX2 16
  Hardware QMgr Queue ID for HSS Packetised Transmit queue 2.
- #define IX_NPE_A_QMQ_HSS_PKT_TX3 17
  Hardware QMgr Queue ID for HSS Packetised Transmit queue 3.
- #define IX_NPE_A_QMQ_HSS_PKT_RX_FREE0 18
  Hardware QMgr Queue ID for HSS Packetised Receive Free queue 0.
- #define IX_NPE_A_QMQ_HSS_PKT_RX_FREE1 19
  Hardware QMgr Queue ID for HSS Packetised Receive Free queue 1.
- #define IX_NPE_A_QMQ_HSS_PKT_RX_FREE2 20
  Hardware QMgr Queue ID for HSS Packetised Receive Free queue 2.
- #define IX_NPE_A_QMQ_HSS_PKT_RX_FREE3 21
  Hardware QMgr Queue ID for HSS Packetised Receive Free queue 3.
- #define IX_NPE_A_QMQ_HSS_PKT_TX_DONE 22
  Hardware QMgr Queue ID for HSS Packetised Transmit Done queue.
- #define IX_NPE_A_QMQ_ATM_FREE_VC0 32
  Hardware QMgr Queue ID for ATM Free VC Queue.
- #define IX_NPE_A_QMQ_ATM_FREE_VC1 IX_NPE_A_QMQ_ATM_FREE_VC0+1
- #define IX_NPE_A_QMQ_ATM_FREE_VC2 IX_NPE_A_QMQ_ATM_FREE_VC1+1
- #define IX_NPE_A_QMQ_ATM_FREE_VC3 IX_NPE_A_QMQ_ATM_FREE_VC2+1
- #define IX_NPE_A_QMQ_ATM_FREE_VC4 IX_NPE_A_QMQ_ATM_FREE_VC3+1
- #define IX_NPE_A_QMQ_ATM_FREE_VC5 IX_NPE_A_QMQ_ATM_FREE_VC4+1
- #define IX_NPE_A_QMQ_ATM_FREE_VC6 IX_NPE_A_QMQ_ATM_FREE_VC5+1

- #define IX_NPE_A_QMQ_ATM_FREE_VC7 IX_NPE_A_QMQ_ATM_FREE_VC6+1
- #define IX_NPE_A_QMQ_ATM_FREE_VC8 IX_NPE_A_QMQ_ATM_FREE_VC7+1
- #define IX_NPE_A_QMQ_ATM_FREE_VC9 IX_NPE_A_QMQ_ATM_FREE_VC8+1
- #define IX_NPE_A_QMQ_ATM_FREE_VC10 IX_NPE_A_QMQ_ATM_FREE_VC9+1
- #define IX_NPE_A_QMQ_ATM_FREE_VC11 IX_NPE_A_QMQ_ATM_FREE_VC10+1
- #define IX_NPE_A_QMQ_ATM_FREE_VC12 IX_NPE_A_QMQ_ATM_FREE_VC11+1
- #define IX_NPE_A_QMQ_ATM_FREE_VC13 IX_NPE_A_QMQ_ATM_FREE_VC12+1
- #define IX_NPE_A_QMQ_ATM_FREE_VC14 IX_NPE_A_QMQ_ATM_FREE_VC13+1
- #define IX_NPE_A_QMQ_ATM_FREE_VC15 IX_NPE_A_QMQ_ATM_FREE_VC14+1
- #define IX_NPE_A_QMQ_ATM_FREE_VC16 IX_NPE_A_QMQ_ATM_FREE_VC15+1
- #define IX_NPE_A_QMQ_ATM_FREE_VC17 IX_NPE_A_QMQ_ATM_FREE_VC16+1
- #define IX_NPE_A_QMQ_ATM_FREE_VC18 IX_NPE_A_QMQ_ATM_FREE_VC17+1
- #define IX_NPE_A_QMQ_ATM_FREE_VC19 IX_NPE_A_QMQ_ATM_FREE_VC18+1
- #define IX_NPE_A_QMQ_ATM_FREE_VC20 IX_NPE_A_QMQ_ATM_FREE_VC19+1
- #define IX_NPE_A_QMQ_ATM_FREE_VC21 IX_NPE_A_QMQ_ATM_FREE_VC20+1
- #define IX_NPE_A_QMQ_ATM_FREE_VC22 IX_NPE_A_QMQ_ATM_FREE_VC21+1
- #define IX_NPE_A_QMQ_ATM_FREE_VC23 IX_NPE_A_QMQ_ATM_FREE_VC22+1
- #define IX_NPE_A_QMQ_ATM_FREE_VC24 IX_NPE_A_QMQ_ATM_FREE_VC23+1
- #define IX_NPE_A_QMQ_ATM_FREE_VC25 IX_NPE_A_QMQ_ATM_FREE_VC24+1
- #define IX_NPE_A_QMQ_ATM_FREE_VC26 IX_NPE_A_QMQ_ATM_FREE_VC25+1
- #define IX_NPE_A_QMQ_ATM_FREE_VC27 IX_NPE_A_QMQ_ATM_FREE_VC26+1
- #define IX_NPE_A_QMQ_ATM_FREE_VC28 IX_NPE_A_QMQ_ATM_FREE_VC27+1
- #define IX_NPE_A_QMQ_ATM_FREE_VC29 IX_NPE_A_QMQ_ATM_FREE_VC28+1
- #define IX_NPE_A_QMQ_ATM_FREE_VC30 IX_NPE_A_QMQ_ATM_FREE_VC29+1
- #define IX_NPE_A_QMQ_ATM_FREE_VC31 IX_NPE_A_QMQ_ATM_FREE_VC30+1
- #define IX_NPE_A_QMQ_ATM_RXFREE_MIN IX_NPE_A_QMQ_ATM_FREE_VC0
  The minimum queue ID for FreeVC queue.
- #define IX_NPE_A_QMQ_ATM_RXFREE_MAX IX_NPE_A_QMQ_ATM_FREE_VC31
  The maximum queue ID for FreeVC queue.
- #define IX_NPE_A_QMQ_OAM_FREE_VC 14
  OAM Rx Free queue ID.
- #define IX_NPE_A_CHAIN_DESC_COUNT_MAX 256
  Maximum number of chained MBufs that can be chained together.

### A.3.19.3 Enumerations

- enum IxNpeA_AalType { IX_NPE_A_AAL_TYPE_INVALID = 0, IX_NPE_A_AAL_TYPE_0_48 = 0x1, IX_NPE_A_AAL_TYPE_0_52 = 0x2, IX_NPE_A_AAL_TYPE_5 = 0x5, IX_NPE_A_AAL_TYPE_OAM = 0xF }

  NPE-A AAL Type.

- enum IxNpeA_PayloadFormat { IX_NPE_A_52_BYTE_PAYLOAD = 0, IX_NPE_A_48_BYTE_PAYLOAD }

  NPE-A Payload format 52-bytes & 48-bytes.

### A.3.19.4 Detailed Description

The Public API for the Intel® IXP42X product line NPE-A.

### A.3.19.5 Define Documentation

#### A.3.19.5.1 #define IX_NPE_A_MSSG_ATM_FP_DISABLE 0x2b

ATM Message ID command triggers the NPE to stop FastPath processing with the completion of any current PDU.

This command will be ignored for a vc already disabled for fastpath.

#### A.3.19.5.2 #define IX_NPE_A_MSSG_ATM_FP_ENABLE 0x2a

ATM Message ID command triggers the NPE to begin FastPath processing for this VC at the first cell of the next PDU.

Re-issuing this command for a vc already enabled for fastpath may cause erroneous behavior

#### A.3.19.5.3 #define IX_NPE_A_MSSG_ATM_RX_DISABLE 0x28

ATM Message ID command triggers the NPE to disable processing for this VC.

This command will be ignored for a VC already disabled

#### A.3.19.5.4 #define IX_NPE_A_MSSG_ATM_RX_ENABLE 0x27

ATM Message ID command triggers the NPE to process any received cells for this VC according to the VC Lookup Table.

Re-issuing this command with different contents for a VC that is not disabled will cause unpredictable behavior.

#### A.3.19.5.5 #define IX_NPE_A_MSSG_ATM_TX_DISABLE 0x26

ATM Message ID command triggers the NPE to disable processing on this port.

This command will be ignored for a port already disabled

#### A.3.19.5.6 #define IX_NPE_A_MSSG_ATM_TX_ENABLE 0x25

ATM Message ID command triggers the NPE to re-enable processing of any entries on the TxVcQ for this port.

This command will be ignored for a port already enabled

#### A.3.19.5.7 #define IX_NPE_A_QMQ_ATM_FREE_VC0 32

Hardware QMgr Queue ID for ATM Free VC Queue.

There are 32 Hardware QMgr Queue ID; from IX_NPE_A_QMQ_ATM_FREE_VC1 to IX_NPE_A_QMQ_ATM_FREE_VC30

#### A.3.19.5.8 #define IX_NPE_A_RXDESCRIPTOR_AAL5CRCRESIDUE_OFFSET 36

ATM Descriptor structure offset for Receive Descriptor AAL5 CRC Residue.

Current CRC value for a PDU

#### A.3.19.5.9 #define IX_NPE_A_RXDESCRIPTOR_CURRMBUFLEN_OFFSET 12

ATM Descriptor structure offset for Receive Descriptor Current MBuf length.

RX - Initialized to zero. The NPE updates this field as each cell is received and zeroes it with every new mbuf for chaining. Will not be bigger than currBbufSize.

#### A.3.19.5.10 #define IX_NPE_A_RXDESCRIPTOR_CURRMBUFSIZE_OFFSET 2

ATM Descriptor structure offset for Receive Descriptor Current Mbuf Size field.

Number of bytes the current mbuf data buffer can hold

#### A.3.19.5.11 #define IX_NPE_A_RXDESCRIPTOR_FPINDEX_OFFSET 14

ATM Descriptor structure offset for Receive Descriptor Fast Path Index.

Contains the FastPath index number of this VC

#### A.3.19.5.12 #define IX_NPE_A_RXDESCRIPTOR_FPMATCHINDEX_OFFSET 15

ATM Descriptor structure offset for Receive Descriptor Fast Path Match Index.

Contains the index of the matching array element of either the IP or TCP/UDP address. The field shall be ignored if the FP_index Classify/Modify templates are not IP or NAPT.

#### A.3.19.5.13 #define IX_NPE_A_RXDESCRIPTOR_FPMBUFDATA_OFFSET 16

ATM Descriptor structure offset for Receive Descriptor Fast MBuf Data.

Points to the beginning of the FastPath MBuf data block.

#### A.3.19.5.14 #define IX_NPE_A_RXDESCRIPTOR_PCURRMBUFDATA_OFFSET 24

ATM Descriptor structure offset for Receive Descriptor Current MBuf Pointer.

Pointer to the next byte to be read or next free location to be written.

#### A.3.19.5.15 #define IX_NPE_A_RXDESCRIPTOR_PCURRMBUFF_OFFSET 20

ATM Descriptor structure offset for Receive Descriptor Current MBuf Pointer.

The current mbuf pointer of a chain of mbufs.

#### A.3.19.5.16 #define IX_NPE_A_RXDESCRIPTOR_PNEXTMBUF_OFFSET 28

ATM Descriptor structure offset for Receive Descriptor Next MBuf Pointer.

Pointer to the next MBuf in a chain of MBufs.

#### A.3.19.5.17 #define IX_NPE_A_RXDESCRIPTOR_PSLOWPATH_OFFSET 8

ATM Descriptor structure offset for Receive Descriptor Slow Path Pointer.

Pointer to a slowpath descriptor which can be used by the Intel XScale core to replace the first FastPath mbuf in the associated chain.

#### A.3.19.5.18 #define IX_NPE_A_RXDESCRIPTOR_SIZE 40

ATM Descriptor structure offset for Receive Descriptor Size.

The size of the Receive descriptor

#### A.3.19.5.19 #define IX_NPE_A_RXDESCRIPTOR_STATUS_OFFSET 0

ATM Descriptor structure offset for Receive Descriptor Status field.

It is used for descriptor error reporting.

#### A.3.19.5.20 #define IX_NPE_A_RXDESCRIPTOR_TOTALLENGTH_OFFSET 32

ATM Descriptor structure offset for Receive Descriptor Total Length.

Total number of bytes written to the chain of MBufs by the NPE

#### A.3.19.5.21 #define IX_NPE_A_RXDESCRIPTOR_VCID_OFFSET 1

ATM Descriptor structure offset for Receive Descriptor VC ID field.

It is used to hold an identifier number for this VC

#### A.3.19.5.22 #define IX_NPE_A_TXDESCRIPTOR_AAL5CRCRESIDUE_OFFSET 24

ATM Descriptor structure offset for Transmit Descriptor AAL5 CRC Residue.

Current CRC value for a PDU

#### A.3.19.5.23 #define IX_NPE_A_TXDESCRIPTOR_CURRMBUFLEN_OFFSET 2

ATM Descriptor structure offset for Transmit Descriptor Current MBuf Length.

TX - Initialized by the Intel XScale core to the number of bytes in the current MBuf data buffer. The NPE decrements this field for every transmitted cell. Thus, when the NPE writes a descriptor the TxDone queue, this field will equal zero.

#### A.3.19.5.24 #define IX_NPE_A_TXDESCRIPTOR_PCURRMBUFDATA_OFFSET 12

ATM Descriptor structure offset for Transmit Descriptor Pointer to the current MBuf Data.

Pointer to the next byte to be read or next free location to be written.

#### A.3.19.5.25 #define IX_NPE_A_TXDESCRIPTOR_PORT_OFFSET 0

ATM Descriptor structure offset for Transmit Descriptor Port.

Port identifier.

#### A.3.19.5.26 #define IX_NPE_A_TXDESCRIPTOR_TOTALLENGTH_OFFSET 20

ATM Descriptor structure offset for Transmit Descriptor Total Length.

Total number of bytes written to the chain of MBufs by the NPE

## A.3.20 NPE-Downloader (IxNpeDl) API

The Public API for the Intel® IXP42X product line NPE Downloader.

### A.3.20.1 Data Structures

- struct IxNpeDlVersionId
  Version Id to identify each version contained in an image.

### A.3.20.2 Defines

- #define IX_NPEDL_PARAM_ERR 2
  NpeDl function return value for a parameter error.
- #define IX_NPEDL_RESOURCE_ERR 3
  NpeDl function return value for a resource error.
- #define IX_NPEDL_CRITICAL_NPE_ERR 4
  NpeDl function return value for a Critical NPE error occurring during download. Assume NPE is left in unstable condition if this value is returned.
- #define IX_NPEDL_CRITICAL_MICROCODE_ERR 5
  NpeDl function return value for a Critical Microcode error discovered during download. Assume NPE is left in unstable condition if this value is returned.

### A.3.20.3 Typedefs

- typedef UINT8 IxNpeDlBuildId
  Used to make up Build ID field of Version Id.
- typedef UINT8 IxNpeDlMajor
  Used to make up Major Release field of Version Id.
- typedef UINT8 IxNpeDlMinor
  Used to make up Minor Revision field of Version Id.

### A.3.20.4 Enumerations

- enum IxNpeDlNpeId { IX_NPEDL_NPEID_NPEA = 0, IX_NPEDL_NPEID_NPEB, IX_NPEDL_NPEID_NPEC, IX_NPEDL_NPEID_MAX }
  NpeId numbers to identify NPE A, B or C.

### A.3.20.5 Functions

- IX_STATUS ixNpeDlMicrocodeImageOverride (UINT32 *clientImage)
  This instructs NPE Downloader to use client-supplied microcode image.
- PUBLIC IX_STATUS ixNpeDlVersionDownload (IxNpeDlVersionId *versionIdPtr, BOOL verify)
  Download microcode to a NPE.
- PUBLIC IX_STATUS ixNpeDlAvailableVersionsCountGet (UINT32 *numVersionsPtr)
  Get the number of versions available in a microcode image.
- PUBLIC IX_STATUS ixNpeDlAvailableVersionsListGet (IxNpeDlVersionId *versionIdListPtr, UINT32 *listSizePtr)
  Get a list of the versions available in a microcode image.
- PUBLIC IX_STATUS ixNpeDlLoadedVersionGet (IxNpeDlNpeId npeId, IxNpeDlVersionId *versionIdPtr)
  Gets the Id of the version currently loaded on a particular NPE.
- PUBLIC IX_STATUS ixNpeDlNpeStopAndReset (IxNpeDlNpeId npeId)
  Stops and Resets an NPE.
- PUBLIC IX_STATUS ixNpeDlNpeExecutionStart (IxNpeDlNpeId npeId)
  Starts code execution on a NPE.
- PUBLIC IX_STATUS ixNpeDlNpeExecutionStop (IxNpeDlNpeId npeId)
  Stops code execution on a NPE.
- PUBLIC void ixNpeDlStatsShow (void)
  This function will display run-time statistics from the IxNpeDl component.
- PUBLIC void ixNpeDlStatsReset (void)
  This function will reset the statistics of the IxNpeDl component.

### A.3.20.6 Detailed Description

The Public API for the Intel® IXP42X product line NPE Downloader.

### A.3.20.7 Enumeration Type Documentation

#### A.3.20.7.1 enum IxNpeDlNpeId

NpeId numbers to identify NPE A, B or C.

**Enumeration values**

- IX_NPEDL_NPEID_NPEA — Identifies NPE A.
- IX_NPEDL_NPEID_NPEB — Identifies NPE B.
- IX_NPEDL_NPEID_NPEC — Identifies NPE C.
- IX_NPEDL_NPEID_MAX — Total Number of NPEs.

### A.3.20.8 Function Documentation

#### A.3.20.8.1 PUBLIC IX_STATUS ixNpeDlAvailableVersionsCountGet (UINT32 * *numVersionsPtr*)

Get the number of versions available in a microcode image.

**Parameters**

***UINT32*** * [out] numVersionsPtr — A pointer to the number of versions in the image.

Gets the number of versions available in the microcode image. Then returns this in a variable pointed to by *numVersionsPtr*.

**Notes**

Preconditions — The Client should declare the variable to which numVersionsPtr points

Postcondition —

**Returns**

- IX_SUCCESS — if the operation was successful
- IX_NPEDL_PARAM_ERR — if a parameter error occurred
- IX_FAIL — otherwise

#### A.3.20.8.2 PUBLIC IX_STATUS ixNpeDlAvailableVersionsListGet (..)

PUBLIC IX_STATUS ixNpeDlAvailableVersionsListGet (IxNpeDlVersionId * versionIdListPtr, UINT32 * listSizePtr)

Get a list of the versions available in a microcode image.

**Parameters**

IxNpeDlVersionId * [out] versionIdListPtr — Array to contain list of version Ids (memory allocated by Client).

***UINT32*** * [inout] listSizePtr — As an input, this param should point to the max number of Ids the *versionIdListPtr* array can hold. NpeDl will replace the input value of this parameter with the number of version Ids actually filled into the array before returning.

Finds list of versions available in the microcode image. Fills these into the array pointed to by *versionIdListPtr*

#### Notes

Preconditions:

- The Client should declare the variable to which numVersionsPtr points
- The Client should create an array (*versionIdListPtr*) large enough to contain all the version Ids in the image
- Postcondition:

#### Returns

- IX_SUCCESS — if the operation was successful
- IX_NPEDL_PARAM_ERR — if a parameter error occurred
- IX_FAIL — otherwise

### A.3.20.8.3 PUBLIC IX_STATUS ixNpeDlLoadedVersionGet (..)

PUBLIC IX_STATUS ixNpeDlLoadedVersionGet (IxNpeDlNpeId npeId, IxNpeDlVersionId * versionIdPtr)

Gets the Id of the version currently loaded on a particular NPE.

#### Parameters

***IxNpeDlNpeId*** [in] npeId — Id of the target NPE.

IxNpeDlVersionId * [out] versionIdPtr — Pointer to the where the version id should be stored.

If a version of microcode was previously downloaded successfully to the NPE by NPE Downloader, this function returns in *versionIdPtr* the version Id of that version loaded on the NPE.

#### Notes

Precondition:

- The Client has allocated memory to the *versionIdPtr* pointer.
- Postcondition:

#### Returns

- IX_SUCCESS — if the operation was successful
- IX_NPEDL_PARAM_ERR — if a parameter error occurred
- IX_FAIL — if the NPE doesn't currently have a version loaded

#### A.3.20.8.4 IX_STATUS ixNpeDlMicrocodeImageOverride (..)

IX_STATUS ixNpeDlMicrocodeImageOverride (UINT32 * clientImage)

This instructs NPE Downloader to use client-supplied microcode image.

**Parameters**

***UINT32*** * [in] clientImage — Pointer to the client-supplied NPE microcode image

This function sets NPE Downloader to use a client-supplied microcode image instead of the standard image which is included by the NPE Downloader. **This function is provided mainly for increased testability and should not be used in normal circumstances.** When not used, NPE Downloader will use a "built-in" image, local to this component, which should always contain the latest microcode for the NPEs.

**Notes**

Precondition:

- *clientImage* should point to a microcode image valid for use by the NPE Downloader component.

Postcondition:

- the client-supplied image will be used for all subsequent operations performed by the NPE Downloader

**Returns**

- IX_SUCCESS — if the operation was successful
- IX_NPEDL_PARAM_ERR — if a parameter error occurred
- IX_FAIL — if the client-supplied image did not contain a valid signature

#### A.3.20.8.5 PUBLIC IX_STATUS ixNpeDlNpeExecutionStart (..)

PUBLIC IX_STATUS ixNpeDlNpeExecutionStart (IxNpeDlNpeId npeId)

Starts code execution on a NPE.

**Parameters**

***IxNpeDlNpeId*** [in] npeId — Id of the target NPE

Starts execution of code on a particular NPE. A client would typically use this after a download to NPE is performed, to start/restart code execution on the NPE.

**Notes**

Precondition:

- The Client is responsible for ensuring mutual access to the NPE.
- Note that this function does not set the NPE Next Program Counter (NextPC), so it should be set beforehand if required by downloading appropriate State Information (using ixNpeDlVersionDownload()).

Postcondition:

#### Returns

- IX_SUCCESS — if the operation was successful
- IX_NPEDL_PARAM_ERR — if a parameter error occurred
- IX_FAIL — otherwise

### A.3.20.8.6 PUBLIC IX_STATUS ixNpeDlNpeExecutionStop (..)

PUBLIC IX_STATUS ixNpeDlNpeExecutionStop (IxNpeDlNpeId npeId)

Stops code execution on a NPE.

#### Parameters

***IxNpeDlNpeId*** [in] npeId — Id of the target NPE

Stops execution of code on a particular NPE. This would typically be used by a client before a download to NPE is performed, to stop code execution on an NPE, unless ixNpeDlNpeStopAndReset() is used instead. Unlike ixNpeDlNpeStopAndReset(), this function only halts the NPE and leaves all registers and settings intact. This is useful, for example, between stages of a multi-stage download, to stop the NPE prior to downloading the next version.

#### Notes

Precondition:

- The Client is responsible for ensuring mutual access to the NPE.

Postcondition:

#### Returns

- IX_SUCCESS — if the operation was successful
- IX_NPEDL_PARAM_ERR — if a parameter error occurred
- IX_FAIL — otherwise

### A.3.20.8.7 PUBLIC IX_STATUS ixNpeDlNpeStopAndReset (..)

PUBLIC IX_STATUS ixNpeDlNpeStopAndReset (IxNpeDlNpeId npeId)

Stops and Resets an NPE.

#### Parameters

***IxNpeDlNpeId*** [in] npeId — Id of the target NPE.

This function performs a soft NPE reset by writing reset values to particular NPE registers. Note that this does not reset NPE co-processors. The client can use this before downloading a new version of microcode to the NPE. This implicitly stops NPE code execution before resetting the NPE.

#### Notes

Precondition:

- The Client is responsible for ensuring mutual access to the NPE.

Postcondition:

#### Returns

- IX_SUCCESS — if the operation was successful
- IX_NPEDL_PARAM_ERR — if a parameter error occurred
- IX_FAIL — otherwise

### A.3.20.8.8 PUBLIC void ixNpeDlStatsReset (void)

This function will reset the statistics of the IxNpeDl component.

#### Returns

none

### A.3.20.8.9 PUBLIC void ixNpeDlStatsShow (void)

This function will display run-time statistics from the IxNpeDl component.

#### Returns

none

### A.3.20.8.10 PUBLIC IX_STATUS ixNpeDlVersionDownload (..)

PUBLIC IX_STATUS ixNpeDlVersionDownload (IxNpeDlVersionId * versionIdPtr, BOOL verify)

Download microcode to a NPE.

#### Parameters

IxNpeDlVersionId * [in] versionIdPtr — Pointer to Id of the microcode version to download.

***BOOL*** [in] verify — ON/OFF option to verify the download. If ON (verify == TRUE), the Downloader will read back each word written to the NPE registers to ensure the download operation was successful.

Using the *versionIdPtr*, this function locates a particular version of microcode in the microcode image in memory, and downloads the microcode to a particular NPE.

#### Notes

Precondition:

- The Client is responsible for ensuring mutual access to the NPE.
- The Client should stop the NPE before this function is used.

Postcondition:

- The NPE Instruction Pipeline will be cleared if State Information has been downloaded.
- If the download fails with a critical error, the NPE may be left in an unusable state.

**Returns**

- IX_SUCCESS — if the download was successful
- IX_NPEDL_PARAM_ERR — if a parameter error occurred
- IX_NPEDL_CRITICAL_NPE_ERR — if a critical NPE error occurred during download
- IX_PARAM_CRITICAL_MICROCODE_ERR — if a critical microcode error occurred during download
- IX_FAIL — otherwise

## A.3.21 NPE Message Handler (IxNpeMh) API

The public API for the Intel® IXP42X product line NPE Message Handler component.

### A.3.21.1 Data Structures

- struct IxNpeMhMessage
  The two-word message structure to send to and receive from the NPEs.

### A.3.21.2 Defines

- #define IX_NPEMH_MIN_MESSAGE_ID (0x00)
  minimum valid message ID.
- #define IX_NPEMH_MAX_MESSAGE_ID (0xFF)
  maximum valid message ID.
- #define IX_NPEMH_SEND_RETRIES_DEFAULT (3)
  default msg send retries.

### A.3.21.3 Typedefs

- typedef UINT32 IxNpeMhMessageId
  message ID.
- typedef void(* IxNpeMhCallback )(IxNpeMhNpeId, IxNpeMhMessage)
  This prototype shows the format of a message callback function.

### A.3.21.4 Enumerations

- enum IxNpeMhNpeId { IX_NPEMH_NPEID_NPEA = 0, IX_NPEMH_NPEID_NPEB, IX_NPEMH_NPEID_NPEC, IX_NPEMH_NUM_NPES }

  The ID of a particular NPE.

- enum IxNpeMhNpeInterrupts { IX_NPEMH_NPEINTERRUPTS_NO = 0, IX_NPEMH_NPEINTERRUPTS_YES }

  Indicator specifying whether or not NPE interrupts should drive receiving of messages from the NPEs.

### A.3.21.5 Functions

- IX_STATUS ixNpeMhInitialize (IxNpeMhNpeInterrupts npeInterrupts)

  This function will initialize the IxNpeMh component.

- IX_STATUS ixNpeMhUnsolicitedCallbackRegister (IxNpeMhNpeId npeId, IxNpeMhMessageId messageId, IxNpeMhCallback unsolicitedCallback)

  This function will register an unsolicited callback for a particular NPE and message ID.

- IX_STATUS ixNpeMhUnsolicitedCallbackForRangeRegister (IxNpeMhNpeId npeId, IxNpeMhMessageId minMessageId, IxNpeMhMessageId maxMessageId, IxNpeMhCallback unsolicitedCallback)

  This function will register an unsolicited callback for a particular NPE and range of message IDs.

- IX_STATUS ixNpeMhMessageSend (IxNpeMhNpeId npeId, IxNpeMhMessage message, UINT32 maxSendRetries)

  This function will send a message to a particular NPE.

- IX_STATUS ixNpeMhMessageWithResponseSend (IxNpeMhNpeId npeId, IxNpeMhMessage message, IxNpeMhMessageId solicitedMessageId, IxNpeMhCallback solicitedCallback, UINT32 maxSendRetries)

  This function is equivalent to the ixNpeMhMessageSend() function, but must be used when the message being sent will solicited a response.

- IX_STATUS ixNpeMhMessagesReceive (IxNpeMhNpeId npeId)

  This function will receive messages from a particular NPE and pass each message to the client via a solicited callback (for solicited messages) or an unsolicited callback (for unsolicited messages).

- IX_STATUS ixNpeMhShow (IxNpeMhNpeId npeId)

  This function will display the current state of the IxNpeMh component.

  Re-entrancy- The WAN MAC address notification must be set before a template is set for a particular VC. Once a template is set this function will fail for that VC.

  This function will be callable from any thread at any time. However, no resource protection will be used so as not to impact system performance. As this function is only reading statistical information then this is acceptable.

- IX_STATUS ixNpeMhShowReset (IxNpeMhNpeId npeId)

  This function will reset the current state of the IxNpeMh component.

  Re-entrancy- The WAN MAC address notification must be set before a template is set for a particular VC. Once a template is set this function will fail for that VC.

This function will be callable from any thread at any time. However, no resource protection will be used so as not to impact system performance. As this function is only writing statistical information then this is acceptable.

## A.3.21.6 Detailed Description

The public API for the Intel® IXP42X product line NPE Message Handler component.

## A.3.21.7 Typedef Documentation

### A.3.21.7.1 IxNpeMhCallback

This prototype shows the format of a message callback function.

This prototype shows the format of a message callback function. The message callback will be passed the message to be handled and will also be told from which NPE the message was received. The message callback will either be registered by ixNpeMhUnsolicitedCallbackRegister() or passed as a parameter to ixNpeMhMessageWithResponseSend(). It will be called from within an ISR triggered by the NPEs “outFIFO not empty” interrupt (see ixNpeMhInitialize()). The parameters passed are the ID of the NPE that the message was received from, and the message to be handled.

Re-entrancy — The WAN MAC address notification must be set before a template is set for a particular VC. Once a template is set this function will fail for that VC.

This function is only a prototype, and will be implemented by the client. It does not need to be re-entrant.

## A.3.21.8 Enumeration Type Documentation

### A.3.21.8.1 enum IxNpeMhNpeId

The ID of a particular NPE.

**Enumeration values**

The WAN MAC address notification must be set before a template is set for a particular VC. Once a template is set this function will fail for that VC.

- ***IX_NPEMH_NPEID_NPEA*** — ID for NPE-A.
- ***IX_NPEMH_NPEID_NPEB*** — ID for NPE-B.
- ***IX_NPEMH_NPEID_NPEC*** — ID for NPE-C.
- ***IX_NPEMH_NUM_NPES*** — Number of NPEs.

### A.3.21.8.2 enum IxNpeMhNpeInterrupts

Indicator specifying whether or not NPE interrupts should drive receiving of messages from the NPEs.

**Enumeration values**

- ***IX_NPEMH_NPEINTERRUPTS_NO*** — Don't use NPE interrupts.
- ***IX_NPEMH_NPEINTERRUPTS_YES*** — Do use NPE interrupts.

## A.3.21.9 Function Documentation

### A.3.21.9.1 IX_STATUS ixNpeMhInitialize (IxNpeMhNpeInterrupts *npeInterrupts*)

This function will initialise the IxNpeMh component.

#### Parameters

IxNpeMhNpeInterrupts npeInterrupts (in) - This parameter dictates whether or not the IxNpeMh component will service NPE "outFIFO not empty" interrupts to trigger receiving and processing of messages from the NPEs. If not then the client must use ixNpeMhMessagesReceive() to control message receiving and processing.

This function will initialise the IxNpeMh component. It should only be called once, prior to using the IxNpeMh component. The following actions will be performed by this function:

1. Initialization of internal data structures (e.g. solicited and unsolicited callback tables).
2. Configuration of the interface with the NPEs (e.g. enabling of NPE "outFIFO not empty" interrupts).
3. Registration of ISRs that will receive and handle messages when the NPEs' "outFIFO not empty" interrupts fire (if npeInterrupts equals IX_NPEMH_NPEINTERRUPTS_YES).

#### Returns

The function returns a status indicating success or failure.

### A.3.21.9.2 IX_STATUS ixNpeMhMessageSend (..)

IX_STATUS ixNpeMhMessageSend (IxNpeMhNpeId npeId, IxNpeMhMessage message, UINT32 maxSendRetries)

This function will send a message to a particular NPE.

#### Parameters

***IxNpeMhNpeId*** npeId (in) - The ID of the NPE to send the message to.

***IxNpeMhMessage*** message (in) - The message to send.

***UINT32*** maxSendRetries (in) - Max num. of retries to perform if the NPE’s inFIFO is full.

This function will send a message to a particular NPE. It will be the client's responsibility to ensure that the message is properly formed. The return status will signify to the client if the message was successfully sent or not.

If the message is sent to the NPE then this function will return a status of success. Note that this will only mean the message has been placed in the NPE’s inFIFO. There will be no way of knowing that the NPE has actually read the message, but once in the incoming message queue it will be safe to assume that the NPE will process it.

The inFIFO may fill up sometimes if the Intel XScale core is sending messages faster than the NPE can handle them. This forces us to retry attempts to send the message until the NPE services the inFIFO. The client should specify a ceiling value for the number of retries suitable to their needs.

IX_NPEMH_SEND_RETRIES_DEFAULT can be used as a default value for the *maxSendRetries* parameter for this function. Each retry exceeding this default number will incur a blocking delay of 1 microsecond, to avoid consuming too much AHB bus bandwidth while performing retries.

Note this function **must** only be used for messages. that do not solicit responses. If the message being sent will solicit a response then the ixNpeMhMessageWithResponseSend() function must be used to ensure that the response is correctly handled.

**Re-entrancy —** This function will be callable from any thread at any time. IxOsServices will be used for any necessary resource protection.

#### Returns

The function returns a status indicating success or failure.

### A.3.21.9.3 IX_STATUS ixNpeMhMessagesReceive (..)

IX_STATUS ixNpeMhMessagesReceive (IxNpeMhNpeId npeId)

This function will receive messages from a particular NPE and pass each message to the client via a solicited callback (for solicited messages) or an unsolicited callback (for unsolicited messages).

#### Parameters

***IxNpeMhNpeId*** npeId (in) — The ID of the NPE to receive and process messages from.

This function will receive messages from a particular NPE and pass each message to the client via a solicited callback (for solicited messages) or an unsolicited callback (for unsolicited messages).

If the IxNpeMh component is initialized to service NPE “outFIFO not empty” interrupts (see ixNpeMhInitialize()) then there is no need to call this function. This function is only provided as an alternative mechanism to control the receiving and processing of messages from the NPEs.

Note this function cannot be called from within an ISR as it will use resource protection mechanisms.

**Re-entrancy —** This function will be callable from any thread at any time. IxOsServices will be used for any necessary resource protection.

#### Returns

The function returns a status indicating success or failure.

### A.3.21.9.4 IX_STATUS ixNpeMhMessageWithResponseSend (..)

IX_STATUS ixNpeMhMessageWithResponseSend (IxNpeMhNpeId npeId, IxNpeMhMessage message, IxNpeMhMessageId solicitedMessageId, IxNpeMhCallback solicitedCallback, UINT32 maxSendRetries)

This function is equivalent to the ixNpeMhMessageSend() function, but must be used when the message being sent will solicited a response.

#### Parameters

***IxNpeMhNpeId*** npeId (in) — The ID of the NPE to send the message to.

***IxNpeMhMessage*** message (in) — The message to send.

***IxNpeMhMessageId*** solicitedMessageId (in) — The ID of the solicited response message.

***IxNpeMhCallback*** solicitedCallback (in) — The function to use to pass the response message back to the client. A value of NULL will cause the response message to be discarded.

***UINT32*** maxSendRetries (in) — Max num. of retries to perform if the NPE's inFIFO is full.

This function is equivalent to the ixNpeMhMessageSend() function, but must be used when the message being sent will solicited a response.

The client must specify the ID of the solicited response message to allow the response to be recognized when it is received. The client must also specify a callback function to handle the received response. The IxNpeMh component will not offer the facility to send a message to a NPE and receive a response within the same context.

Note if the client is not interested in the response, specifying a NULL callback will cause the response message to be discarded.

The solicited callback will be stored and called some time later from an ISR that will be triggered by the NPE's "outFIFO not empty" interrupt (see ixNpeMhInitialize()) to handle the response message corresponding to the message sent. Response messages will be handled in the order they are received.

The inFIFO may fill up sometimes if the Intel XScale core is sending messages faster than the NPE can handle them. This forces us to retry attempts to send the message until the NPE services the inFIFO. The client should specify a ceiling value for the number of retries suitable to their needs. IX_NPEMH_SEND_RETRIES_DEFAULT can be used as a default value for the *maxSendRetries* parameter for this function. Each retry exceeding this default number will incur a blocking delay of 1 microsecond, to avoid consuming too much AHB bus bandwidth while performing retries.

**Re-entrancy —** This function will be callable from any thread at any time. IxOsServices will be used for any necessary resource protection.

#### Returns

The function returns a status indicating success or failure.

### A.3.21.9.5 IX_STATUS ixNpeMhShow (..)

IX_STATUS ixNpeMhShow (IxNpeMhNpeId npeId)

This function will display the current state of the IxNpeMh component.

**Re-entrancy —** This function will be callable from any thread at any time. However, no resource protection will be used so as not to impact system performance. As this function is only reading statistical information then this is acceptable.

#### Parameters

***IxNpeMhNpeId*** npeId (in) — The ID of the NPE to display state information for.

#### Returns

The function returns a status indicating success or failure.

#### A.3.21.9.6 IX_STATUS ixNpeMhShowReset (..d)

IX_STATUS ixNpeMhShowReset (IxNpeMhNpeId npeId)

This function will reset the current state of the IxNpeMh component.

**Re-entrancy —** This function will be callable from any thread at any time. However, no resource protection will be used so as not to impact system performance. As this function is only writing statistical information then this is acceptable.

##### Parameters

***IxNpeMhNpeId*** npeId (in) — The ID of the NPE to reset state information for.

##### Returns

The function returns a status indicating success or failure.

#### A.3.21.9.7 IX_STATUS ixNpeMhUnsolicitedCallbackForRangeRegister (..)

IX_STATUS ixNpeMhUnsolicitedCallbackForRangeRegister (IxNpeMhNpeId npeId, IxNpeMhMessageId minMessageId, IxNpeMhMessageId maxMessageId, IxNpeMhCallback unsolicitedCallback)

This function will register an unsolicited callback for a particular NPE and range of message IDs.

##### Parameters

IxNpeMhNpeId npeId (in) — The ID of the NPE whose messages the unsolicited callback will handle.

IxNpeMhMessageId minMessageId (in) — The minimum message ID in the range of message IDs the unsolicited callback will handle.

IxNpeMhMessageId maxMessageId (in) — The maximum message ID in the range of message IDs the unsolicited callback will handle.

IxNpeMhCallback unsolicitedCallback (in) — The unsolicited callback function. A value of NULL will deregister any previously registered callback(s) for this NPE and range of message IDs.

This function will register an unsolicited callback for a particular NPE and range of message IDs. It is a convenience function that is effectively the same as calling ixNpeMhUnsolicitedCallbackRegister() for each ID in the specified range. See ixNpeMhUnsolicitedCallbackRegister() for more information.

Re-entrancy — This function will be callable from any thread at any time. IxOsServices will be used for any necessary resource protection.

##### Returns

The function returns a status indicating success or failure.

### A.3.21.9.8 IX_STATUS ixNpeMhUnsolicitedCallbackRegister (..)

IX_STATUS ixNpeMhUnsolicitedCallbackRegister (IxNpeMhNpeId npeId, IxNpeMhMessageId messageId, IxNpeMhCallback unsolicitedCallback)

This function will register an unsolicited callback for a particular NPE and message ID.

#### Parameters

***IxNpeMhNpeId*** npeId (in) — The ID of the NPE whose messages the unsolicited callback will handle.

***IxNpeMhMessageId*** messageId (in) — The ID of the messages the unsolicited callback will handle.

***IxNpeMhCallback*** unsolicitedCallback (in) — The unsolicited callback function. A value of NULL will deregister any previously registered callback for this NPE and message ID.

This function will register an unsolicited message callback for a particular NPE and message ID.

If an unsolicited callback is already registered for the specified NPE and message ID then the callback will be overwritten. Only one client will be responsible for handling a particular message ID associated with a NPE. Registering a NULL unsolicited callback will deregister any previously registered callback.

The callback function will be called from an ISR that will be triggered by the NPE's "outFIFO not empty" interrupt (see ixNpeMhInitialize()) to handle any unsolicited messages of the specific message ID received from the NPE. Unsolicited messages will be handled in the order they are received.

If no unsolicited callback can be found for a received message then it is assumed that the message is solicited.

If more than one client may be interested in a particular unsolicited message then the suggested strategy is to register a callback for the message that can itself distribute the message to multiple clients as necessary.

#### See also:

ixNpeMhUnsolicitedCallbackForRangeRegister().

**Re-entrancy —** This function will be callable from any thread at any time. IxOsServices will be used for any necessary resource protection.

#### Returns

The function returns a status indicating success or failure.

## A.3.22 OS Memory Buffer Management (IxOsBuffMgt) API

Intel® IXP42X product line OS Memory Buffer Management.

### A.3.22.1 Defines

- #define IX_MBUF  M_BLK
  Memory buffer.
- #define IX_MBUF_MDATA(m_blk_ptr)  (m_blk_ptr)->m_data
  Return pointer to the data in the mbuf.
- #define IX_MBUF_MLEN(m_blk_ptr)  (m_blk_ptr)->m_len
  Return pointer to the data length.
- #define IX_MBUF_TYPE(m_blk_ptr)  (m_blk_ptr)->m_type
  Return pointer to the data type in the mbuf.
- #define IX_MBUF_NEXT_BUFFER_IN_PKT_PTR(m_blk_ptr)  (m_blk_ptr)->m_next
  Return pointer to the next mbuf in a single packet.
- #define IX_MBUF_NEXT_PKT_IN_CHAIN_PTR(m_blk_ptr)  (m_blk_ptr)->m_nextpkt
  Return pointer to the next packet in the chain.
- #define IX_MBUF_ALLOCATED_BUFF_LEN(m_blk_ptr)  (m_blk_ptr)->m_len
  Return pointer to the allocated buffer size.
- #define IX_MBUF_PKT_LEN(m_blk_ptr)  (m_blk_ptr)->mBlkPktHdr.len
  Return pointer to the total length of all the data in the mbuf chain for this packet.

### A.3.22.2 Detailed Description

Intel® IXP42X product line OS Memory Buffer Management.

## A.3.23 OS Memory Buffer Pool Management (IxOsBuffPoolMgt) API

The Public API for the Buffer Pool Management component.

### A.3.23.1 Data Structures

- struct IxMbufPool
  Implementation of buffer pool structure for use with non-VxWorks OS.
- struct IxMbufPool
  Implementation of buffer pool structure for use with non-VxWorks OS.

### A.3.23.2 Defines

- #define IX_MBUF_POOL_SIZE_ALIGN(size)
  This macro takes an integer as an argument and rounds it up to be a multiple of the memory cache-line size.

- #define IX_MBUF_POOL_MBUF_AREA_SIZE_ALIGNED(count) (count * IX_MBUF_POOL_SIZE_ALIGN(sizeof(IX_MBUF)))

  This macro calculates, from the number of mbufs required, the size of the memory area required to contain the mbuf headers for the buffers in the pool. The size to be used for each mbuf header is rounded up to a multiple of the cache-line size, to ensure each mbuf header aligns on a cache-line boundary. This macro is used by IX_MBUF_POOL_MBUF_AREA_ALLOC().

- #define IX_MBUF_POOL_DATA_AREA_SIZE_ALIGNED(count, size) (count * (IX_MBUF_POOL_SIZE_ALIGN(size)))

  This macro calculates, from the number of mbufs required and the size of the data portion for each mbuf, the size of the data memory area required. The size is adjusted to ensure alignment on cache line boundaries. This macro is used by IX_MBUF_POOL_DATA_AREA_ALLOC().

- #define IX_MBUF_POOL_MBUF_AREA_ALLOC(count, memAreaSize)

  Allocates the memory area needed for the number of mbuf headers specified by count. This macro ensures the mbuf headers align on cache line boundaries. This macro evaluates to a pointer to the memory allocated.

- #define IX_MBUF_POOL_DATA_AREA_ALLOC(count, size, memAreaSize)

  Allocates the memory pool for the data portion of the pool mbufs. The number of mbufs is specified by count. The size of the data portion of each mbuf is specified by size. This macro ensures the mbufs are aligned on cache line boundaries This macro evaluates to a pointer to the memory allocated.

- #define IX_MBUF_POOL_FREE_COUNT(poolPtr) ((poolPtr)->freeBufsInPool)

  Returns the number of free buffers currently in the specified pool.

- #define IX_MBUF_MAX_POOLS 24

  The maximum number of pools that can be allocated.

- #define IX_MBUF_POOL_NAME_LEN 64

  The maximum string length of the pool name.

- #define IX_MBUF_POOL_INIT(poolPtrPtr, count, size, name) ixOsBuffPoolInit((poolPtrPtr), (count), (size), (name))

  Wrapper macro for ixOsBuffPoolInit() See function description below for details.

- #define IX_MBUF_POOL_INIT_NO_ALLOC(poolPtrPtr, bufPtr, dataPtr, count, size, name) ixOsBuffPoolInitNoAlloc((poolPtrPtr), (bufPtr), (dataPtr), (count), (size), (name))

  Wrapper macro for ixOsBuffPoolInitNoAlloc() See function description below for details.

- #define IX_MBUF_POOL_GET(poolPtr, bufPtrPtr) ixOsBuffPoolUnchainedBufGet((poolPtr), (bufPtrPtr))

  Wrapper macro for ixOsBuffPoolUnchainedBufGet() See function description below for details.

- #define IX_MBUF_POOL_PUT(bufPtr) ixOsBuffPoolBufFree(bufPtr)

  Wrapper macro for ixOsBuffPoolBufFree() See function description below for details.

- #define IX_MBUF_POOL_PUT_CHAIN(bufPtr) ixOsBuffPoolBufChainFree(bufPtr)

  Wrapper macro for ixOsBuffPoolBufChainFree() See function description below for details.

- #define IX_MBUF_POOL_SHOW(poolPtr) ixOsBuffPoolShow(poolPtr)

  Wrapper macro for ixOsBuffPoolShow() See function description below for details.

- #define IX_MBUF_POOL_MDATA_RESET(bufPtr) ixOsBuffPoolBufDataPtrReset(bufPtr)
  Wrapper macro for ixOsBuffPoolBufDataPtrReset() See function description below for details.

### A.3.23.3 Typedefs

- typedef IxMbufPool IX_MBUF_POOL

### A.3.23.4 Functions

- IX_STATUS ixOsBuffPoolInit (IX_MBUF_POOL **poolPtrPtr, int count, int size, char *name)
  This function creates a new buffer pool.
- IX_STATUS ixOsBuffPoolInitNoAlloc (IX_MBUF_POOL **poolPtrPtr, void *poolBufPtr, void *poolDataPtr, int count, int size, char *name)
  This function creates a new buffer pool, with user-allocated memory.
- IX_STATUS ixOsBuffPoolUnchainedBufGet (IX_MBUF_POOL *poolPtr, IX_MBUF **newBufPtrPtr)
  This function gets a buffer from the buffer pool.
- IX_MBUF * ixOsBuffPoolBufFree (IX_MBUF *bufPtr)
  This function returns a buffer to the buffer pool.
- void ixOsBuffPoolBufChainFree (IX_MBUF *bufPtr)
  This function returns a buffer chain to the buffer pool.
- IX_STATUS ixOsBuffPoolShow (IX_MBUF_POOL *poolPtr)
  This function prints pool statistics.
- IX_STATUS ixOsBuffPoolBufDataPtrReset (IX_MBUF *bufPtr)
  This function resets the data pointer of a buffer.

### A.3.23.5 Detailed Description

The Public API for the Buffer Pool Management component.

### A.3.23.6 Define Documentation

#### A.3.23.6.1 #define IX_MBUF_MAX_POOLS 24

The maximum number of pools that can be allocated.

**Note**

This can safely be increased if more pools are required.

#### A.3.23.6.2 #define IX_MBUF_POOL_DATA_AREA_ALLOC(count, size, memAreaSize)

Value:

```
IX_ACC_DRV_DMA_MALLOC((memAreaSize =                    \
        IX_MBUF_POOL_DATA_AREA_SIZE_ALIGNED(count,size)))
```

Allocates the memory pool for the data portion of the pool mbufs. The number of mbufs is specified by *count*. The size of the data portion of each mbuf is specified by *size*. This macro ensures the mbufs are aligned on cache line boundaries This macro evaluates to a pointer to the memory allocated.

#### Parameters

***int*** [in] count — the number of mbufs the pool will contain

***int*** [in] size — the desired size (in bytes) required for the data portion of each mbuf. Note that this size may be rounded up to ensure alignment on cache-line boundaries.

***int*** [out] memAreaSize — the total amount of memory allocated

#### Returns

void * — a pointer to the allocated memory area

### A.3.23.6.3 #define IX_MBUF_POOL_DATA_AREA_SIZE_ALIGNED(..)

#define IX_MBUF_POOL_DATA_AREA_SIZE_ALIGNED(count, size) (count * (IX_MBUF_POOL_SIZE_ALIGN(size)))

This macro calculates, from the number of mbufs required and the size of the data portion for each mbuf, the size of the data memory area required. The size is adjusted to ensure alignment on cache line boundaries. This macro is used by IX_MBUF_POOL_DATA_AREA_ALLOC().

#### Note

Refer to the WindRiver* “VxWorks* 5.5 OS Libraries API Reference” manual for “netBufLib” library documentation, which explains the vxWorks implementation of this macro below.

#### Parameters

***int*** [in] count — Number of mbufs in the pool.

***int*** [in] size — Desired size for each mbuf data portion. This size will be rounded up to a multiple of the cache-line size to ensure alignment on cache-line boundaries for each data block.

#### Returns

int - the total size required for the pool data area (aligned)

### A.3.23.6.4 #define IX_MBUF_POOL_FREE_COUNT(..)

#define IX_MBUF_POOL_FREE_COUNT(poolPtr) ((poolPtr)->freeBufsInPool)

Returns the number of free buffers currently in the specified pool.

#### Parameters

***IX_MBUF_POOL*** * [in] poolPtr — a pointer to the pool to query

#### Returns

int — the number of free buffers in the pool

### A.3.23.6.5 #define IX_MBUF_POOL_MBUF_AREA_ALLOC(..)

#define IX_MBUF_POOL_MBUF_AREA_ALLOC(count, memAreaSize)

Value:

```
IX_ACC_DRV_DMA_MALLOC((memAreaSize =                \
        IX_MBUF_POOL_MBUF_AREA_SIZE_ALIGNED(count)))
```

Allocates the memory area needed for the number of mbuf headers specified by *count*. This macro ensures the mbuf headers align on cache line boundaries. This macro evaluates to a pointer to the memory allocated.

#### Parameters

***int*** [in] count — the number of mbufs the pool will contain

***int*** [out] memAreaSize — the total amount of memory allocated

#### Returns

void * — a pointer to the allocated memory area

### A.3.23.6.6 #define IX_MBUF_POOL_MBUF_AREA_SIZE_ALIGNED(..)

#define IX_MBUF_POOL_MBUF_AREA_SIZE_ALIGNED(count) (count * IX_MBUF_POOL_SIZE_ALIGN(sizeof(IX_MBUF)))

This macro calculates, from the number of mbufs required, the size of the memory area required to contain the mbuf headers for the buffers in the pool. The size to be used for each mbuf header is rounded up to a multiple of the cache-line size, to ensure each mbuf header aligns on a cache-line boundary. This macro is used by IX_MBUF_POOL_MBUF_AREA_ALLOC().

#### Note

Refer to the WindRiver “VxWorks 5.5 OS Libraries API Reference” manual for “netBufLib” library documentation, which explains the vxWorks implementation of this macro below.

#### Parameters

***int*** [in] count — Number of buffers the pool will contain.

#### Returns

int — Total size required for the pool mbuf area (aligned).

### A.3.23.6.7 #define IX_MBUF_POOL_SIZE_ALIGN(size)

Value:

```
(((size + (IX_XSCALE_CACHE_LINE_SIZE - 1)) /      \
        IX_XSCALE_CACHE_LINE_SIZE) *                  \
            IX_XSCALE_CACHE_LINE_SIZE)
```

This macro takes an integer as an argument and rounds it up to be a multiple of the memory cache-line size.

#### Parameters

***int*** [in] size — the size integer to be rounded up

#### Returns

int — the size, rounded up to a multiple of the cache-line size

## A.3.23.7 Function Documentation

### A.3.23.7.1 ixOsBuffPoolBufChainFree (IX_MBUF * *bufPtr*)

This function returns a buffer chain to the buffer pool.

#### Parameters

***IX_MBUF*** * [in] bufPtr — Pointer to head of the chain.

This function returns a buffer chain to the pool, making the buffers available again to ixOsBuffPoolUnchainedBufGet(). The buffer pointed to by bufPtr can be chained or unchained. If it is chained, all buffers in the chain will be returned to the pool.

Thread Safe — yes

#### Notes

Precondition — bufPtr should point to a valid IX_MBUF structure

Postcondition — The buffer (or chain of buffers) supplied will be returned to the pool for reuse.

#### Returns

none

### A.3.23.7.2 ixOsBuffPoolBufDataPtrReset (IX_MBUF * *bufPtr*)

This function resets the data pointer of a buffer.

#### Parameters

***IX_MBUF*** * [in] bufPtr — Pointer to a valid IX_MBUF buffer.

This function resets the data pointer of a buffer to point to the start of the memory area allocated to the buffer for data (the buffer payload).

#### Notes

***Warning:*** This function can NOT be used if BOTH of following conditions are true:

- the pool was created using ixOsBuffPoolInitNoAlloc()
- a NULL value was supplied at the time of creation for the bufDataPtr parameter of ixOsBuffPoolInitNoAlloc(). See the API description of the function ixOsBuffPoolInitNoAlloc().

  Thread Safe: yes

Preconditions:

- *bufPtr* should point to a valid IX_MBUF structure
- A non-NULL pointer to the data memory area was supplied when the pool was created (see note above)
- The data pointer is pointing to somewhere within the buffer payload

Postcondition: The data pointer of the mbuf header will point to the start of the data payload section of the buffer, as it did when it was originally obtained from the pool

#### Returns

- IX_SUCCESS — if the operation was successful
- IX_FAIL — if the operation was not successful

### A.3.23.7.3 ixOsBuffPoolBufFree (IX_MBUF * *bufPtr*)

This function returns a buffer to the buffer pool.

#### Parameters

***IX_MBUF*** * [in] bufPtr — Pointer to a valid IX_MBUF buffer.

This function returns a buffer to the pool, making it available again to ixOsBuffPoolUnchainedBufGet(). The buffer pointed to by bufPtr can be chained or unchained. If it is chained, only the head of the chain will be freed to the pool, and a pointer to the next buffer in the chain will be returned to the caller.

Thread Safe — yes

#### Notes

Precondition — bufPtr should point to a valid IX_MBUF structure

Postcondition — The buffer supplied will be returned to the pool for reuse.

#### Returns

- If supplied buffer was chained, a pointer to the next buffer in the chain is returned
- Otherwise NULL is returned

### A.3.23.7.4 ixOsBuffPoolInit (..)

ixOsBuffPoolInit (IX_MBUF_POOL ** poolPtrPtr, int count, int size, char * name)

This function creates a new buffer pool.

@ingroup

#### Parameters

***IX_MBUF_POOL*** ** [out] poolPtrPtr — Pointer to a pool pointer.

***int*** [in] count — The number of buffers to have in the pool.

***int*** [in] size — The size of each buffer in the pool.

***char*** * [in] name — A name string for the pool (used in pool show).

This function initializes a pool of *count* buffers, each of size *size*. It allocates memory for the pool, fills in the pool and buffer data structures, and returns a pointer to the pool in the *poolPtrPtr* parameter. This pointer should be used with other functions on this API to use the pool. In the current implementation, only a limited number of pools can be allocated. The number of pools is decided by the value of *IX_MBUF_MAX_POOLS*.

#### Note

This function has 2 implementations, depending on which OS the code is compiled for. If compiled for VxWorks, an mbuf pool will be created using the VxWorks “netBufLib” OS library. This will produce a pool of mbufs which can be used with the netBufLib library routines if required. If compiled for a different OS, a pool of generic buffers will be produced. These may need to be converted to a different buffer format (such as sk_buffs for Linux*) to be used with OS network buffer manipulation routines if required. See the header file IxOsBuffMgt.h which maps the buffer implementations for each OS supported.

Thread Safe — no

#### Notes

Precondition — *poolPtrPtr* should point to a valid IX_MBUF_POOL pointer

Postcondition — A pool will be initialized and all memory required by the pool will be dynamically allocated from memory.

#### Returns

- IX_SUCCESS — if the operation was successful
- IX_FAIL — if the pool could not be created

### A.3.23.7.5 ixOsBuffPoolInitNoAlloc (..)

ixOsBuffPoolInitNoAlloc (IX_MBUF_POOL ** poolPtrPtr, void * poolBufPtr, void * poolDataPtr, int count, int size, char * name)

This function creates a new buffer pool, with user-allocated memory.

#### Parameters

IX_MBUF_POOL ** [out] poolPtrPtr — Pointer to a pool pointer.

void * [in] poolBufPtr — pointer to memory allocated with IX_MBUF_POOL_MBUF_AREA_ALLOC()

void * [in] poolDataPtr — pointer to memory allocated with IX_MBUF_POOL_DATA_AREA_ALLOC()

int [in] count — The number of buffers to have in the pool.

int [in] size — The size of each buffer in the pool (i.e. the amount of payload data octets each buffer can hold).

char * [in] name — A name string for the pool (used in pool show).

This function initializes a pool of *count* buffers, each of size *size*. It fills in the pool and buffer data structures, and returns a pointer to the pool in the *poolPtrPtr* parameter. This pointer should be used with other functions on this API to use the pool. In the current implementation, only a limited number of pools can be allocated. The number of pools is decided by the value of *IX_MBUF_MAX_POOLS*.

#### Notes

This function has 2 implementations, depending on which OS the code is compiled for. If compiled for VxWorks, an mbuf pool will be created using the VxWorks “netBufLib” OS library. This will produce a pool of mbufs which can be used with the netBufLib library routines if required. If compiled for a different OS, a pool of generic buffers will be produced. These may need to be converted to a different buffer format (such as sk_buffs for Linux) to be used with OS network buffer manipulation routines if required. See the header file IxOsBuffMgt.h which maps the buffer implementations for each OS supported.

The pointer to the data area can optionally be NULL, to indicate that the data memory area for the mbuf payload will be assigned by the user later on. In this case, it is expected that the user would assign the data pointer of each mbuf returned by ixOsBuffPoolUnchainedBufGet(). This also means that the function ixOsBuffPoolBufDataPtrReset() cannot be used on buffers from this pool. WARNING - This pointer CANNOT be NULL if VxWorks implementation is used!

Thread Safe — no

Precondition:

- *poolPtrPtr* should point to a valid IX_MBUF_POOL pointer
- The memory required for the pool, for mbuf structures and data (if required), should be allocated with the specified macros. See parameters *poolBufPtr* and *poolDataPtr*

Postcondition: A pool will be initialized and all memory required by the pool will be dynamically allocated from memory.

#### Returns

- IX_SUCCESS — if the operation was successful
- IX_FAIL i — f the pool could not be created

### A.3.23.7.6 ixOsBuffPoolShow (..)

ixOsBuffPoolShow (IX_MBUF_POOL * poolPtr)

This function prints pool statistics.

#### Parameters

***IX_MBUF_POOL*** * [in] poolPtr — A pointer to a valid pool.

This function prints pool statistics, such as the number of free buffers in each pool. The actual statistics printed depends on the implementation which may differ between platforms. This function can serve as a useful debugging aid.

Thread Safe — yes

#### Notes

Precondition — poolPtr should point to a valid IX_MBUF_POOL structure

Postcondition —

#### Returns

- IX_SUCCESS — if the operation was successful
- IX_FAIL — if the pool statistics could not be printed.

### A.3.23.7.7 ixOsBuffPoolUnchainedBufGet (..)

ixOsBuffPoolUnchainedBufGet (IX_MBUF_POOL * poolPtr, IX_MBUF ** newBufPtrPtr)

This function gets a buffer from the buffer pool.

#### Parameters

***IX_MBUF_POOL*** * [in] poolPtr — Pointer to a valid pool.

***IX_MBUF*** ** [out] newBufPtrPtr — A pointer to a valid IX_MBUF pointer.

This function gets a free buffer from the specified pool, and returns a pointer to the buffer in the *newBufPtrPtr* parameter. The buffer obtained will be a single unchained buffer of the size specified when the pool was initialized.

Thread Safe — yes

#### Notes

Precondition:

- *poolPtr* should point to a valid IX_MBUF_POOL structure
- *newBufPtrPtr* should point to a valid IX_MBUF pointer

Postcondition: A free buffer will be allocated from the pool and newBufPtrPtr can be dereferenced to access the pointer to the buffer.

#### Returns

- IX_SUCCESS — if the operation was successful
- IX_FAIL — if a free buffer could not be obtained

## A.3.24 OS Services (IxOsServices) API

This service provides a very thin layer of OS dependency services.

### A.3.24.1 Typedefs

- typedef int IX_IRQ_STATUS
- typedef pthread_mutex_t IxMutex
  Mutex object.

- typedef pthread_mutex_t IxFastMutex
  Fast mutex object.

### A.3.24.2 Enumerations

- enum IxOsServTraceLevels { LOG_NONE = 0, LOG_USER = 1, LOG_FATAL = 2, LOG_ERROR = 3, LOG_WARNING = 4, LOG_MESSAGE = 5, LOG_DEBUG1 = 6, LOG_DEBUG2 = 7, LOG_DEBUG3 = 8, LOG_ALL }
  Trace levels.

### A.3.24.3 Functions

- PUBLIC IX_STATUS ixOsServIntBind (int vector, void(*routine)(void *), void *parameter)
  binds a routine to a hardware interrupt.
- PUBLIC IX_STATUS ixOsServIntUnbind (int vector)
  unbinds a routine from a hardware interrupt.
- PUBLIC int ixOsServIntLock (void)
  locks out IRQs.
- PUBLIC void ixOsServIntUnlock (int lockKey)
  unlocks IRQs.
- PUBLIC int ixOsServIntLevelSet (int level)
  sets the interrupt level.
- PUBLIC IX_STATUS ixOsServMutexInit (IxMutex *mutex)
  initializes a mutex.
- PUBLIC IX_STATUS ixOsServMutexLock (IxMutex *mutex)
  locks the given mutex.
- PUBLIC IX_STATUS ixOsServMutexUnlock (IxMutex *mutex)
  unlocks the given mutex.
- PUBLIC IX_STATUS ixOsServMutexDestroy (IxMutex *mutex)
  destroys a mutex object.
- PUBLIC IX_STATUS ixOsServFastMutexInit (IxFastMutex *mutex)
  initializes a fast mutex.
- PUBLIC IX_STATUS ixOsServFastMutexTryLock (IxFastMutex *mutex)
  attempts to lock the fast mutex object.
- PUBLIC IX_STATUS ixOsServFastMutexUnlock (IxFastMutex *mutex)
  unlocks the fast mutex object.
- PUBLIC int ixOsServLog (int level, char *format, int arg1, int arg2, int arg3, int arg4, int arg5, int arg6)
  Logs a formatted message.
- PUBLIC int ixOsServLogLevelSet (int level)
  sets the logging level.

- PUBLIC void ixOsServSleep (int microseconds)
  execution block for a number of microseconds.
- PUBLIC void ixOsServTaskSleep (int milliseconds)
  preemptive execution block for a number of milliseconds.
- PUBLIC unsigned int ixOsServTimestampGet (void)
  used to retrieve the system timestamp.

### A.3.24.4 Detailed Description

This service provides a very thin layer of OS dependency services.

This file contains the API to the functions which are some what OS dependant and would require porting to a particular OS. A primary focus of the component development is to make them as OS independent as possible. All other components should abstract their OS dependency to this module.

#### Services Overview

- Trace Service - a simple debugging mechanism, with compile time debug trace level (depends on existent OS logging feature e.g. in Linux use kprintf, in VxWorks logMsg)
- Mutual Exclusion
  - — interrupt binding and locking mechanisms
  - — mutex locks and fast mutexes
- Timer Services
  - — timed delays, busy loop - microsecond granularity
  - — timed delays, OS dependent yielding - millisecond granularity
  - — timestamp measurements - Intel XScale core clock granularity

### A.3.24.5 Function Documentation

#### A.3.24.5.1 PUBLIC IX_STATUS ixOsServFastMutexInit (..)

PUBLIC IX_STATUS ixOsServFastMutexInit (IxFastMutex * mutex)

initializes a fast mutex.

#### Parameters

***mutex*** IxFastMutex * (in) — pointer to the mutex object

Initializes a fast mutex, placing it in “unlocked” state.

Can be called from interrupt level — yes

#### Returns

- IX_SUCCESS — if the operation succeeded or IX_FAIL otherwise

#### A.3.24.5.2 PUBLIC IX_STATUS ixOsServFastMutexTryLock (..)

PUBLIC IX_STATUS ixOsServFastMutexTryLock (IxFastMutex * mutex)

attempts to lock the fast mutex object.

**Parameters**

***mutex*** IxFastMutex * (in) — pointer to the mutex object

If the mutex is in “unlocked” state it becomes locked and the function returns “IX_SUCCESS”. If the mutex is already locked the function returns immediately with a IX_FAIL result.

Can be called from interrupt level — yes

**Returns**

- IX_SUCCESS — if the operation succeeded or IX_FAIL otherwise

#### A.3.24.5.3 PUBLIC IX_STATUS ixOsServFastMutexUnlock (..)

PUBLIC IX_STATUS ixOsServFastMutexUnlock (IxFastMutex * mutex)

unlocks the fast mutex object.

**Parameters**

***mutex*** IxFastMutex * (in) — pointer to the mutex object

Unlocks the given mutex object if locked, otherwise returns an error.

Can be called from interrupt level — yes

**Returns**

- IX_SUCCESS — if the operation succeeded or IX_FAIL otherwise

#### A.3.24.5.4 PUBLIC IX_STATUS ixOsServIntBind (..)

PUBLIC IX_STATUS ixOsServIntBind (int vector, void(* routine)(void *), void * parameter)

binds a routine to a hardware interrupt.

**Parameters**

***vector*** int (in) — interrupt vector to bind to

***routine*** void (*)(void *) (in) — routine to connect

***parameter*** void * (in) — parameter to pass to the routine when called

This functions binds the specified C routine to an interrupt vector. When called, the “parameter” value will be passed to the routine.

Can be called from interrupt level — no

#### Returns

- IX_SUCCESS — if the operation succeeded or IX_FAIL otherwise

### A.3.24.5.5 PUBLIC int ixOsServIntLevelSet (int *level*)

sets the interrupt level.

#### Parameters

***level*** int (in) — new interrupt level

This routine changes the interrupt mask in the status register to take on the value specified by level. Interrupts are locked out at or, depending on the implementation, below that level.

Can be called from interrupt level — yes

#### Returns

the previous interrupt level

***Warning:*** Do not call system functions when interrupts are locked

### A.3.24.5.6 PUBLIC int ixOsServIntLock (void)

locks out IRQs.

This function disables IRQs, returning a lock key which can be used later with ixOsServIntUnlock to re-enable the IRQs.

Can be called from interrupt level — yes

#### Returns

a lock key used by ixOsServIntUnlock to unlock IRQs

***Warning:*** Do not call system routines when IRQs are locked

### A.3.24.5.7 PUBLIC IX_STATUS ixOsServIntUnbind (int *vector*)

unbinds a routine from a hardware interrupt.

#### Parameters

***vector*** int (in) — interrupt vector to unbind from

This function unbinds from an interrupt vector a C routine previously bound with ixOsServIntBind

Can be called from interrupt level — no

#### Returns

- IX_SUCCESS — if the operation succeeded or IX_FAIL otherwise

### A.3.24.5.8 PUBLIC void ixOsServIntUnlock (int *lockKey*)

unlocks IRQs.

#### Parameters

***lockKey*** int (in) — lock key previously obtained with ixOsServIntLock()

This function reenables the IRQs locked out by ixOsServIntLock().

Can be called from interrupt level — yes

### A.3.24.5.9 PUBLIC int ixOsServLog (..)

PUBLIC int ixOsServLog (int level, char * format, int arg1, int arg2, int arg3, int arg4, int arg5, int arg6)

logs a formatted message.

#### Parameters

***level*** int (in) — trace level

***format*** char * (in) — format string, similar to printf()

***arg1*** int (in) — first argument to display

***arg2*** int (in) — second argument to display

***arg3*** int (in) — third argument to display

***arg4*** int (in) — fourth argument to display

***arg5*** int (in) — fifth argument to display

***arg6*** int (in) — sixth argument to display

This function logs the specified message via the logging task. It is similar to printf(), however it requires a log level and a fixed number of arguments. Unless called with the LOG_USER log level it will prefix the user message with a tag string like “[debug2]”.

Can be called from interrupt level — yes

#### Returns

the number of characters written

### A.3.24.5.10 PUBLIC int ixOsServLogLevelSet (int *level*)

sets the logging level.

#### Parameters

***level*** int (in) — logging level

This function sets the maximum level allowed for logging. Setting it to LOG_WARNING will disable all levels like LOG_MESSAGE, LOG_DEBUG1 etc. It returns the previous level which can be useful for restoring log states.

Can be called from interrupt level — yes

#### Returns

the previous logging level

### A.3.24.5.11 PUBLIC IX_STATUS ixOsServMutexDestroy (..)

PUBLIC IX_STATUS ixOsServMutexDestroy (IxMutex * mutex)

destroys a mutex object.

#### Parameters

***mutex*** IxMutex * (in) — pointer to mutex to destroy

Can be called from interrupt level — no

Destroys the mutex object, freeing the resources it might hold.

### A.3.24.5.12 PUBLIC IX_STATUS ixOsServMutexInit (..)

PUBLIC IX_STATUS ixOsServMutexInit (IxMutex * mutex)

initializes a mutex.

#### Parameters

***mutex*** IxMutex * (in) — pointer to mutex to initialize

Initializes the given mutex (MUTual EXclusion device)

Can be called from interrupt level — no

#### Returns

- IX_SUCCESS — if the operation succeeded or IX_FAIL otherwise

### A.3.24.5.13 PUBLIC IX_STATUS ixOsServMutexLock (..)

PUBLIC IX_STATUS ixOsServMutexLock (IxMutex * mutex)

locks the given mutex.

#### Parameters

***mutex*** IxMutex * (in) — pointer to mutex to lock

If the mutex is unlocked it becomes locked and owned by the caller, and the function returns immediately. If the mutex is already locked calling this function will suspend the caller until the mutex is unlocked.

Can be called from interrupt level — no

#### Returns

- IX_SUCCESS — if the operation succeeded or IX_FAIL otherwise

***Warning:*** Two or more consecutive calls from the same thread of ixOsServMutexLock() are likely to deadlock the calling thread. Mutexes are NOT recursive.

### A.3.24.5.14 PUBLIC IX_STATUS ixOsServMutexUnlock (..)

PUBLIC IX_STATUS ixOsServMutexUnlock (IxMutex * mutex)

unlocks the given mutex.

#### Parameters

***mutex*** IxMutex * (in) — pointer to mutex to unlock

Unlocks the given mutex, returning it to the “unlocked” state. If the mutex was not locked an error is returned.

Can be called from interrupt level — yes

#### Returns

- IX_SUCCESS — if the operation succeeded or IX_FAIL otherwise

### A.3.24.5.15 PUBLIC void ixOsServSleep (..)

PUBLIC void ixOsServSleep (int microseconds)

execution block for a number of microseconds.

#### Parameters

***microseconds*** int (in) — delay to block execution for

This function blocks the calling task using a timed busy loop.

Can be called from interrupt level — yes

### A.3.24.5.16 PUBLIC void ixOsServTaskSleep (..)

PUBLIC void ixOsServTaskSleep (int milliseconds)

preemptive execution block for a number of milliseconds.

#### Parameters

***milliseconds*** int (in) — delay to block execution for

This function blocks the calling task using an OS-dependent preemptive timed delay.

Can be called from interrupt level — no

### A.3.24.5.17 PUBLIC int ixOsServTimestampGet (void)

used to retrieve the system timestamp.

Can be called from interrupt level — yes

**Returns**

the current timestamp value

## A.3.25 Queue Manager (IxQMgr) API

The public API for the Intel® IXP42X product line QMgr component.

### A.3.25.1 Defines

- #define IX_QMGR_INLINE __inline__ extern
  Inline definition, for inlining of Queue Access functions on API.
- #define IX_QMGR_MAX_NUM_QUEUES 64
  Number of queues supported by the AQM.
- #define IX_QMGR_MIN_QID IX_QMGR_QUEUE_0
  Minimum queue identifier.
- #define IX_QMGR_MAX_QID IX_QMGR_QUEUE_63
  Maximum queue identifier.
- #define IX_QMGR_MIN_QUEUPP_QID 32
  Minimum queue identifier for reduced functionality queues.
- #define IX_QMGR_MAX_QNAME_LEN 16
  Maximum queue name length.
- #define IX_QMGR_WARNING 2
  Warning return code.
- #define IX_QMGR_PARAMETER_ERROR 3
  Parameter error return code (NULL pointer etc.).
- #define IX_QMGR_INVALID_Q_ENTRY_SIZE 4
  Invalid entry size return code.
- #define IX_QMGR_INVALID_Q_ID 5
  Invalid queue identifier return code.
- #define IX_QMGR_INVALID_CB_ID 6
  Invalid callback identifier return code.
- #define IX_QMGR_CB_ALREADY_SET 7
  Callback set error return code.
- #define IX_QMGR_NO_AVALIABLE_SRAM 8
  Sram consumed return code.
- #define IX_QMGR_INVALID_INT_SOURCE_ID 9
  Invalid queue interrupt source identifier return code.
- #define IX_QMGR_INVALID_QSIZE 10
  Invalid queue size error code.

- #define IX_QMGR_INVALID_Q_WM 11
  Invalid queue watermark return code.
- #define IX_QMGR_Q_NOT_CONFIGURED 12
  Queue not configured return code.
- #define IX_QMGR_Q_ALREADY_CONFIGURED 13
  Queue already configured return code.
- #define IX_QMGR_Q_UNDERFLOW 14
  Underflow return code.
- #define IX_QMGR_Q_OVERFLOW 15
  Overflow return code.
- #define IX_QMGR_Q_INVALID_PRIORITY 16
  Invalid priority return code.
- #define IX_QMGR_ENTRY_INDEX_OUT_OF_BOUNDS 17
  Entry index out of bounds return code.

### A.3.25.2 Typedefs

- typedef UINT32 IxQMgrQStatus
  Queue status.
- typedef unsigned IxQMgrCallbackId
  Uniquely identifies a callback function.
- typedef void(* IxQMgrCallback )(IxQMgrQId qId, IxQMgrCallbackId cbId)
  QMgr notification callback type.

### A.3.25.3 Enumerations

- enum IxQMgrQId { IX_QMGR_QUEUE_0 = 0, IX_QMGR_QUEUE_1, IX_QMGR_QUEUE_2, IX_QMGR_QUEUE_3, IX_QMGR_QUEUE_4, IX_QMGR_QUEUE_5, IX_QMGR_QUEUE_6, IX_QMGR_QUEUE_7, IX_QMGR_QUEUE_8, IX_QMGR_QUEUE_9, IX_QMGR_QUEUE_10, IX_QMGR_QUEUE_11, IX_QMGR_QUEUE_12, IX_QMGR_QUEUE_13, IX_QMGR_QUEUE_14, IX_QMGR_QUEUE_15, IX_QMGR_QUEUE_16, IX_QMGR_QUEUE_17, IX_QMGR_QUEUE_18, IX_QMGR_QUEUE_19, IX_QMGR_QUEUE_20, IX_QMGR_QUEUE_21, IX_QMGR_QUEUE_22, IX_QMGR_QUEUE_23, IX_QMGR_QUEUE_24, IX_QMGR_QUEUE_25, IX_QMGR_QUEUE_26, IX_QMGR_QUEUE_27, IX_QMGR_QUEUE_28, IX_QMGR_QUEUE_29, IX_QMGR_QUEUE_30, IX_QMGR_QUEUE_31, IX_QMGR_QUEUE_32, IX_QMGR_QUEUE_33, IX_QMGR_QUEUE_34, IX_QMGR_QUEUE_35, IX_QMGR_QUEUE_36, IX_QMGR_QUEUE_37, IX_QMGR_QUEUE_38, IX_QMGR_QUEUE_39, IX_QMGR_QUEUE_40, IX_QMGR_QUEUE_41, IX_QMGR_QUEUE_42, IX_QMGR_QUEUE_43, IX_QMGR_QUEUE_44, IX_QMGR_QUEUE_45, IX_QMGR_QUEUE_46, IX_QMGR_QUEUE_47, IX_QMGR_QUEUE_48, IX_QMGR_QUEUE_49, IX_QMGR_QUEUE_50, IX_QMGR_QUEUE_51, IX_QMGR_QUEUE_52, IX_QMGR_QUEUE_53, IX_QMGR_QUEUE_54, IX_QMGR_QUEUE_55, IX_QMGR_QUEUE_56, IX_QMGR_QUEUE_57, IX_QMGR_QUEUE_58,

IX_QMGR_QUEUE_59, IX_QMGR_QUEUE_60, IX_QMGR_QUEUE_61, IX_QMGR_QUEUE_62, IX_QMGR_QUEUE_63, IX_QMGR_QUEUE_INVALID }

Generic identifiers for AQM queues.

- enum IxQMgrQStatusMask { IX_QMGR_Q_STATUS_E_BIT_MASK = 0x1, IX_QMGR_Q_STATUS_NE_BIT_MASK = 0x2, IX_QMGR_Q_STATUS_NF_BIT_MASK = 0x4, IX_QMGR_Q_STATUS_F_BIT_MASK = 0x8, IX_QMGR_Q_STATUS_UF_BIT_MASK = 0x10, IX_QMGR_Q_STATUS_OF_BIT_MASK = 0x20 }

  Queue status mask.

- enum IxQMgrSourceId { IX_QMGR_Q_SOURCE_ID_E = 0, IX_QMGR_Q_SOURCE_ID_NE, IX_QMGR_Q_SOURCE_ID_NF, IX_QMGR_Q_SOURCE_ID_F, IX_QMGR_Q_SOURCE_ID_NOT_E, IX_QMGR_Q_SOURCE_ID_NOT_NE, IX_QMGR_Q_SOURCE_ID_NOT_NF, IX_QMGR_Q_SOURCE_ID_NOT_F }

  Queue interrupt source select.

- enum IxQMgrQEntrySizeInWords { IX_QMGR_Q_ENTRY_SIZE1 = 1, IX_QMGR_Q_ENTRY_SIZE2 = 2, IX_QMGR_Q_ENTRY_SIZE4 = 4 }

  QMgr queue entry sizes.

- enum IxQMgrQSizeInWords { IX_QMGR_Q_SIZE16 = 16, IX_QMGR_Q_SIZE32 = 32, IX_QMGR_Q_SIZE64 = 64, IX_QMGR_Q_SIZE128 = 128, IX_QMGR_Q_SIZE_INVALID = 129 }

  QMgr queue sizes.

- enum IxQMgrWMLevel { IX_QMGR_Q_WM_LEVEL0 = 0, IX_QMGR_Q_WM_LEVEL1 = 1, IX_QMGR_Q_WM_LEVEL2 = 2, IX_QMGR_Q_WM_LEVEL4 = 4, IX_QMGR_Q_WM_LEVEL8 = 8, IX_QMGR_Q_WM_LEVEL16 = 16, IX_QMGR_Q_WM_LEVEL32 = 32, IX_QMGR_Q_WM_LEVEL64 = 64 }

  QMgr watermark levels.

- enum IxQMgrDispatchGroup { IX_QMGR_QUELOW_GROUP = 0, IX_QMGR_QUEUPP_GROUP }

  QMgr dispatch group select identifiers.

- enum IxQMgrPriority { IX_QMGR_Q_PRIORITY_0 = 0, IX_QMGR_Q_PRIORITY_1, IX_QMGR_Q_PRIORITY_2, IX_QMGR_Q_PRIORITY_INVALID }

  Dispatcher priority levels.

### A.3.25.4 Functions

- PUBLIC IX_STATUS ixQMgrInit (void)

  Initialize the QMgr.

- PUBLIC void ixQMgrShow (void)

  Describe queue configuration and statistics for active queues.

- PUBLIC IX_STATUS ixQMgrQShow (IxQMgrQId qId)

  Display a queue configuration and statistics for a queue.

- PUBLIC IX_STATUS ixQMgrQConfig (char *qName, IxQMgrQId qId, IxQMgrQSizeInWords qSizeInWords, IxQMgrQEntrySizeInWords qEntrySizeInWords)

  Configure an AQM queue.

- PUBLIC IX_STATUS ixQMgrQSizeInEntriesGet (IxQMgrQId qId, unsigned *qSizeInEntries)

  Return the size of a queue in entries.

- PUBLIC IX_STATUS ixQMgrWatermarkSet (IxQMgrQId qId, IxQMgrWMLevel ne, IxQMgrWMLevel nf)

  Set the Nearly Empty and Nearly Full Watermarks for a queue.

- PUBLIC IX_STATUS ixQMgrAvailableSramAddressGet (UINT32 *address, unsigned *sizeOfFreeSram)

  Return the address of available AQM SRAM.

- PUBLIC IX_STATUS ixQMgrQReadWithChecks (IxQMgrQId qId, UINT32 *entry)

  Read an entry from a queue.

- IX_QMGR_INLINE PUBLIC IX_STATUS ixQMgrQRead (IxQMgrQId qId, UINT32 *entry)

  Fast read of an entry from a queue.

- IX_QMGR_INLINE PUBLIC IX_STATUS ixQMgrQBurstRead (IxQMgrQId qId, UINT32 numEntries, UINT32 *entries)

  Read a number of entries from an AQM queue.

- IX_STATUS ixQMgrQPeek (IxQMgrQId qId, unsigned int entryIndex, UINT32 *entry)

  Read an entry from a queue without moving the read pointer.

- PUBLIC IX_STATUS ixQMgrQWriteWithChecks (IxQMgrQId qId, UINT32 *entry)

  Write an entry to an AQM queue.

- IX_QMGR_INLINE PUBLIC IX_STATUS ixQMgrQWrite (IxQMgrQId qId, UINT32 *entry)

  Fast write of an entry to a queue.

- IX_QMGR_INLINE PUBLIC IX_STATUS ixQMgrQBurstWrite (IxQMgrQId qId, unsigned numEntries, UINT32 *entries)

  Write a number of entries to an AQM queue.

- IX_STATUS ixQMgrQPoke (IxQMgrQId qId, unsigned int entryIndex, UINT32 *entry)

  Write an entry to a queue without moving the write pointer.

- PUBLIC IX_STATUS ixQMgrQNumEntriesGet (IxQMgrQId qId, unsigned *numEntries)

  Get a snapshot of the number of entries in a queue.

- PUBLIC IX_STATUS ixQMgrQStatusGetWithChecks (IxQMgrQId qId, IxQMgrQStatus *qStatus)

  Get a queues status.

- IX_QMGR_INLINE PUBLIC IX_STATUS ixQMgrQStatusGet (IxQMgrQId qId, IxQMgrQStatus *qStatus)

  Fast get of a queue's status.

- PUBLIC IX_STATUS ixQMgrDispatcherPrioritySet (IxQMgrQId qId, IxQMgrPriority priority)

  Set the dispatch priority of a queue.

- PUBLIC IX_STATUS ixQMgrNotificationEnable (IxQMgrQId qId, IxQMgrSourceId sourceId)

  Enable notification on a queue for a specified queue source flag.

- PUBLIC IX_STATUS ixQMgrNotificationDisable (IxQMgrQId qId)
  Disable notifications on a queue.
- PUBLIC void ixQMgrDispatcherLoopRun (IxQMgrDispatchGroup group)
  Run the callback dispatcher.
- PUBLIC IX_STATUS ixQMgrNotificationCallbackSet (IxQMgrQId qId, IxQMgrCallback callback, IxQMgrCallbackId callbackId)
  Set the notification callback for a queue.

### A.3.25.5 Detailed Description

The public API for the Intel® IXP42X product line QMgr component.

IxQMgr is a low level interface to the AHB Queue Manager

### A.3.25.6 Define Documentation

#### A.3.25.6.1 #define IX_QMGR_CB_ALREADY_SET 7

Callback set error return code.

The specified callback has already been for this queue

#### A.3.25.6.2 #define IX_QMGR_ENTRY_INDEX_OUT_OF_BOUNDS 17

Entry index out of bounds return code.

Entry index is greater than number of entries in queue.

#### A.3.25.6.3 #define IX_QMGR_INLINE __inline__ extern

Inline definition, for inlining of Queue Access functions on API.

Please read the header information in this file for more details on the use of function inlining in this component.

#### A.3.25.6.4 #define IX_QMGR_INVALID_CB_ID 6

Invalid callback identifier return code.

Invalid callback id

#### A.3.25.6.5 #define IX_QMGR_INVALID_INT_SOURCE_ID 9

Invalid queue interrupt source identifier return code.

Invalid queue interrupt source given for notification enable

#### A.3.25.6.6 #define IX_QMGR_INVALID_Q_ENTRY_SIZE 4

Invalid entry size return code.

Invalid queue entry size for a queue read/write

#### A.3.25.6.7 #define IX_QMGR_INVALID_Q_ID 5

Invalid queue identifier return code.

Invalid queue id, not in range 0-63

#### A.3.25.6.8 #define IX_QMGR_INVALID_Q_WM 11

Invalid queue watermark return code.

Invalid queue watermark given for watermark set

#### A.3.25.6.9 #define IX_QMGR_INVALID_QSIZE 10

Invalid queue size error code.

Invalid queue size not one of 16,32, 64, 128

#### A.3.25.6.10 #define IX_QMGR_MAX_NUM_QUEUES 64

Number of queues supported by the AQM.

This constant is used to indicate the number of AQM queues

#### A.3.25.6.11 #define IX_QMGR_MAX_QID IX_QMGR_QUEUE_63

Maximum queue identifier.

This constant is used to indicate the largest queue identifier

#### A.3.25.6.12 #define IX_QMGR_MAX_QNAME_LEN 16

Maximum queue name length.

This constant is used to indicate the maximum null terminated string length (excluding '\0') for a queue name

#### A.3.25.6.13 #define IX_QMGR_MIN_QID IX_QMGR_QUEUE_0

Minimum queue identifier.

This constant is used to indicate the smallest queue identifier

#### A.3.25.6.14 #define IX_QMGR_MIN_QUEUPP_QID 32

Minimum queue identifier for reduced functionality queues.

This constant is used to indicate Minimum queue identifier for reduced functionality queues

#### A.3.25.6.15 #define IX_QMGR_NO_AVALIABLE_SRAM 8

Sram consumed return code.

All AQM Sram is consumed by queue configuration

#### A.3.25.6.16 #define IX_QMGR_PARAMETER_ERROR 3

Parameter error return code (NULL pointer etc.).

parameter error out of range/invalid

#### A.3.25.6.17 #define IX_QMGR_Q_ALREADY_CONFIGURED 13

Queue already configured return code.

Returned to client to indicate that a queue has already been configured

#### A.3.25.6.18 #define IX_QMGR_Q_INVALID_PRIORITY 16

Invalid priority return code.

Invalid priority, not one of 0,1,2

#### A.3.25.6.19 #define IX_QMGR_Q_NOT_CONFIGURED 12

Queue not configured return code.

Returned to the client when a function has been called on an unconfigured queue.

#### A.3.25.6.20 #define IX_QMGR_Q_OVERFLOW 15

Overflow return code.

Overflow on a queue write has occurred

#### A.3.25.6.21 #define IX_QMGR_Q_UNDERFLOW 14

Underflow return code.

Underflow on a queue read has occurred

#### A.3.25.6.22 #define IX_QMGR_WARNING 2

Warning return code.

Execution complete, but there is a special case to handle

### A.3.25.7 Typedef Documentation

#### A.3.25.7.1 IxQMgrCallback

QMgr notification callback type.

This defines the interface to all client callback functions.

**Parameters**

***IxQMgrQId*** qId(in) — the queue identifier

***IxQMgrCallbackId*** cbId(in) — the callback identifier

#### A.3.25.7.2 IxQMgrCallbackId

Uniquely identifies a callback function.

A unique callback identifier associated with each callback registered by clients.

#### A.3.25.7.3 typedef UINT32 IxQMgrQStatus

Queue status.

A queues status is defined by its relative fullness or relative emptiness. Each of the queues 0-31 have Nearly Empty, Nearly Full, Empty, Full, Underflow and Overflow status flags. Queues 32-63 have just Nearly Empty and Full status flags. The flags bit positions are outlined below

```
OF UF F NF NE E ---------------------- 5 | 4 | 3 | 2 | 1 | 0
```

### A.3.25.8 Enumeration Type Documentation

#### A.3.25.8.1 enum IxQMgrDispatchGroup

QMgr dispatch group select identifiers.

This enum defines the groups over which the dispatcher will process when called. One of the enum values must be used as a input to ixQMgrDispatcherLoopRun().

**Enumeration values**

- ***IX_QMGR_QUELOW_GROUP*** — Queues 0-31.
- ***IX_QMGR_QUEUPP_GROUP*** — Queues 32-63.

#### A.3.25.8.2 enum IxQMgrPriority

Dispatcher priority levels.

This enum defines the different queue dispatch priority levels. The lowest priority number (0) is the highest priority level.

**Enumeration values**

- ***IX_QMGR_Q_PRIORITY_0*** — Priority level 0.
- ***IX_QMGR_Q_PRIORITY_1*** — Priority level 1.
- ***IX_QMGR_Q_PRIORITY_2*** — Priority level 2.
- ***IX_QMGR_Q_PRIORITY_INVALID*** — Invalid Priority level.

#### A.3.25.8.3 enum IxQMgrQEntrySizeInWords

QMgr queue entry sizes.

The entry size of a queue specifies the size of a queues entry in words.

**Enumeration values**

- ***IX_QMGR_Q_ENTRY_SIZE1*** — 1 word entry.
- ***IX_QMGR_Q_ENTRY_SIZE2*** — 2 word entry.
- ***IX_QMGR_Q_ENTRY_SIZE4*** — 4 word entry.

#### A.3.25.8.4 enum IxQMgrQId

Generic identifiers for AQM queues.

These enum values are used in the AQM queue config header file. This enum defines generic identifiers for the queues in that the application of the queue is not specified. The connection between queue number and queue application is done in the AQM queue config header file. Clients of this component should NOT use these values to identify queues, the defines in the AQM queue configuration header should be used.

#### A.3.25.8.5 enum IxQMgrQSizeInWords

QMgr queue sizes.

These values define the allowed queue sizes for AQM queue. The sizes are specified in words.

**Enumeration values**

- ***IX_QMGR_Q_SIZE16*** — 16 word buffer.
- ***IX_QMGR_Q_SIZE32*** — 32 word buffer.
- ***IX_QMGR_Q_SIZE64*** — 64 word buffer.
- ***IX_QMGR_Q_SIZE128*** — 128 word buffer.
- ***IX_QMGR_Q_SIZE_INVALID*** — Ensures that this is greater than largest queue size supported by the hardware.

#### A.3.25.8.6 enum IxQMgrQStatusMask

Queue status mask.

Masks for extracting the individual status flags from the IxQMgrStatus word.

#### A.3.25.8.7 enum IxQMgrSourceId

Queue interrupt source select.

This enum defines the different source conditions on a queue that result in an interrupt being fired by the AQM. Interrupt source is configurable for queues 0-31 only. The interrupt source for queues 32-63 is hardwired to the NE(Nearly Empty) status flag.

#### Enumeration values

- ***IX_QMGR_Q_SOURCE_ID_E*** — Queue Empty due to last read.
- ***IX_QMGR_Q_SOURCE_ID_NE*** — Queue Nearly Empty due to last read.
- ***IX_QMGR_Q_SOURCE_ID_NF*** — Queue Nearly Full due to last write.
- ***IX_QMGR_Q_SOURCE_ID_F*** — Queue Full due to last write.
- ***IX_QMGR_Q_SOURCE_ID_NOT_E*** — Queue Not Empty due to last write.
- ***IX_QMGR_Q_SOURCE_ID_NOT_NE*** — Queue Not Nearly Empty due to last write.
- ***IX_QMGR_Q_SOURCE_ID_NOT_NF*** — Queue Not Nearly Full due to last read.
- ***IX_QMGR_Q_SOURCE_ID_NOT_F*** — Queue Not Full due to last read.

### A.3.25.8.8 enum IxQMgrWMLevel

QMgr watermark levels.

These values define the valid watermark levels(in ENTRIES) for queues. Each queue 0-63 have configurable Nearly full and Nearly empty watermarks. For queues 32-63 the Nearly full watermark has NO EFFECT. If the Nearly full watermark is set to IX_QMGR_Q_WM_LEVEL16 this means that the nearly full flag will be set by the hardware when there are >= 16 empty entries in the specified queue. If the Nearly empty watermark is set to IX_QMGR_Q_WM_LEVEL16 this means that the Nearly empty flag will be set by the hardware when there are <= 16 full entries in the specified queue.

#### Enumeration values

- ***IX_QMGR_Q_WM_LEVEL0*** — 0 entry watermark.
- ***IX_QMGR_Q_WM_LEVEL1*** — 1 entry watermark.
- ***IX_QMGR_Q_WM_LEVEL2*** — 2 entry watermark.
- ***IX_QMGR_Q_WM_LEVEL4*** — 4 entry watermark.
- ***IX_QMGR_Q_WM_LEVEL8*** — 8 entry watermark.
- ***IX_QMGR_Q_WM_LEVEL16*** — 16 entry watermark.
- ***IX_QMGR_Q_WM_LEVEL32*** — 32 entry watermark.
- ***IX_QMGR_Q_WM_LEVEL64*** — 64 entry watermark.

## A.3.25.9 Function Documentation

### A.3.25.9.1 ixQMgrAvailableSramAddressGet (..)

ixQMgrAvailableSramAddressGet (UINT32 * address, unsigned * sizeOfFreeSram)

Return the address of available AQM SRAM.

This function returns the starting address in AQM SRAM not used by the current queue configuration and should only be called after all queues have been configured. Calling this function before all queues have been configured will return the currently available SRAM. A call to configure another queue will use some of the available SRAM. The amount of SRAM available is specified in sizeOfFreeSram. The address is the address of the bottom of available SRAM. Available SRAM extends from address from address to address + sizeOfFreeSram.

#### Parameters

***UINT32*** (out) **address — the address of the available SRAM, NULL if none available.

***unsigned*** (out) *sizeOfFreeSram — the size in words of available SRAM

#### Returns

- IX_SUCCESS — there is available SRAM and is pointed to by address
- IX_QMGR_PARAMETER_ERROR — invalid parameter(s)
- IX_QMGR_NO_AVALIABLE_SRAM — all AQM SRAM is consumed by the queue configuration.

### A.3.25.9.2 ixQMgrDispatcherLoopRun (..)

ixQMgrDispatcherLoopRun (IxQMgrDispatchGroup group)

Run the callback dispatcher.

The function runs the dispatcher for a group of queues. Callbacks are made for interrupts that have occurred on queues within the group that have registered callbacks. The order in which queues are serviced depends on the queue priorities set by the client. This function may be called from interrupt or task context.

This function is not re-entrant.

#### Parameters

***IxQMgrDispatchGroup*** (in) group — the group of queues over which the dispatcher will run

#### Returns

void

### A.3.25.9.3 ixQMgrDispatcherPrioritySet (..)

ixQMgrDispatcherPrioritySet (IxQMgrQId qId, IxQMgrPriority priority)

Set the dispatch priority of a queue.

This function is called to set the dispatch priority of queue. The effect of this function is to add a priority change request to a queue. This queue is serviced by *ixQMgrDispatcherLoopRun*.

This function is re-entrant. and can be used from an interrupt context

#### Parameters

***IxQMgrQId*** (in) qId — the queue identifier

***IxQMgrPriority*** (in) priority — the new queue dispatch priority

#### Returns

- IX_SUCCESS — priority change request is queued
- IX_QMGR_Q_NOT_CONFIGURED — the specified qId has not been configured

- IX_QMGR_Q_INVALID_PRIORITY — specified priority is invalid

#### A.3.25.9.4 ixQMgrInit (void)

Initialize the QMgr.

This function must be called before and other QMgr function. It sets up internal data structures.

**Returns**

- IX_SUCCESS — the IxQMgr successfully initialized
- IX_FAIL — failed to initialize the Qmgr

#### A.3.25.9.5 ixQMgrNotificationCallbackSet (..)

ixQMgrNotificationCallbackSet (IxQMgrQId qId, IxQMgrCallback callback, IxQMgrCallbackId callbackId

Set the notification callback for a queue.

This function sets the callback for the specified queue. This callback will be called by the dispatcher, and may be called in the context of a interrupt If callback has a value of NULL the previously registered callback, if one exists will be unregistered.

**Parameters**

***IxQMgrQId*** (in) qId — queue identifier

***IxQMgrCallback*** (in) callback — the callback registered for this queue

***IxQMgrCallbackId*** (in) callbackid — the callback identifier

**Returns**

- IX_SUCCESS — the callback for the specified queue has been set
- IX_QMGR_Q_NOT_CONFIGURED — the specified qId has not been configured

#### A.3.25.9.6 ixQMgrNotificationDisable (..)

ixQMgrNotificationDisable (IxQMgrQId qId)

Disable notifications on a queue.

This function is called to disable notifications on a specified queue.

This function is re-entrant. and can be used from an interrupt context

**Parameters**

***IxQMgrQId*** (in) qId — the queue identifier

**Returns**

- IX_SUCCESS — the interrupt has been disabled
- IX_QMGR_Q_NOT_CONFIGURED — the specified qId has not been configured

### A.3.25.9.7 ixQMgrNotificationEnable (..)

ixQMgrNotificationEnable (IxQMgrQId qId, IxQMgrSourceId sourceId)

Enable notification on a queue for a specified queue source flag.

This function is called by a client of the QMgr to enable notifications on a specified condition. If the condition for the notification is set after the client has called this function but before the function has enabled the interrupt source, then the notification will not occur. For queues 32-63 the notification source is fixed to the NE(Nearly Empty) flag and cannot be changed so the sourceId parameter is ignored for these queues. The status register is read before the notification is enabled and is read again after the notification has been enabled, if they differ then the warning status is returned.

This function is re-entrant. and can be used from an interrupt context

#### Parameters

***IxQMgrQId*** (in) qId — the queue identifier

***IxQMgrSourceId*** (in) sourceId — the interrupt src condition identifier

#### Returns

- IX_SUCCESS — the interrupt has been enabled for the specified source
- IX_QMGR_Q_NOT_CONFIGURED — the specified qId has not been configured
- IX_QMGR_INVALID_INT_SOURCE_ID — interrupt source invalid for this queue
- IX_QMGR_WARNING — the status register may not be consistent

### A.3.25.9.8 ixQMgrQBurstRead (..)

ixQMgrQBurstRead (IxQMgrQId qId, UINT32 numEntries, UINT32 * entries)

Read a number of entries from an AQM queue.

This function will burst read a number of entries from the specified queue. The entry size of queue is auto-detected. The function will attempt to read as many entries as specified by the numEntries parameter and will return an UNDERFLOW if any one of the individual entry reads fail.

***Warning:*** IX_QMGR_Q_UNDERFLOW is only returned for queues 0-31 as queues 32-63 do not have an underflow status maintained, hence there is a potential for silent failure here. This function must be used with caution.

#### Note

This function is intended for fast draining of queues, so to make it as efficient as possible, it has the following features:

- This function is inlined, to reduce unnecessary function call overhead.
- It does not perform any parameter checks, or update any statistics.
- It does not check that the queue specified by qId has been configured.
- It does not check that the queue has the number of full entries that have been specified to be read. It will read until it finds a NULL entry or until the number of specified entries have been

read. It always checks for underflow after all the reads have been performed. Therefore, the client should ensure before calling this function that there are enough entries in the queue to read. ixQMgrQNumEntriesGet() will provide the number of full entries in a queue. ixQMgrQRead() or ixQMgrQReadWithChecks(), which only reads a single queue entry per call, should be used instead if the user requires checks for UNDERFLOW after each entry read.

#### Parameters

***IxQMgrQId*** (in) qId — the queue identifier.

***unsigned*** (in) numEntries — the number of entries to write.

***UINT32*** (out) *entries — the word(s) read.

#### Returns

- IX_SUCCESS — entries were successfully read.
- IX_QMGR_Q_UNDERFLOW — attempt to read from an empty queue

### A.3.25.9.9 ixQMgrQBurstWrite (..)

ixQMgrQBurstWrite (IxQMgrQId qId, unsigned numEntries, UINT32 * entries)

Write a number of entries to an AQM queue.

This function will burst write a number of entries to the specified queue. The entry size of queue is auto-detected. The function will attempt to write as many entries as specified by the numEntries parameter and will return an OVERFLOW if any one of the individual entry writes fail.

***Warning:*** IX_QMGR_Q_OVERFLOW is only returned for queues 0-31 as queues 32-63 do not have an overflow status maintained, hence there is a potential for silent failure here. This function must be used with caution.

#### Notes

This function is intended for fast population of queues, so to make it as efficient as possible, it has the following features:

- This function is inlined, to reduce unnecessary function call overhead.
- It does not perform any parameter checks, or update any statistics.
- It does not check that the queue specified by qId has been configured.
- It does not check that the queue has enough free space to hold the entries before writing, and only checks for overflow after all writes have been performed. Therefore, the client should ensure before calling this function that there is enough free space in the queue to hold the number of entries to be written. ixQMgrQWrite() or ixQMgrQWriteWithChecks(), which only writes a single queue entry per call, should be used instead if the user requires checks for OVERFLOW after each entry written.

#### Parameters

***IxQMgrQId*** (in) qId — the queue identifier.

***unsigned*** (in) numEntries — the number of entries to write.

***UINT32*** (in) *entries — the word(s) to write.

#### Returns

- IX_SUCCESS — value was successfully written.
- IX_QMGR_Q_OVERFLOW — attempt to write to a full queue

### A.3.25.9.10 ixQMgrQConfig (..)

ixQMgrQConfig (char * qName, IxQMgrQId qId, IxQMgrQSizeInWords qSizeInWords, IxQMgrQEntrySizeInWords qEntrySizeInWords)

Configure an AQM queue.

This function is called by a client to setup a queue. The size and entrySize qId and qName(NULL pointer) are checked for valid values. This function must be called for each queue, before any queue accesses are made and after ixQMgrInit() has been called. qName is assumed to be a '\0' terminated array of 16 characters or less.

#### Parameters

***char*** (in) *qName — is the name provided by the client and is associated with a QId by the QMgr.

***IxQMgrQId*** (in) qId — the qId of this queue

***IxQMgrQSize*** (in) qSizeInWords — the size of the queue can be one of 16,32 64, 128 words.

***IxQMgrQEntrySizeInWords*** (in) qEntrySizeInWords — the size of a queue entry can be one of 1,2,4 words.

#### Returns

- IX_SUCCESS — a specified queue has been successfully configured.
- IX_QMGR_PARAMETER_ERROR — invalid parameter(s).
- IX_QMGR_INVALID_QSIZE — invalid queue size
- IX_QMGR_INVALID_Q_ID — invalid queue id
- IX_QMGR_INVALID_Q_ENTRY_SIZE — invalid queue entry size
- IX_QMGR_Q_ALREADY_CONFIGURED — queue already configured

### A.3.25.9.11 ixQMgrQNumEntriesGet (..)

ixQMgrQNumEntriesGet (IxQMgrQId qId, unsigned * numEntries)

Get a snapshot of the number of entries in a queue.

This function gets the number of entries in a queue.

#### Parameters

***IxQMgrQId*** (in) qId — queue identifier

***unsigned*** (out) *numEntries — number of entries in a queue

### Returns

- IX_SUCCESS — got the number of entries for the queue
- IX_QMGR_PARAMETER_ERROR — invalid parameter(s).
- IX_QMGR_Q_NOT_CONFIGURED — the specified qId has not been configured
- IX_QMGR_WARNING — could not determine num entries at this time

## A.3.25.9.12 ixQMgrQPeek (..)

ixQMgrQPeek (IxQMgrQId qId, unsigned int entryIndex, UINT32 * entry)

Read an entry from a queue without moving the read pointer.

This function inspects an entry in a queue. The entry is inspected directly in AQM SRAM and is not read from queue access registers. The entry is NOT removed from the queue and the read/write pointers are unchanged. The queue should not be accessed when this function is called.

### Parameters

***IxQMgrQId*** (in) qId — the queue identifier.

***unsigned*** int(in) entryIndex — index of entry in queue in the range [0].....[current number of entries in queue].

***UINT32*** (out) *entry — pointer to the entry word(s).

### Returns

- IX_SUCCESS — entry was successfully inspected.
- IX_QMGR_PARAMETER_ERROR — invalid parameter(s).
- IX_QMGR_Q_NOT_CONFIGURED — queue not configured for this QId.
- IX_QMGR_ENTRY_INDEX_OUT_OF_BOUNDS — an entry does not exist at specified index.
- IX_FAIL — failed to inspect the queue entry.

## A.3.25.9.13 ixQMgrQPoke (..)

ixQMgrQPoke (IxQMgrQId qId, unsigned int entryIndex, UINT32 * entry)

Write an entry to a queue without moving the write pointer.

This function modifies an entry in a queue. The entry is modified directly in AQM SRAM and not using the queue access registers. The entry is NOT added to the queue and the read/write pointers are unchanged. The queue should not be accessed when this function is called.

### Parameters

***IxQMgrQId*** (in) qId — the queue identifier.

***unsigned*** int(in) entryIndex — index of entry in queue in the range [0].....[current number of entries in queue].

***UINT32*** (in) *entry — pointer to the entry word(s).

### Returns

- IX_SUCCESS — entry was successfully modified.
- IX_QMGR_PARAMETER_ERROR — invalid parameter(s).
- IX_QMGR_Q_NOT_CONFIGURED — queue not configured for this QId.
- IX_QMGR_ENTRY_INDEX_OUT_OF_BOUNDS — an entry does not exist at specified index.
- IX_FAIL — failed to modify the queue entry.

## A.3.25.9.14 ixQMgrQRead (..)

ixQMgrQRead (IxQMgrQId qId, UINT32 * entry)

Fast read of an entry from a queue.

This function is a heavily streamlined version of ixQMgrQReadWithChecks(), but performs essentially the same task. It reads an entire entry from a queue, returning it in entry which must be a pointer to a previously allocated array of sufficient size to hold an entry.

### Notes

- This function is inlined, to reduce unnecessary function call overhead. It does not perform any parameter checks, or update any statistics. Also, it does not check that the queue specified by qId has been configured. or is in range. It simply reads an entry from the queue, and checks for underflow.
- IX_QMGR_Q_UNDERFLOW is only returned for queues 0-31 as queues 32-63 do not have an underflow status maintained.

### Parameters

***IxQMgrQId*** (in) qId — the queue identifier.

***UINT32*** (out) *entry — pointer to the entry word(s).

### Returns

- IX_SUCCESS — entry was successfully read.
- IX_QMGR_Q_UNDERFLOW — attempt to read from an empty queue

## A.3.25.9.15 ixQMgrQReadWithChecks (..)

ixQMgrQReadWithChecks (IxQMgrQId qId, UINT32 * entry)

Read an entry from a queue.

This function reads an entire entry from a queue returning it in entry. The queue configuration word is read to determine what entry size this queue is configured for and then the number of words specified by the entry size is read. entry must be a pointer to a previously allocated array of sufficient size to hold an entry.

### Note

- IX_QMGR_Q_UNDERFLOW — Only returned for queues 0-31 as queues 32-63 do not have an underflow status maintained.

#### Parameters

***IxQMgrQId*** (in) qId — the queue identifier.

***UINT32*** (out) *entry — pointer to the entry word(s).

#### Returns

- IX_SUCCESS — entry was successfully read.
- IX_QMGR_PARAMETER_ERROR — invalid parameter(s).
- IX_QMGR_Q_NOT_CONFIGURED — queue not configured for this QId
- IX_QMGR_Q_UNDERFLOW — attempt to read from an empty queue

### A.3.25.9.16 ixQMgrQShow (..)

ixQMgrQShow (IxQMgrQId qId)

Display queue configuration and statistics for a queue.

This function shows queue configuration and statistics for a queue.

#### Parameters

***IxQMgrQIdin*** (qId) — the queue identifier.

#### Returns

- IX_SUCCESS — success
- IX_QMGR_Q_NOT_CONFIGURED — queue not configured for this QId

### A.3.25.9.17 ixQMgrQSizeInEntriesGet (..)

ixQMgrQSizeInEntriesGet (IxQMgrQId qId, unsigned * qSizeInEntries)

Return the size of a queue in entries.

This function returns the size of the queue in entries.

#### Parameters

***IxQMgrQId*** (in) qId — the queue identifier

***IxQMgrQSize*** (out) *qSizeInEntries — queue size in entries

#### Returns

- IX_SUCCESS — successfully retrieved the number of full entries.
- IX_QMGR_Q_NOT_CONFIGURED — queue not configured for this QId.
- IX_QMGR_PARAMETER_ERROR — invalid parameter(s).

### A.3.25.9.18 ixQMgrQStatusGet (..)

ixQMgrQStatusGet (IxQMgrQId qId, IxQMgrQStatus * qStatus)

Fast get of a queue's status.

This function is a streamlined version of ixQMgrQStatusGetWithChecks(), but performs essentially the same task. It reads the specified queue's status. A queues status is defined by its status flags. For queues 0-31 these flags are E,NE,NF,F,UF,OF. For queues 32-63 these flags are NE and F.

#### Note

This function is inlined, to reduce unnecessary function call overhead. It does not perform any parameter checks, or update any statistics. Also, it does not check that the queue specified by qId has been configured. It simply reads the specified queue's status.

#### Parameters

***IxQMgrQId*** (in) qId — the queue identifier.

***IxQMgrQStatus*** (out) *qStatus — the status of the specified queue.

#### Returns

void.

### A.3.25.9.19 ixQMgrQStatusGetWithChecks (..)

ixQMgrQStatusGetWithChecks (IxQMgrQId qId, IxQMgrQStatus * qStatus)

Get a queues status.

This function reads the specified queues status. A queues status is defined by its status flags. For queues 0-31 these flags are E,NE,NF,F,UF,OF. For queues 32-63 these flags are NE and F.

#### Parameters

***IxQMgrQId*** (in) qId — the queue identifier.

***IxQMgrQStatus*** (out) *qStatus — the status of the specified queue.

#### Returns

- IX_SUCCESS — queue status was successfully read.
- IX_QMGR_Q_NOT_CONFIGURED — specified qId has not been configured.
- IX_QMGR_PARAMETER_ERROR — invalid parameter.

### A.3.25.9.20 ixQMgrQWrite (..)

ixQMgrQWrite (IxQMgrQId qId, UINT32 * entry)

Fast write of an entry to a queue.

This function is a heavily streamlined version of ixQMgrQWriteWithChecks(), but performs essentially the same task. It will write the entry size number of words pointed to by entry to the queue specified by qId.

#### Notes

This function is inlined, to reduce unnecessary function call overhead. It does not perform any parameter checks, or update any statistics. Also, it does not check that the queue specified by qId has been configured. It simply writes an entry to the queue, and checks for overflow.

IX_QMGR_Q_OVERFLOW is only returned for queues 0-31 as queues 32-63 do not have an overflow status maintained.

#### Parameters

***IxQMgrQId*** (in) qId — the queue identifier.

***UINT32*** (in) *entry — pointer to the entry word(s).

#### Returns

- IX_SUCCESS — entry was successfully read.
- IX_QMGR_Q_OVERFLOW — attempt to write to a full queue

### A.3.25.9.21 ixQMgrQWriteWithChecks (..)

ixQMgrQWriteWithChecks (IxQMgrQId qId, UINT32 * entry)

Write an entry to an AQM queue.

This function will write the entry size number of words pointed to by entry to the queue specified by qId. The queue configuration word is read to determine the entry size of queue and the corresponding number of words is then written to the queue.

#### Note

IX_QMGR_Q_OVERFLOW is only returned for queues 0-31 as queues 32-63 do not have an overflow status maintained.

#### Parameters

***IxQMgrQId*** (in) qId — the queue identifier.

***UINT32*** (in) *entry — the word(s) to write.

#### Returns

- IX_SUCCESS — value was successfully written.
- IX_QMGR_PARAMETER_ERROR — invalid parameter(s).
- IX_QMGR_Q_NOT_CONFIGURED — queue not configured for this QId
- IX_QMGR_Q_OVERFLOW — attempt to write to a full queue

### A.3.25.9.22 ixQMgrShow (void)

Describe queue configuration and statistics for active queues.

This function shows active queues, their configurations and statistics.

#### Returns

void

### A.3.25.9.23 ixQMgrWatermarkSet (..)

ixQMgrWatermarkSet (IxQMgrQId qId, IxQMgrWMLevel ne, IxQMgrWMLevel nf)

Set the Nearly Empty and Nearly Full Watermarks for a queue.

This function is called by a client to set the watermarks NE and NF for the queue specified by qId. The queue must be empty at the time this function is called, it is the clients responsibility to ensure that the queue is empty. This function will read the status of the queue before the watermarks are set and again after the watermarks are set. If the status register has changed, due to a queue access by an NPE for example, a warning is returned. Queues 32-63 only support the NE flag, therefore the value of nf will be ignored for these queues.

#### Parameters

***IxQMgrQId*** (in) qId — the QId of the queue.

***IxQMgrWMLevel*** (in) ne — the NE(Nearly Empty) watermark for this queue. Valid values are 0,1,2,4,8,16,32 and 64 entries.

***IxQMgrWMLevel*** (in) nf — the NF(Nearly Full) watermark for this queue. Valid values are 0,1,2,4,8,16,32 and 64 entries.

#### Returns

- IX_SUCCESS — watermarks have been set for the queue.
- IX_QMGR_Q_NOT_CONFIGURED — queue not configured for this QId.
- IX_QMGR_INVALID_Q_WM — invalid watermark.
- IX_QMGR_WARNING — status register may not be consistent.

## A.3.26 Timer Control (IxTimerCtrl) API

The public API for the Intel® IXP42X product line Timer Control Component.

### A.3.26.1 Defines

- #define IX_TIMERCTRL_NO_FREE_TIMERS 2
  Timer schedule return code.
- #define IX_TIMERCTRL_PARAM_ERROR 3
  Timer schedule return code.

### A.3.26.2 Typedefs

- typedef void(* IxTimerCtrlTimerCallback )(void *userParam)
  A typedef for a pointer to a timer callback function.

### A.3.26.3 Enumerations

- enum IxTimerCtrlPurpose { IxTimerCtrlAdslPurpose, IxTimerCtrlMaxPurpose }
  List used to identify the users of timers.

### A.3.26.4 Functions

- IX_STATUS ixTimerCtrlSchedule (IxTimerCtrlTimerCallback func, void *userParam, IxTimerCtrlPurpose purpose, UINT32 time, unsigned *timerId)
  Schedules a callback function to be called after a period of “time.” The callback function should not block or run for more than 100ms. This function.
- IX_STATUS ixTimerCtrlScheduleRepeating (IxTimerCtrlTimerCallback func, void *param, IxTimerCtrlPurpose purpose, UINT32 interval, unsigned *timerId)
  Schedules a callback function to be called after a period of “time.” The callback function should not block or run for more than 100ms.
- IX_STATUS ixTimerCtrlCancel (unsigned id)
  Cancels a scheduled callback.
- IX_STATUS ixTimerCtrlInit (void)
  Initialize the Timer Control Component.
- void ixTimerCtrlShow (void)
  Display the status of the Timer Control Component.

### A.3.26.5 Detailed Description

The public API for the Intel® IXP42X product line Timer Control Component.

### A.3.26.6 Define Documentation

#### A.3.26.6.1 #define IX_TIMERCTRL_NO_FREE_TIMERS 2

Timer schedule return code.

Indicates that the request to start a timer failed because all available timer resources are used.

#### A.3.26.6.2 #define IX_TIMERCTRL_PARAM_ERROR 3

Timer schedule return code.

Indicates that the request to start a timer failed because the client has supplied invalid parameters.

### A.3.26.7 Typedef Documentation

#### A.3.26.7.1 typedef void(..)

typedef void(* IxTimerCtrlTimerCallback)(void *userParam)

a void * - This parameter is supplied by the client when the timer is started and passed back to the client in the callback.

**Note**

In general timer callback functions should not block or take longer than 100ms. This constraint is required to ensure that higher priority callbacks are not held up. All callbacks are called from the same thread. This thread is a shared resource. The parameter passed is provided when the timer is scheduled.

## A.3.26.8 Enumeration Type Documentation

### A.3.26.8.1 enum IxTimerCtrlPurpose

List used to identify the users of timers.

**Note**

The order in this list indicates priority. Components appearing higher in the list will be given priority over components lower in the list. When adding components, please insert at an appropriate position for priority (i.e values should be less than IxTimerCtrlMaxPurpose).

## A.3.26.9 Function Documentation

### A.3.26.9.1 ixTimerCtrlCancel (..)

ixTimerCtrlCancel (unsigned id)

Cancels a scheduled callback.

**Parameters**

***id*** — the id of the callback to be cancelled.

**Returns**

- IX_SUCCESS — Timer was successfully stopped.
- IX_FAIL — id parameter did not correspond to any running timer.

**Note**

This function is re-entrant. The function accesses a list of running timers and may suspend the calling thread if this list is being accessed by another thread.

### A.3.26.9.2 ixTimerCtrlInit (void)

Initialize the Timer Control Component.

**Returns**

- IX_SUCCESS — The timer control component initialized successfully.
- IX_FAIL — The timer control component initialization failed, or the component was already initialized.

**Note**

This must be done before any other API function is called. This function should be called once only and is not re-entrant.

### A.3.26.9.3 ixTimerCtrlSchedule (..)

ixTimerCtrlSchedule (IxTimerCtrlTimerCallback func, void * userParam, IxTimerCtrlPurpose purpose, UINT32 time, unsigned * timerId)

Schedules a callback function to be called after a period of"time." The callback function should not block or run for more than 100ms. This function.

#### Parameters

***func*** (in) — the callback function to be called.

***userParam*** (in) — a parameter to send to the callback function, can be NULL.

***purpose*** (in) — the purpose of the callback, internally this component will decide the priority of callbacks with different purpose.

***time*** (in) — time relative to now in milliseconds after which the callback will be called. The time must be greater than the duration of one OS tick.

***timerId*** (out) — An id for the callback scheduled. This id can be used to cancel the callback.

#### Returns

- IX_SUCCESS — The timer was started successfully.
- IX_TIMERCTRL_NO_FREE_TIMERS — The timer was not started because the maximum number of running timers has been exceeded.
- IX_TIMERCTRL_PARAM_ERROR — The timer was not started because the client has supplied a NULL callback func, or the requested timeout is less than one OS tick.

#### Note

This function is re-entrant. The function accesses a list of running timers and may suspend the calling thread if this list is being accessed by another thread.

### A.3.26.9.4 ixTimerCtrlScheduleRepeating (..)

ixTimerCtrlScheduleRepeating (IxTimerCtrlTimerCallback func, void * param, IxTimerCtrlPurpose purpose, UINT32 interval, unsigned * timerId)

Schedules a callback function to be called after a period of "time." The callback function should not block or run for more than 100ms.

#### Parameters

***func*** (in) — callback function to be called.

***userParam*** (in) — parameter to send to the callback function, can be NULL.

***purpose*** (in) — purpose of the callback, internally this component will decide the priority of callbacks with different purpose.

***interval*** — interval in milliseconds between calls to func.

***timerId*** (out) — id for the callback scheduled. This id can be used to cancel the callback.

#### Returns

- IX_SUCCESS — The timer was started successfully.
- IX_TIMERCTRL_NO_FREE_TIMERS — The timer was not started because the maximum number of running timers has been exceeded.
- IX_TIMERCTRL_PARAM_ERROR — The timer was not started because the client has supplied a NULL callback func, or the requested timeout is less than one OS tick.

#### Note

This function is re-entrant. The function accesses a list of running timers and may suspend the calling thread if this list is being accessed by another thread.

### A.3.26.9.5 ixTimerCtrlShow (void)

Display the status of the Timer Control Component.

#### Note

Displays a list of running timers. This function is not re-entrant. This function does not suspend the calling thread.

## A.3.27 Types (IxTypes)

Basic data types used by the Intel® IXP42X product line project.

### A.3.27.1 Defines

- #define OK  0
- #define ERROR  (-1)
- #define NELEMENTS(array)  (sizeof(array) / sizeof((array)[0]))
- #define MCLBYTES  2048
- #define BZERO(buf)  memset(&(buf), 0, sizeof(buf))
- #define IX_SUCCESS  0
  Standard return values.
- #define IX_FAIL  1
- #define TRUE  1
  Boolean TRUE and FALSE definitions.
- #define FALSE  0
- #define NULL  0L
  definition of NULL.
- #define PRIVATE

### A.3.27.2 Typedefs

- typedef void(* IxVoidFnPtr )()
- typedef int(* FUNCPTR )()
- typedef int STATUS
- typedef int BOOL
- typedef unsigned char UCHAR
- typedef unsigned short USHORT
- typedef unsigned int UINT
- typedef unsigned long ULONG
- typedef char INT8
- typedef short INT16
- typedef int INT32
- typedef unsigned char UINT8
- typedef unsigned short UINT16
- typedef unsigned int UINT32
- typedef void VOID
- typedef volatile UINT32 VUINT32
- typedef volatile INT32 VINT32
- typedef UINT32 IX_STATUS

### A.3.27.3 Detailed Description

Basic data types used by the Intel® IXP42X product line project.

## A.3.28 UART Access (IxUARTAcc) API

Intel® IXP42X product line UARTAcc Driver Public API.

### A.3.28.1 Data Structures

- struct _ixUARTDev
- struct _ixUARTStats
- struct ixUARTDev
  Device descriptor for the UART.
- struct ixUARTStats
  Statistics for the UART.

### A.3.28.2 Defines

- #define IX_UART_DEF_OPTS (CLOCAL | CS8)
  The default hardware options to set the UART to - no flow control, 8 bit word, 1 stop bit, no parity.
- #define IX_UART_DEF_XMIT 64
  The default UART FIFO size - must be no bigger than 64.
- #define IX_UART_DEF_BAUD 9600
  The default UART baud rate - 9600.
- #define IX_UART_MIN_BAUD 9600
  The minimum UART baud rate - 9600.
- #define IX_UART_MAX_BAUD 926100
  The maximum UART baud rate - 926100.
- #define IX_UART_XTAL 14745600
  The UART clock speed.
- #define IX_BAUD_SET 0
  set the baud rate.
- #define IX_BAUD_GET 1
  get the baud rate.
- #define IX_MODE_SET 2
  set the UART mode of operation.
- #define IX_MODE_GET 3
  get the current UART mode of operation.
- #define IX_OPTS_SET 4
  set the UART device options.
- #define IX_OPTS_GET 5
  get the UART device options.
- #define IX_STATS_GET 6
  get the UART statistics.
- #define CLOCAL 0x1
  software flow control.
- #define CREAD 0x2
  enable interrupt receiver.
- #define CSIZE 0xc
  characters size.
- #define CS5 0x0
  5 bits.
- #define CS6 0x4
  6 bits.
- #define CS7 0x8

7 bits.

- #define CS8  0xc
  8 bits.
- #define STOPB  0x20
  send two stop bits (else one).
- #define PARENB  0x40
  parity detection enabled (else disabled).
- #define PARODD  0x80
  odd parity (else even).

### A.3.28.3 Typedefs

- typedef enum _ixUARTMode ixUARTMode
  The mode to set to UART to.
- typedef _ixUARTStats ixUARTStats
- typedef _ixUARTDev ixUARTDev

### A.3.28.4 Enumerations

- enum _ixUARTMode { INTERRUPT = 0, POLLED, LOOPBACK }

### A.3.28.5 Functions

- PUBLIC IX_STATUS ixUARTInit (ixUARTDev *pUART)
  Initialize the UART. This puts the chip in a quiescent state.
- PUBLIC IX_STATUS ixUARTPollOutput (ixUARTDev *pUART, int outChar)
  Transmit a character in polled mode.
- PUBLIC IX_STATUS ixUARTPollInput (ixUARTDev *pUART, char *inChar)
  Receive a character in polled mode.
- PUBLIC IX_STATUS ixUARTIoctl (ixUARTDev *pUART, int cmd, void *arg)
  Perform I/O control routines on the device.

### A.3.28.6 Detailed Description

Intel® IXP42X product line UARTAcc Driver Public API.

### A.3.28.7 Enumeration Type Documentation

#### A.3.28.7.1 enum _ixUARTMode

**Enumeration values**

- ***INTERRUPT*** — Interrupt mode.
- ***POLLED*** — Polled mode.

- ***LOOPBACK*** — Loopback mode.

## A.3.28.8 Function Documentation

### A.3.28.8.1 IX_STATUS ixUARTInit (..)

IX_STATUS ixUARTInit (ixUARTDev * pUART)

Initialize the UART. This puts the chip in a quiescent state.

#### Parameters

***pUART*** — pointer to UART structure describing our device.

#### Returns

IX_SUCCESS — UART device successfully initialized. IX_FAIL - Critical error, device not initialized.

### A.3.28.8.2 IX_STATUS ixUARTIoctl (..)

IX_STATUS ixUARTIoctl (ixUARTDev * pUART, int cmd, void * arg)

Perform I/O control routines on the device.

#### Parameters

***pUART*** — pointer to UART structure describing our device.

***cmd*** — ioctl request code.

***arg*** — optional argument used to set the device mode, baud rate, and hardware options.

#### Returns

IX_SUCCESS — requested feature was set/read successfully. IX_FAIL - error setting/reading the requested feature.

### A.3.28.8.3 IX_STATUS ixUARTPollInput (..)

IX_STATUS ixUARTPollInput (ixUARTDev * pUART, char * inChar)

Receive a character in polled mode.

#### Parameters

***pUART*** — pointer to UART structure describing our device.

***inChar*** — character read from the device.

#### Returns

IX_SUCCESS — character was successfully read.

IX_FAIL - input buffer empty (try again).

#### A.3.28.8.4 IX_STATUS ixUARTPollOutput (..)

IX_STATUS ixUARTPollOutput (ixUARTDev * pUART, int outChar)

Transmit a character in polled mode.

**Parameters**

***pUART*** — pointer to UART structure describing our device.

***outChar*** — character to transmit.

**Returns**

IX_SUCCESS — character was successfully transmitted.

IX_FAIL - output buffer is full (try again).

## A.3.29 Version ID (IxVersionId)

Version Identifier.

### A.3.29.1 Defines

- #define IX_VERSION_ID "1_1"
  Version Identifier String.

### A.3.29.2 Detailed Description

Version Identifier.

### A.3.29.3 Define Documentation

#### A.3.29.3.1 #define IX_VERSION_ID "1_1"

Version Identifier String.

This string will be updated with each customer release of the IXP400 software.

## A.3.30 USB Access (IxUsb) API

Intel® IXP42X product line USB Driver Public API.

### A.3.30.1 Functions

- PUBLIC IX_STATUS ixUSBDriverInit (USBDevice *device)
  Initialize driver and USB Device Controller.
- PUBLIC IX_STATUS ixUSBDeviceEnable (USBDevice *device, BOOL enableDevice)
  Enable or disable the device.

- PUBLIC IX_STATUS ixUSBEndpointStall (USBDevice *device, UINT16 endpointNumber, BOOL stallFlag)

  Enable or disable endpoint stall (or halt feature).
- PUBLIC IX_STATUS ixUSBEndpointClear (USBDevice *device, UINT16 endpointNumber)

  Free all Rx/Tx buffers associated with an endpoint.
- PUBLIC IX_STATUS ixUSBSignalResume (USBDevice *device)

  Trigger signal resuming on the bus.
- PUBLIC IX_STATUS ixUSBFrameCounterGet (USBDevice *device, UINT16 *counter)

  Retrieve the 11-bit frame counter.
- PUBLIC IX_STATUS ixUSBReceiveCallbackRegister (USBDevice *device, USBReceiveCallback callbackFunction)

  Register a data receive callback.
- PUBLIC IX_STATUS ixUSBSetupCallbackRegister (USBDevice *device, USBSetupCallback callbackFunction)

  Register a setup receive callback.
- PUBLIC IX_STATUS ixUSBBufferSubmit (USBDevice *device, UINT16 destinationEndpoint, IX_USB_MBLK *sendBuffer)

  Submit a buffer for transmit.
- PUBLIC IX_STATUS ixUSBBufferCancel (USBDevice *device, UINT16 destinationEndpoint, IX_USB_MBLK *sendBuffer)

  Cancel a buffer previously submitted for transmitting.
- PUBLIC IX_STATUS ixUSBEventCallbackRegister (USBDevice *device, USBEventCallback eventCallback, USBEventMap eventMap)

  Register an event callback.
- PUBLIC IX_STATUS ixUSBIsEndpointStalled (USBDevice *device, UINT16 endpointNumber, BOOL *stallState)

  Retrieve an endpoint's stall status.

## A.3.30.2 Detailed Description

Intel® IXP42X product line USB Driver Public API.

## A.3.30.3 Function Documentation

### A.3.30.3.1 PUBLIC IX_STATUS ixUSBBufferCancel (..)

PUBLIC IX_STATUS ixUSBBufferCancel (USBDevice * device, UINT16 destinationEndpoint, IX_USB_MBLK * sendBuffer)

Cancel a buffer previously submitted for transmitting.

**Parameters**

***device*** USBDevice * (in) — a structure identifying the device

***destinationEndpoint*** UINT16 (in) — endpoint originally used for transmitting the data buffer

***sendBuffer*** IX_USB_MBLK * (in) — submitted data buffer

#### Returns

- IX_SUCCESS — if the initialization was successful
- IX_FAIL — otherwise, in which case a detailed error code will be set in the *lastError* field, unless the *device* parameter is NULL.

### A.3.30.3.2 PUBLIC IX_STATUS ixUSBBufferSubmit (..)

PUBLIC IX_STATUS ixUSBBufferSubmit (USBDevice * device, UINT16 destinationEndpoint, IX_USB_MBLK * sendBuffer)

Submit a buffer for transmit.

#### Parameters

***device*** USBDevice * (in) — a structure identifying the device

***destinationEndpoint*** UINT16 (in) — endpoint to be used for transmitting the data buffer

***sendBuffer*** IX_USB_MBLK * (in) — data buffer

#### Returns

- IX_SUCCESS — if the initialization was successful
- IX_FAIL — otherwise, in which case a detailed error code will be set in the *lastError* field, unless the *device* parameter is NULL.

### A.3.30.3.3 PUBLIC IX_STATUS ixUSBDeviceEnable (..)

PUBLIC IX_STATUS ixUSBDeviceEnable (USBDevice * device, BOOL enableDevice)

Enable or disable the device.

#### Parameters

***device*** USBDevice * (in) — a structure identifying the device

***enableDevice*** BOOL (in) —

- **true** to enable the device
- **false** to disable it

This function enables or disables the device. A disabled device doesn't generate events and cannot send or receive data.

Disabling the device frees and discards all existent Rx/Tx buffers (received buffers that weren't dispatched yet and buffers waiting to be transmitted)

#### Returns

- IX_SUCCESS — if the initialization was successful
- IX_FAIL — otherwise, in which case a detailed error code will be set in the *lastError* field, unless the *device* parameter is NULL.

### A.3.30.3.4 PUBLIC IX_STATUS ixUSBDriverInit (..)

PUBLIC IX_STATUS ixUSBDriverInit (USBDevice * device)

Initialize driver and USB Device Controller.

#### Parameters

***device*** USBDevice * (inout) — a structure identifying the device

This function initializes the UDC and all the data structures used to interact with the controller.

It is the responsibility of the caller to create the USBDevice structure and fill in the correct baseIOAddress and interruptLevel fields.

After successful initialization the device will be inactive - use ixUSBDeviceEnable to activate the device.

Use the flags component of the device structure to pass in additional flags such as ENABLE_RX_SEQ or ENABLE_TX_SEQ. Changing these flags later will have no effect.

The driver will assign a device number which will be placed in the *deviceIndex* field.

The initialized *device* structure must be used for all iterations with the USB controller. The same *device* pointer will be passed in to all the registered client callbacks.

The *deviceIndex* and *deviceContext* should be treated as read-only fields.

***Warning:*** This function is not reentrant.

#### Returns

- IX_SUCCESS — if the initialization was successful
- IX_FAIL — otherwise, in which case a detailed error code will be set in the *lastError* field, unless the *device* parameter is NULL.

### A.3.30.3.5 PUBLIC IX_STATUS ixUSBEndpointClear (..)

PUBLIC IX_STATUS ixUSBEndpointClear (USBDevice * device, UINT16 endpointNumber)

Free all Rx/Tx buffers associated with an endpoint.

#### Parameters

***device*** USBDevice * (in) — a structure identifying the device

***endpointNumber*** UINT16 (in) — endpoint number

This function discards and frees all Tx/Rx buffers associated with an endpoint. The corresponding endpoint dropped packet counters will also be incremented.

#### Returns

- IX_SUCCESS — if the initialization was successful
- IX_FAIL — otherwise, in which case a detailed error code will be set in the *lastError* field, unless the *device* parameter is NULL.

### A.3.30.3.6 PUBLIC IX_STATUS ixUSBEndpointStall (..)

PUBLIC IX_STATUS ixUSBEndpointStall (USBDevice * device, UINT16 endpointNumber, BOOL stallFlag)

Enable or disable endpoint stall (or *halt* feature).

#### Parameters

***device*** USBDevice * (in) — a structure identifying the device

***endpointNumber*** UINT16 (in) — endpoint number

***stallFlag*** BOOL (in) —

- **true** to set endpoint stall
- **false** to clear it

This function clears or sets the endpoint stall (or *halt*) feature.

Both IN and OUT endpoints can be stalled. A stalled endpoint will not send or receive data. Instead, it will send USB STALL packets in response to IN or OUT tokens.

Unstalling endpoints can be done only by using this function with the exception of endpoint 0 which unstalls itself automatically upon receiving a new SETUP packet, as required by the USB 1.1 Specification. Isochronous endpoints cannot be stalled and attempting to do so will return an IX_USB_NO_STALL_CAPABILITY failure.

#### Returns

- IX_SUCCESS — if the initialization was successful
- IX_FAIL — otherwise, in which case a detailed error code will be set in the *lastError* field, unless the *device* parameter is NULL.

### A.3.30.3.7 PUBLIC IX_STATUS ixUSBEventCallbackRegister (..)

PUBLIC IX_STATUS ixUSBEventCallbackRegister (USBDevice * device, USBEventCallback eventCallback, USBEventMap eventMap)

Register an event callback.

#### Parameters

***device*** USBDevice * (in) — a structure identifying the device

***eventCallback*** USBEventCallback (in) — event callback function

***eventMap*** USBEventMap (in) — event map

#### Returns

- IX_SUCCESS — if the initialization was successful
- IX_FAIL — otherwise, in which case a detailed error code will be set in the *lastError* field, unless the *device* parameter is NULL.

### A.3.30.3.8 PUBLIC IX_STATUS ixUSBFrameCounterGet (..)

PUBLIC IX_STATUS ixUSBFrameCounterGet (USBDevice * device, UINT16 * counter)

Retrieve the 11-bit frame counter.

#### Parameters

***device*** USBDevice * (in) — a structure identifying the device

***counter*** UINT16 * (out) — the 11-bit frame counter

This function returns the hardware USB frame counter.

Since the counter is 11-bit wide it rolls over after every 2048 frames.

#### Returns

- IX_SUCCESS — if the initialization was successful
- IX_FAIL — otherwise, in which case a detailed error code will be set in the *lastError* field, unless the *device* parameter is NULL.

### A.3.30.3.9 PUBLIC IX_STATUS ixUSBIsEndpointStalled (..)

PUBLIC IX_STATUS ixUSBIsEndpointStalled (USBDevice * device, UINT16 endpointNumber, BOOL * stallState)

Retrieve an endpoint's stall status.

#### Parameters

***device*** USBDevice * (in) — a structure identifying the device

***endpointNumber*** UINT16 (in) — endpoint number

***stallState*** BOOL * (out) — stall state

- **true** if the endpoint is stalled (*halted*)
- **false** otherwise

#### Returns

- IX_SUCCESS
- IX_FAIL — if the device pointer is invalid or the endpoint doesn't exist

### A.3.30.3.10 PUBLIC IX_STATUS ixUSBReceiveCallbackRegister (..)

PUBLIC IX_STATUS ixUSBReceiveCallbackRegister (USBDevice * device, USBReceiveCallback callbackFunction)

Register a data receive callback.

#### Parameters

***device*** USBDevice * (in) — a structure identifying the device

***callbackFunction*** USBReceiveCallback (in) — receive callback function

#### Returns

- IX_SUCCESS — if the initialization was successful
- IX_FAIL — otherwise, in which case a detailed error code will be set in the *lastError* field, unless the *device* parameter is NULL.

### A.3.30.3.11 PUBLIC IX_STATUS ixUSBSetupCallbackRegister (..)

PUBLIC IX_STATUS ixUSBSetupCallbackRegister (USBDevice * device, USBSetupCallback callbackFunction)

Register a setup receive callback.

#### Parameters

***device*** USBDevice * (in) — a structure identifying the device

***callbackFunction*** USBSetupCallback (in) — setup callback function

#### Returns

- IX_SUCCESS — if the initialization was successful
- IX_FAIL — otherwise, in which case a detailed error code will be set in the *lastError* field, unless the *device* parameter is NULL.

### A.3.30.3.12 PUBLIC IX_STATUS ixUSBSignalResume (..)

PUBLIC IX_STATUS ixUSBSignalResume (USBDevice * device)

Trigger signal resuming on the bus.

#### Parameters

***device*** USBDevice * (in) — a structure identifying the device

This function triggers signal resuming on the bus, waking up the USB host. Is should be used only if the host has enabled the device to do so using the standard SET_FEATURE USB request, otherwise the function will return IX_FAIL and set the *lastError* field to IX_USB_NO_PERMISSION.

#### Returns

- IX_SUCCESS — if the initialization was successful
- IX_FAIL — otherwise, in which case a detailed error code will be set in the *lastError* field, unless the *device* parameter is NULL.

## A.4 Intel® IXP400 SoftwareCodelets

Intel® IXP42X product line demonstration codelets.

## A.4.1 Codelet Overview

### A.4.1.1 Modules

- Intel® IXP42X product line USB RNDIS Codelet (IxUSBRNDIS) API
  Intel® IXP42X product line codelet USB RNDIS API.
- Intel® IXP42X product line UART Access Codelet (IxUARTAccCodelet) API
  The interface prototypes for Intel® IXP42X product line UART codelet.
- Intel® IXP42X product line OAM Codelet (IxOamCodelet) API
  Intel® IXP42X product line codelet OAM demo component API.
- Intel® IXP42X product line HSS Access Codelet (IxHssAccCodelet) API
  Intel® IXP42X product line demo HSS Acc component API. The interface for the HSS Access Codelet.
- Intel® IXP42X product line Fast Path Access Codelet (IxFpathAccCodelet) API
  Intel® IXP42X product line demo Fast Path Access component API.
- Intel® IXP42X product line Ethernet Access Codelet (IxEthAccCodelet) API
  Intel® IXP42X product line demo Ethernet Access component API.
- Intel® IXP42X product line Ethernet Aal5 (IxEthAal5App) API
  Intel® IXP42X product line demo Ethernet Aal5 driver component API.
- Intel® IXP42X product line DMA Access Codelet (IxAal5Demo) API
  Intel® IXP42X product line demo DMA Access component API.
- Intel® IXP42X product line Demo Utils (IxDemoUtils) API
  Intel® IXP42X product line Demo Utils component API.
- Intel® IXP42X product line Crypto Access Codelet (IxCryptoAccCodelet) API
  Intel® IXP42X product line demo Crypto Access component API.
- Intel® IXP42X product line Atm Codelet (IxAtmCodelet) API
  Intel® IXP42X product line codelet ATM component API.
- Intel® IXP42X product line Aal5 HSS Codelet (IxAal5HssCodelet) API
  *API of the* Intel® IXP42X product line AAL5/HSS Codelet.
- Intel® IXP42X product line Aal5 Main Codelet (IxAal5CodeletMain) API
  Intel® IXP42X product line codelet AAL5 Main component API.

### A.4.1.2 Detailed Description

Intel® IXP42X product line demo codelets.

## A.4.2 AAL5 Main Codelet (IxAal5CodeletMain) API

Intel® IXP42X product line codelet AAL5 Main component API.

### A.4.2.1 Functions

- PUBLIC IX_STATUS ixAal5CodeletMainUtopiaLoopback (void)
  Function prototype for Utopia loopback.
- PUBLIC IX_STATUS ixAal5CodeletMainSoftwareLoopback (void)
  Function prototype for Software loopback.
- PUBLIC IX_STATUS ixAal5CodeletMainRemoteLoopback (void)
  Function prototype for Remote loopback.
- PUBLIC IX_STATUS ixAal5CodeletMainUtopiaLoopbackRun (void)
  Function prototype for Utopia loopback run.
- PUBLIC IX_STATUS ixAal5CodeletMainSoftwareLoopbackRun (void)
  Function prototype for Software loopback run.

### A.4.2.2 Detailed Description

Intel® IXP42X product line codelet AAL5 Main component API.

## A.4.3 Aal5 HSS Codelet (IxAal5HssCodelet) API

API of the Intel® IXP42X product line AAL5/HSS Codelet.

### A.4.3.1 Defines

- #define IX_AAL5_HSS_CODELET_QMGR_INTERRUPT_MODE
  Run IxQMgrDispatcher from interrupt.

### A.4.3.2 Functions

- PUBLIC void ixAal5HssCodeletMain (void)
  Start the IxAal5HssCodelet.

### A.4.3.3 Detailed Description

API of the Intel® IXP42X product line AAL5/HSS Codelet.

This codelet demonstrates running the ATM(aal5Demo) and HSS(hssAcc) codelets simultaneously.

***Note:*** Both codelets are run independently, there is NO movement of data between the codelets.

There is one configuration parameter IX_AAL5_HSS_CODELET_QMGR_INTERRUPT_MODE. If this is defined then the IxQMgrDispatcher is hooked to interrupt otherwise it is periodically from a task. This parameter is contained within this file.

The ATM codelet is setup for software loopback mode of operation.

The HSS codelet is setup for running one channelised client using 16 voice channels, and four packetised clients each using 16 timeslots to send 16K packets once every 200ms.

### A.4.3.4 Define Documentation

#### A.4.3.4.1 #define IX_AAL5_HSS_CODELET_QMGR_INTERRUPT_MODE

Run IxQMgrDispatcher from interrupt.

This define specifies that the IxQMgrDispatcher is hooked to interrupt.

### A.4.3.5 Function Documentation

#### A.4.3.5.1 PUBLIC void ixAal5HssCodeletMain (void)

Start the IxAal5HssCodelet.

**Parameters**

none

**Returns**

void

## A.4.4 Atm Codelet (IxAtmCodelet) API

Intel® IXP42X product line codelet ATM component API.

### A.4.4.1 Enumerations

- enum IxAtmCodeletMode { IX_ATMCODELET_UTOPIA_LOOPBACK = 0, IX_ATMCODELET_SOFTWARE_LOOPBACK, IX_ATMCODELET_REMOTE_LOOPBACK }

  Mode of operation of the AtmCodelet codelet.

### A.4.4.2 Functions

- PUBLIC IX_STATUS ixAtmCodeletSystemInit (unsigned int numPorts, IxAtmCodeletMode mode)

  This function is used to initialize the lower level system component used by the IxAtmCodelet. It should be called before any other IxAtmCodelet API function UNLESS the system initialization is done elsewhere.

- IX_STATUS ixAtmCodeletInit (unsigned int numPorts, unsigned int rxToTxRatio)

  This function is used to initialize the IxAtmCodelet. It should be called before any other IxAtmCodelet API function. This function can be called from the VxWorks windSh.

- IX_STATUS ixAtmCodeletUbrChannelsProvision (unsigned int numPorts, unsigned int numChannels, IxAtmdAccAalType aalType)

This function is used to provision a number of bidirectional Atm UBR channels on the specified number of ports. This function may be called a number of times for different Aal types until all of the available 32 channels are used.

- IX_STATUS ixAtmCodeletUbrChannelsRemove (void)

  This function is used to remove all channels provisioned earlier in a call to ixAtmCodeletUbrChannelsProvision().

- IX_STATUS ixAtmCodeletAal0PacketsSend (unsigned int cellsPerPacket, unsigned int numPackets)

  This function is used to send the specified number of Aal0 packets, with the number of cells per packet specified. The packets are multiplexed across all provisioned Aal0 48/52 channels. This function will block until the transmission is complete.

- IX_STATUS ixAtmCodeletAal5CpcsSdusSend (unsigned int sduSize, unsigned int numSdus)

  This function is used to send the specified number of Aal5 CPCS SDUs of specified SDU size multiplexed across all Aal5 provisioned channels. This function will block until the transmission is complete.

- IX_STATUS ixAtmCodeletPortRateModify (IxAtmLogicalPort port, unsigned int portRate)

  This function is called to modify the portRate on a previously initialized port.

- void ixAtmCodeletShow (void)

  This function is used to dump VC configuration and sdu send and receive counters.

### A.4.4.3 Detailed Description

Intel® IXP42X product line codelet ATM component API.

This codelet demonstrates an example implementation of a working Atm driver that makes use of the AtmdAcc component, as well as demonstrating how the lower layer IxAtmdAcc component can be used for configuration and control.

#### Features

An interface is provided to setup Aal5/Aal0 Transmit and Receive VCs on one or more Ports. A maximum of 32 Tx and 32 Rx VCs can be setup on a maximum of 12 Ports. Only UBR VCs can be setup. When a channel is setup using this interface both a Transmit and a Receive VC is created. Aal0 and Aal5 VCs can be setup at the same time.

An interface is provided to remove all registered Aal5/Aal0 VCs.

Both remote and local loopback of Aal5/Aal0 is provided by this codelet. Local loopback refers to loopback provided by the UTOPIA interface. In local loopback packets generated by the Intel® IXP42X product line are looped back to the Atm driver by the UTOPIA hardware. Software loopback refers to the software looping all packets received on the wire back out onto the wire. Remote loopback refers to where the far end is expected to perform a software loopback, i.e. any packets sent by the codelet are expected to be looped back by the far end into the codelet.

Both interrupt and polled mode of operation is provided by this codelet.

An interface is provided to send Aal5 SDUs specifying the sdu size and the number of Aal5 sdus to send.

An interface is provided to send Aal0 packets specifying the packet size and the number of Aal0 packets to send.

Both the Transmit port rate and the Receive port rate can be modified. The Tx port rate is used by IxAtmSch in performing the shaping functions. The Rx port rate is not used by any component.

An interface is provided to allow the querying of the ATM ports and registered VCs.

Intel® IXP42X product line ATM Components used by this codelet

IxAtmdAcc. This component is the low level interface by which AAL packets get transmitted to, and received from the UTOPIA bus.

### ATM Codelet components used by this codelet

IxAtmSch. This component demonstrates an ATM Traffic Shaper implementation. IxAtmdAcc gets scheduling information from this component.

IxAtmm. This component provides ATM port and VC management facilities. This component also manages the configuration of the UTOPIA co-processor.

### IxAtmCodelet modes of operation

This codelet can be initialized to operate in one of three configurations.

IX_ATMCODELET_UTOPIA_LOOPBACK. In this mode the UTOPIA interface will loopback all traffic transmitted.

### Buffer management

In this mode a simple buffering mechanism is used; mbufs are allocated from the vxWorks pool as needed for RxFree replenishing/Tx and are returned to the vxWorks pool for TxDone/Rx.

### Interrupt/Task based processing

In this mode of operation the IxQMgrDispatcher is hooked to interrupts. This means that TxDone/RxLo will be performed from a task level and IxAtmdAcc Tx processing/RxHi and RxFree will be precessed from an interrupt level.

### Sending Aal5 PDUs

In this mode of operation an interface is available to send Aal5 PDUs of specified size in bytes.

### Sending Aal0 Packets

In this mode of operation an interface is available to send Aal0 Packets of specified size in cells.

IX_ATMCODELET_REMOTE_LOOPBACK. In this mode packets are sent and are expected to be looped back on the remote end.

IX_ATMCODELET_SOFTWARE_LOOPBACK. In this mode of operation all traffic sent are looped back to receive in the Intel® IXP42X product line device.

### Buffer management

In this mode a more complex buffering mechanism is used; mbufs are allocated from the vxWorks pool and stored in a software queue. These mbufs are fetched from this software queue as needed for RxFree replenishing/Tx and are returned to the software queue for TxDone/Rx.

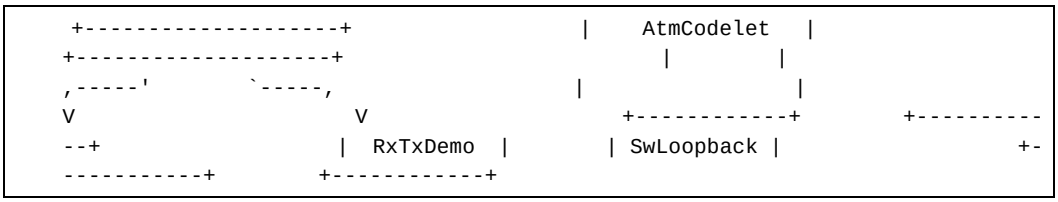
#### Interrupt/Task based processing

In this mode of operation the IxQMgrDispatcher is called form a task every 1 msec. This means that TxDone/RxLo/IxAtmdAcc Tx processing/RxHi and RxFree will be processed from a task level.

#### Sending PDUs/Packets

This mode of operation does not provide an interface for sending Aal5 PDUs/Aal0 Packets.

#### IxAtmCodelet sub-components




#### AtmCodelet

This implements the API functions.

#### RxTxDemo

This sub-component implements the IX_ATMCODELET_UTOPIA_LOOPBACK and IX_ATMCODELET_REMOTE_LOOPBACK modes of operation.

#### SwLoopback

This sub-component implements the IX_ATMCODELET_SOFTWARE_LOOPBACK mode operation.

#### BufMan

This sub-component implements the interfaces used internally for getting and returning vxWorks mbufs.

#### UTOPIA Receive and Transmit PHY addresses

In this codelet UTOPIA Phy Addresses are assigned numbers starting at UTOPIA_PHY_ADDR and are incremented for each port used. These addresses _WILL_ need to be changed depending on the hardware setup.

Modification history 21 Aug 02 Ported functionality from IxAal5DemoCodelet.h. Added Aal0 + OAM support

### A.4.4.4 Enumeration Type Documentation

enum IxAtmCodeletMode

Mode of operation of the AtmCodelet codelet.

#### Enumeration values

- ***IX_ATMCODELET_UTOPIA_LOOPBACK*** — All cells sent out are looped back in by the UTOPIA hardware.

  Useful for demonstrating traffic on a board with no ADSL PHY or external test equipment
- ***IX_ATMCODELET_SOFTWARE_LOOPBACK*** — All cells received are looped back out again by the software.
- ***IX_ATMCODELET_REMOTE_LOOPBACK*** — All cells sent are anticipated to be looped back by external equipment.

### A.4.4.5 Function Documentation

#### A.4.4.5.1 IX_STATUS ixAtmCodeletAal0PacketsSend (..)

IX_STATUS ixAtmCodeletAal0PacketsSend (unsigned int cellsPerPacket, unsigned int numPackets)

This function is used to send the specified number of Aal0 packets, with the number of cells per packet specified. The packets are multiplexed across all provisioned Aal0 48/52 channels. This function will block until the transmission is complete.

#### Parameters

***unsigned int cellsPerPacket*** — The number of cells per packet to send.

***unsigned int numPackets*** — The total number of packets to send.

#### Returns

- **IX_SUCCESS** — Indicates that the Atm Codelet has successfully sent all of the packets.
- **IX_FAIL** — Some internal error has prevented the Atm Codelet from sending the packets successfully.

#### A.4.4.5.2 IX_STATUS ixAtmCodeletAal5CpcsSdusSend (..)

IX_STATUS ixAtmCodeletAal5CpcsSdusSend (unsigned int sduSize, unsigned int numSdus)

This function is used to send the specified number of Aal5 CPCS SDUs of specified SDU size multiplexed across all Aal5 provisioned channels. This function will block until the transmission is complete.

#### Parameters

***unsigned int sduSize*** — The SDU size of the sdus to send in bytes. The Atm trailer + padding will be appended to this.

***unsigned int numsdus*** — The total number of sdus to send.

#### Returns

- **IX_SUCCESS** — Indicates that the Atm Codelet has successfully sent all of the sdus.
- **IX_FAIL** — Some internal error has prevented the Atm Codelet from sending the sdus successfully.

#### A.4.4.5.3 IX_STATUS ixAtmCodeletInit (..)

IX_STATUS ixAtmCodeletInit (unsigned int numPorts, unsigned int rxToTxRatio)

This function is used to initialize the IxAtmCodelet. It should be called before any other IxAtmCodelet API function. This function can be called from the VxWorks windSh.

**Parameters**

***unsigned int numPorts*** — The number of ATM ports to configure. Valid values range from 1 to 12.

***unsigned int rxToTxRatio*** — Ratio of PDUs received to attempted PDU sends when softwareLoopbackEnabled = TRUE. e.g. if rxToTxRatio is 6 then 1 PDU will be sent for every 6 PDUs received.

**Returns**

- **IX_SUCCESS** — Indicates that the Atm Codelet has been successfully initialized.
- **IX_FAIL** — Some internal error has prevented the Atm Codelet from initializing.

#### A.4.4.5.4 IX_STATUS ixAtmCodeletPortRateModify (..)

IX_STATUS ixAtmCodeletPortRateModify (IxAtmLogicalPort port, unsigned int portRate)

This function is called to modify the portRate on a previously initialized port.

**Parameters**

***IxAtmLogicalPort port*** — Specifies the ATM port which is to be modified.

**Returns**

- **IX_SUCCESS** — Port rate has been successfully modified.
- **IX_FAIL** — Port rate could not be modified, either because the input data was invalid, or the new port rate is insufficient to support established ATM VC contracts on this port.

#### A.4.4.5.5 void ixAtmCodeletShow (void)

This function is used to dump VC configuration and sdu send and receive counters.

**Parameters**

none

**Returns**

void

#### A.4.4.5.6 PUBLIC IX_STATUS ixAtmCodeletSystemInit (..)

PUBLIC IX_STATUS ixAtmCodeletSystemInit (unsigned int numPorts, IxAtmCodeletMode mode)

This function is used to initialize the lower level system component used by the IxAtmCodelet. It should be called before any other IxAtmCodelet API function UNLESS the system initialization is done elsewhere.

#### Parameters

***unsigned int numPorts*** — The number of ATM ports to configure. Valid values range from 1 to 12.

***IxAtmDemoCodeletMode mode*** — This indicates the codelet mode.

#### Returns

- **IX_SUCCESS** — Indicates that the Atm Codelet system init successful.
- **IX_FAIL** — Some internal error has prevented the Atm Codelet system components from initializing.

### A.4.4.5.7 IX_STATUS ixAtmCodeletUbrChannelsProvision (..)

IX_STATUS ixAtmCodeletUbrChannelsProvision (unsigned int numPorts, unsigned int numChannels, IxAtmdAccAalType aalType)

This function is used to provision a number of bidirectional Atm UBR channels on the specified number of ports. This function may be called a number of times for different Aal types until all of the available 32 channels are used.

#### Parameters

***unsigned int numPorts*** — The number of ATM ports to configure channels on. This must be less than or equal to the number of ports configured in the call to ixAtmCodeletInit().

***unsigned int numChannels*** — The number of bidirectional Atm channels to provision. Channels are provisioned across numPorts in the following manner, for 2 ports, 4 VCs, Channel 0 on port 0, channel 1 on port 1, channel 2 on port 0, channel 3 on port 1. Valid values range from 1 to 32. Subsequent calls will assign channel and port numbers in ascending order.

***IxAtmdAccAalType aalType*** — The aalType for the channels to provision, i.e. Aal5, Aal0 48 byte mode or Aal0 52 byte cell mode.

#### Returns

- **IX_SUCCESS** — Indicates that the Atm Codelet has successfully provisioned the requested number of channels.
- **IX_FAIL** — Some internal error has prevented the ATM codelet from provisioning the channels.

### A.4.4.5.8 IX_STATUS ixAtmCodeletUbrChannelsRemove (void)

This function is used to remove all channels provisioned earlier in a call to ixAtmCodeletUbrChannelsProvision().

#### Parameters

none

**Returns**

- **IX_SUCCESS** — Indicates that the Atm Codelet has successfully removed all provisioned channels.
- **IX_FAIL** — Some internal error has prevented the Atm Codelet from removing provisioned channels.

## A.4.5 Crypto Access Codelet (IxCryptoAccCodelet) API

Intel® IXP42X product line demo Crypto Access component API.

### A.4.5.1 Defines

- #define INLINE __inline__
- #define IX_CRYPTO_ACC_CODELET_DRV_DMA_FREE(pData) (pData)
  To free the memory allocated through IX_CRYPTO_ACC_DRV_DMA_MALLOC function.

### A.4.5.2 Functions

- IX_STATUS ixCryptoAccCodeletInit (void)
  This function is the entry point to the Crypto Access codelet. It must be called before any of the demos as it initializes the necessary component.
- void ixCryptoAccCodeletMbufToQTailAdd (IX_MBUF *pMbufToAdd)
  This function will add an mbuf to the queue. The mbuf will be added to the end of the queue.
- void ixCryptoAccCodeletMbufFromQHeadRemove (IX_MBUF **pMbufToRem)
  This function will remove a mbuf from the queue. The mbuf will be removed from the head of the queue.
- PRIVATE IX_STATUS ixCryptoAccCodeletMemPoolInit (void)
  This function will initialize mbuf pool for the Crypto Access Codelet.
- PRIVATE IX_STATUS ixCryptoAccCodeletNpeInit (IxNpeDlNpeId npeId)
  This function is to download microcode to the NPE. This function always downloads the latest crypto version of NPE microcode available.
- PRIVATE IX_STATUS ixCryptoAccCodeletDispatcherStart (BOOL useInterrupt)
  This function starts the Queue manager dispatcher.
- PRIVATE void ixCryptoAccCodeletDispatcherPoll (void)
  This function polls the Queue manager for the poll mode set in the function ixCryptoAccCodeletDispatcherStart.
- IX_STATUS ixCryptoAccCodeletRegisterCtx (UINT32 opType)
  This function will register to the Crypto Access Component for three different operation which will demonstrate the capability of the security hardware accelerator. The types of operation are:
  - — Encryption and Decryption
  - — Authentication: Calculation and Check
  - — Encryption/Decryption with authentication.

- void test_main (void)
  This is the main function of the Codelet where registration and perform is done. This function will also rotate between encryption/authentication and authentication/decryption.
- INLINE BOOL ixCryptoAccCodeletBufIsQEmpty (void)
  This inline function is used to check whether all the buffer queue is empty.
- INLINE BOOL ixCryptoAccCodeletBufIsQFull (void)
  This inline function is used to check whether all the buffer queue is full.

### A.4.5.3 Detailed Description

Intel® IXP42X product line demo Crypto Access component API.

#### Functionality of the cryptoAcc Codelet

The codelet shall demonstrate how the Security Hardware Accelerator API can be used for cryptographic purposes.

Three different implementations are demonstrated.

- Encryption and decryption with DES in CBC mode.
- Authentication calculation and check for SHA1.
- Combined service of encryption/authentication calculation and authentication check/ decryption. The combined services uses DES CBC mode with SHA1.

Here is the list of functionality of the codelet and they are in the order in which the codelet must follow in order to use the Security Hardware Accelerator, i.e. Initialize the related component and followed by registration, and then perform followed by unregistration, etc.

- The codelet shall demonstrate how the Network Processor Engine (NPE) is initialized and started. It will also show how the latest NPE crypto image is downloaded to the NPE.
- The codelet shall demonstrate how the Queue Manager is initialized to polling or interrupt mode.
- The codelet will show an example of register and perform callback functions and how they are set up.
- The codelet will demonstrate how the registration should be done for the three different implementation using the register API. Note that the authentication data doesn't matter when encrypt/decrypt only operation is chosen and vice-versa.
- The codelet shall demonstrate how the perform API can be used after successful registration.
- The codelet shall demonstrate the use of the unregister API in the event of re-starting a new functionality or implementation.
- The codelet will measure the performance of the Security Hardware Accelerator for the three implementation stated above.

For COEXIST functionality, -DCOEXIST should be defined in cryptoAcc codelet component.mk. The COEXIST codelet shall: Demonstrate that cryptographic process can be run simultaneously with ethernet packet transmission in port 2 in vxWorks only.

***Notes:*** For interrupt mode:

— Packets could be lost due to the usage of log messages, in crypto and COEXIST mode. Longer packets will eliminate this effect due to the fact that the NPE is processing faster than Intel XScale core.

— Longer packets for the crypto is required (>1024 bytes) for COEXIST to work and the line rate utilization of approximately 50% is required. Higher utilization rate will result in the cryptoAcc callback not being serviced.

For Linux*, only one callback is demonstrated per buffer. Perform function is not executed in the Perform callback function.

## A.4.5.4 Define Documentation

### A.4.5.4.1 #define IX_CRYPTO_ACC_CODELET_DRV_DMA_FREE(..)

#define IX_CRYPTO_ACC_CODELET_DRV_DMA_FREE(pData)  (pData)

To free the memory allocated through IX_CRYPTO_ACC_DRV_DMA_MALLOC function.

**Note**

Example of usage: IX_CRYPTO_ACC_CODELET_DRV_DMA_FREE(pData)

**Parameters**

***UINT32*** [in] *pData — memory block pointer to be freed.

**Returns**

None

## A.4.5.5 Function Documentation

### A.4.5.5.1 ixCryptoAccCodeletBufIsQEmpty (void)

This inline function is used to check whether all the buffer queue is empty.

**Returns**

- BOOL
  - — TRUE - the queue is full.
  - — FALSE - the queue is not full.

### A.4.5.5.2 ixCryptoAccCodeletBufIsQFull (void)

This inline function is used to check whether all the buffer queue is full.

**Returns**

- BOOL
  - — TRUE - the queue is full.
  - — FALSE - the queue is not full.

#### A.4.5.5.3 ixCryptoAccCodeletDispatcherPoll (void)

This function polls the Queue manager for the poll mode set in the function ixCryptoAccCodeletDispatcherStart.

**Parameters**

None

**Returns**

None

Reentrant: no ISR Callable — no

#### A.4.5.5.4 ixCryptoAccCodeletDispatcherStart (..)

ixCryptoAccCodeletDispatcherStart (BOOL useInterrupt)

This function starts the Queue manager dispatcher.

**Parameters**

***BOOL*** [in] useInterrupt — start in interrupt or polled mode. TRUE is interrupt mode while FALSE is poll mode.

**Returns**

- IX_SUCCESS - Dispatcher successfully started.
- IX_FAIL - Error starting dispatcher.

  — Reentrant: no ISR Callable — no

#### A.4.5.5.5 ixCryptoAccCodeletInit (void)

This function is the entry point to the Crypto Access codelet. It must be called before any of the demos as it initializes the necessary component.

**Note**

The initialization sequence is as follows:

- Initialize Network Processor Engine (NPE)
- Initialize Queue manager
- Initialize Crypto Access component
- Start NPE
- Start Queue manager dispatcher.

**Parameters**

None

**Returns**

- IX_SUCCESS - Successful initialization.

- IX_FAIL - Initialization failed.

  — Reentrant: no ISR Callable — no

#### A.4.5.5.6 ixCryptoAccCodeletMbufFromQHeadRemove (..)

ixCryptoAccCodeletMbufFromQHeadRemove (IX_MBUF ** pMbufToRem)

This function will remove a mbuf from the queue. The mbuf will be removed from the head of the queue.

**Parameters**

***IX_MBUF*** ** [in] pMbufToRem — Pointer to a pointer of the mbuf to be removed from the head of the queue.

**Returns**

None.

Reentrant : yes ISR Callable — no

#### A.4.5.5.7 ixCryptoAccCodeletMbufToQTailAdd (..)

ixCryptoAccCodeletMbufToQTailAdd (IX_MBUF * pMbufToAdd)

This function will add an mbuf to the queue. The mbuf will be added to the end of the queue.

**Parameters**

***IX_MBUF*** * [in] pMbufToAdd — Pointer of the mbuf to be added to the tail of the queue.

**Returns**

None.

Reentrant: yes ISR Callable — no

#### A.4.5.5.8 ixCryptoAccCodeletMemPoolInit (void)

This function will initialize mbuf pool for the Crypto Access Codelet.

**Parameters**

None

**Returns**

- IX_SUCCESS - Successful mbuf pool initialization.
- IX_FAIL - Mbuf pool initialization failed.

  — Reentrant: no ISR Callable — no

#### A.4.5.5.9 ixCryptoAccCodeletNpeInit (.)

ixCryptoAccCodeletNpeInit (IxNpeDlNpeId npeId)

This function is to download microcode to the NPE. This function always downloads the latest crypto version of NPE microcode available.

**Parameters**

***IxNpeDlNpeId*** [in] npeId — The ID of the NPE which the microcode downloaded to.

**Returns**

- IX_SUCCESS - Successful NPE initialization.
- IX_FAIL - NPE initialization failed.

— Reentrant: no ISR Callable — no

#### A.4.5.5.10 ixCryptoAccCodeletRegisterCtx (..)

ixCryptoAccCodeletRegisterCtx (UINT32 opType)

This function will register to the Crypto Access Component for three different operation which will demonstrate the capability of the security hardware accelerator. The types of operation are: 1 - Encryption and Decryption 2 - Authentication: Calculation and Check 3 - Encryption/Decryption with authentication.

**Parameters**

***UINT32*** [in] opType — Type of operation to be registered. This parameter will only take 3 different unsigned integer as shown above in the brief, e.g. 1 for encryption and decryption only and so on.

**Returns**

IX_SUCCESS - Register successfully

IX_FAIL - Error in registering

Reentrant: no ISR Callable — no

#### A.4.5.5.11 test_main (void)

This is the main function of the Codelet where registration and perform is done. This function will also rotate between encryption/authentication and authentication/decryption.

**Returns**

integer.

Reentrant: no ISR Callable — no

## A.4.6 Demo Utils (IxDemoUtils) API

Intel® IXP42X product line Demo Utils component API.

### A.4.6.1 Functions

- PUBLIC IX_STATUS ixDemoUtilsAtmVcRegisterConnect (IxAtmLogicalPort port, unsigned vpi, unsigned vci, IxAtmdAccAalType aalType, IxAtmServiceCategory atmService, IxAtmRxQueueId rxQueueId, IxAtmdAccRxVcRxCallback rxCallback, unsigned int minimumReplenishCount, IxAtmdAccTxVcBufferReturnCallback bufferFreeCallback, IxAtmdAccRxVcFreeLowCallback rxFreeLowCallback, IxAtmdAccUserId userId, IxAtmConnId *rxConnId, IxAtmConnId *txConnId)

  Register and connect an Aal Pdu receive and transmit VC for a particular port/vpi/vci.
- PUBLIC IX_STATUS ixDemoUtilsAtmVcUnregisterDisconnect (IxAtmConnId rxConnId, IxAtmConnId txConnId)

  Disconnect and unregister an Aal transmit and receive VC.
- PUBLIC IX_STATUS ixDemoUtilsAtmAllVcsDisconnect (void)

  Disconnect and unregister all registered VCs.
- PUBLIC IX_STATUS ixDemoUtilsAtmAllVcsRemove (void)

  Disconnect and unregister all registered VCs.
- PUBLIC IX_STATUS ixDemoUtilsAtmImageDownload (unsigned numPorts, IxAtmmPhyMode *phyMode)

  Download an ATM image to the NPEA.
- PUBLIC IX_STATUS ixDemoUtilsNpeImageDownload (IxNpeDlNpeId npeId, IxNpeDlBuildId buildId)

  Download an image to the NPE.
- PUBLIC BOOL ixDemoUtilsUtopiaFpgaStimulusAsMasterSet (void)

  Configure test and stimulus FPGA clocks for Intel® IXP42X product line as ATM master.
- IX_STATUS ixDemoUtilsMbufPoolInit (void)

  Initialize the demo utils buffer pool.
- void ixDemoUtilsMbufGet (unsigned int bufSize, IX_MBUF **buf)

  Get an unchained buffer.
- void ixDemoUtilsMbufFree (IX_MBUF *buf)

  Free a buffer chain.
- void ixDemoUtilsMbufShow (void)

  Show statistics for the buffer pool; free, used buffer counts etc.

### A.4.6.2 Detailed Description

Intel® IXP42X product line Demo Utils component API.

These utilities provide support for

- adding/removing ATM VCs via the IxAtmdAcc component
- downloading the NPEA image

## A.4.6.3 Function Documentation

### A.4.6.3.1 PUBLIC IX_STATUS ixDemoUtilsAtmAllVcsDisconnect (void)

Disconnect and unregister all registered VCs.

#### Parameters

none

#### Returns

- IX_SUCCESS successful call to ixDemoUtilsAtmAllVcsDisconnect
- IX_FAIL parameter error, failed to disconnect, not initialized or some other error occurs during processing.

### A.4.6.3.2 PUBLIC IX_STATUS ixDemoUtilsAtmAllVcsRemove (void)

Disconnect and unregister all registered VCs.

#### Parameters

none

#### Returns

- IX_SUCCESS successful call to ixDemoUtilsAtmAllVcsDisconnect
- IX_FAIL parameter error, failed to disconnect, not initialized or some other error occurs during processing.

### A.4.6.3.3 PUBLIC IX_STATUS ixDemoUtilsAtmImageDownload (..)

PUBLIC IX_STATUS ixDemoUtilsAtmImageDownload (unsigned numPorts, IxAtmmPhyMode * phyMode)

Download an ATM image to the NPEA.

#### Parameters

***numports*** (in) — Number of ports in the system. A value > 1 requires a MPHY_MULTIPORT to NPEA image to be downloaded. A value of 1 can use a SPHY or MPHY NPEA image.

***imagePtr*** (in) — Pointer to image file that can contain multiple images.

***phyMode*** (out) — This parameter will be set depending on which images is available in imagePtr. See description for numports above.

#### Returns

- IX_SUCCESS successful call to ixDemoUtilsAtmNpeImagedownload
- IX_FAIL parameter error, failed to disconnect, not initialized or some other error occurs during processing.

### A.4.6.3.4 PUBLIC IX_STATUS ixDemoUtilsAtmVcRegisterConnect (..)

PUBLIC IX_STATUS ixDemoUtilsAtmVcRegisterConnect (IxAtmLogicalPort port, unsigned vpi, unsigned vci, IxAtmdAccAalType aalType, IxAtmServiceCategory atmService, IxAtmRxQueueId rxQueueId, IxAtmdAccRxVcRxCallback rxCallback, unsigned int minimumReplenishCount, IxAtmdAccTxVcBufferReturnCallback bufferFreeCallback, IxAtmdAccRxVcFreeLowCallback rxFreeLowCallback, IxAtmdAccUserId userId, IxAtmConnId * rxConnId, IxAtmConnId * txConnId)

Register and connect an Aal Pdu receive and transmit VC for a particular port/vpi/vci.

This function allows a user to connect to an Aal5 Pdu receive and transmit service for a particular port/vpi/vci. It registers the callback and allocates internal resources and a Connection Id to be used in further API calls related to this VCC.

#### Parameters

***port*** (in) VC identification — logical PHY port [0..7]

***vpi*** (in) VC identification — ATM Vpi [0..255]

***vci*** (in) VC identification — ATM Vci [0..65535]

***atmService*** (in) — type of service AAL5 **only** in ixp425 1.0

***rxQueueId*** (in) — identifies which of two Qs the VC should use when incoming traffic is processed

***rxCallback*** (in) — function called when mbufs are received. This parameter cannot be a null pointer.

***minimumReplenishCount*** (in) — number of free mbufs to be used with this channel. Use a high number when the expected traffic rate on this channel is high, or when the user's mbufs are small, or when the RxVcFreeLow Notification has to be invoked less often.

***bufferFreeCallback*** (in) — function to be called to return ownership of buffers to IxAtmDemoUtils user.

***userId*** (in) — user Id to use in callback communications

***rxConnId*** (out) — Rx Conn Id passed back from IxAtmdAcc

***txConnId*** (out) — Tx Conn Id passed back from IxAtmdAcc

#### Returns

- IX_SUCCESS successful call to ixDemoUtilsAtmVcRegisterConnect
- IX_FAIL parameter error, VC already in use or port is not initialized or some other error occurs during processing.

### A.4.6.3.5 PUBLIC IX_STATUS ixDemoUtilsAtmVcUnregisterDisconnect (..)

PUBLIC IX_STATUS ixDemoUtilsAtmVcUnregisterDisconnect (IxAtmConnId rxConnId, IxAtmConnId txConnId)

Disconnect and unregister an Aal transmit and receive VC.

#### Parameters

***rxConnId*** (in) — conn Id of the Rx VC to disconnect and unregister

***txConnId*** (in) — conn Id of the Tx VC to disconnect and unregister

#### Returns

- IX_SUCCESS successful call to ixDemoUtilsAtmVcUnregisterDisconnect
- IX_FAIL parameter error, failed to disconnect, not initialized or some other error occurs during processing.

### A.4.6.3.6 void ixDemoUtilsMbufFree (..)

void ixDemoUtilsMbufFree (IX_MBUF * buf)

Free a buffer chain.

#### Parameters

***buf*** (out) Pointer to buffer to be freed.

#### Returns

- IX_SUCCESS successful call to ixDemoUtilsMbufGet
- IX_FAIL parameter error, failed to get a buffer, or some other error occurs during processing.

### A.4.6.3.7 void ixDemoUtilsMbufGet (..)

void ixDemoUtilsMbufGet (unsigned int bufSize, IX_MBUF ** buf)

Get an unchained buffer.

#### Parameters

***bufSize*** (in) — Buffer capacity.

***buf*** (out) — Pointer to buffer; will be set the new buffer

#### Returns

- IX_SUCCESS — successful call to ixDemoUtilsMbufGet
- IX_FAIL — parameter error, failed to get a buffer, or some other error occurs during processing.

### A.4.6.3.8 IX_STATUS ixDemoUtilsMbufPoolInit (void)

Initialize the demo utils buffer pool.

#### Parameters

none

#### Returns

- IX_SUCCESS — successful call to ixDemoUtilsMbufPoolInit

- IX_FAIL — parameter error, failed initialize buffer pool or some other error occurs during processing.

#### A.4.6.3.9 void ixDemoUtilsMbufShow (void)

Show statistics for the buffer pool; free, used buffer counts etc.

**Returns**

none

#### A.4.6.3.10 PUBLIC IX_STATUS ixDemoUtilsNpeImageDownload (..)

PUBLIC IX_STATUS ixDemoUtilsNpeImageDownload (IxNpeDlNpeId npeId, IxNpeDlBuildId buildId)

Download an image to the NPE.

**Note**

The latest version which has the requested capability will be downloaded.

**Parameters**

***npeId*** (in) — Identifier of the NPE to download the image to.

***buildId*** (in) — Identifier of the build within the microCodeImage

**Returns**

- IX_SUCCESS — successful call to ixDemoUtilsNpeImageDownload
- IX_FAIL — parameter error, failed to disconnect, not initialized or some other error occurs during processing.

#### A.4.6.3.11 PUBLIC BOOL ixDemoUtilsUtopiaFpgaStimulusAsMasterSet (void)

Configure test and stimulus FPGA clocks for Intel® IXP42X product line as ATM master.

**Parameters**

none

**Returns**

- IX_SUCCESS — successful call to ixDemoUtilsUtopiaFpgaStimulusAsMasterSet
- IX_FAIL — parameter error, failed to configure stimulus FPGA, not initialized or some other error occurs during processing.

## A.4.7 DMA Access Codelet (IxAal5Demo) API

Intel® IXP42X product line demo DMA Access component API.

### A.4.7.1 Functions

- IX_STATUS ixDmaAccCodeletInit (IxDmaNpeId npeId)
  This function is the entry point to the Dma Access codelet.
- void ixDmaAccCodeletTestPerform (UINT16 transferLength, IxDmaTransferMode transferMode, IxDmaAddressingMode addressingMode, IxDmaTransferWidth transferWidth)
  This function performs a dma transfer with parameters provided by the client This function can only function after ixDmaAccCodeletInit is performed.

### A.4.7.2 Detailed Description

Intel® IXP42X product line demo DMA Access component API.

This file contains the interface of the Dma Access Codelet.

The API ixDmaAccCodeletTestPerform() allows the user to perform a Dma transfer of block size 0 to 65535 bytes between two locations in the SRAM. The user can specify any combination of the following modes.

DMA Transfer Modes 1. Copy 2. Copy and Clear Source 3. Copy with Bytes Swap 4. Copy with Bytes Reversed

DMA Addressing Modes 1. Incremental Source to Incremental Destination Addressees 2. Fixed Source to Incremental Destination Addressees 3. Incremental Source to Fixed Destination Addressees

DMA Transfer Widths 1. 32-bit Transfer 2. 16-bit Transfer 3. 8-bit Transfer 4. Burst Transfer

**Note**

The user must initialize the system with ixDmaAccCodeletInit prior to calling the function ixDmaAccCodeletiTestPerform()

### A.4.7.3 Function Documentation

#### A.4.7.3.1 IX_STATUS ixDmaAccCodeletInit (..)

IX_STATUS ixDmaAccCodeletInit (IxDmaNpeId npeId)

This function is the entry point to the Dma Access codelet.

It must be called before any of the demos as it initializes the necessary components.

**Parameters**

***npeID*** Id for the Npe to setup for Dma function

- IX_DMA_NPEID_A for Npe A
- IX_DMA_NPEID_B for Npe B

- IX_DMA_NPEID_C for Npe C

The initialization sequence is as follows:

1. Initialize Queue manager Start Queue manager dispatcher loop
2. Download NPE microcode Start NPEs
3. Initialize Dma Access component

#### Returns

IX_SUCCESS — Codelet successfully initialized

IX_FAIL - Error initializing codelet

### A.4.7.3.2 void ixDmaAccCodeletTestPerform (..)

void ixDmaAccCodeletTestPerform (UINT16 transferLength, IxDmaTransferMode transferMode, IxDmaAddressingMode addressingMode, IxDmaTransferWidth transferWidth)

This function performs a dma transfer with parameters provided by the client This function can only function after ixDmaAccCodeletInit is performed.

#### Parameters

***transferLength*** 1 to 65535 bytes

***transferMode***

- 0 = Copy
- 1 = Copy and Clear Source
- 2 = Copy and Byte Swap
- 3 = Copy and Reverse

***addressMode***

- 0 = Increment Source Addr and Increment Destination Addr");
- 1 = Increment Source Addr and Fix Destination Addr");
- 2 = Fix Source Addr and Increment Destination Addr");

***transferWidth*** :

- 0 = 32 bit src, 32 bit dest
- 1 = 32 bit src, 16 bit dest
- 2 = 32 bit src, 8 bit dest
- 3 = 16 bit src, 32 bit dest
- 4 = 16 bit src, 16 bit dest
- 5 = 16 bit src, 8 bit dest
- 6 = 8 bit src, 32 bit dest
- 7 = 8 bit src, 16 bit dest

- 8 = 8 bit src, 8 bit dest
- 9 = 8 bit src, Burst dest
- 10 = 16 bit src, Burst dest
- 11 = 32 bit src, Burst dest
- 12 = Burst src, 8 bit dest
- 13 = Burst src, 16 bit dest
- 14 = Burst src, 32 bit dest
- 15 = Burst src, Burst des

**Returns**

none

## A.4.8 Ethernet Aal5 (IxEthAal5App) API

Intel® IXP42X product line demo Ethernet AAL5 driver component API.

### A.4.8.1 Defines

- #define IX_EAA_NUM_BUFFERS_PER_ETH 64
  This is the number of buffers, which can be stored in free buffer queue for each ethernet port.
- #define IX_EAA_NUM_ATM_PORTS 8
  Define number of supported atm ports by this application.
- #define IX_EAA_NUM_BUFFERS_PER_VC 16
  This is the number of buffers per atm port.
- #define IX_EAA_PORT1_VPI 1
  Define default VPI and VCI for 8 ports. User can define them in EthAal5User.h file to overwrite definitions below.
- #define IX_EAA_PORT1_VCI 100
- #define IX_EAA_PORT2_VPI 2
- #define IX_EAA_PORT2_VCI 100
- #define IX_EAA_PORT3_VPI 3
- #define IX_EAA_PORT3_VCI 100
- #define IX_EAA_PORT4_VPI 4
- #define IX_EAA_PORT4_VCI 100
- #define IX_EAA_PORT5_VPI 5
- #define IX_EAA_PORT5_VCI 100
- #define IX_EAA_PORT6_VPI 6
- #define IX_EAA_PORT6_VCI 100
- #define IX_EAA_PORT7_VPI 7

- #define IX_EAA_PORT7_VCI 100
- #define IX_EAA_PORT8_VPI 8
- #define IX_EAA_PORT8_VCI 100

### A.4.8.2 Functions

- int ixEAAMain (void)

  Initialize MAC data base, to contain 0xffs - no valid Mac addresses present.Also initialize QMGR, NpeMh, Eth phys - 100Mbit, FULL DUPLEX, NO AUTONEGOTIATION (User can change those settings accordingly to required configuration), ATM, and UTOPIA interface.

- int ixEAAShow (void)

  display the current counters.

- void ixEAAAddMAC (int port, UINT8 mac1, UINT8 mac2, UINT8 mac3, UINT8 mac4, UINT8 mac5, UINT8 mac6)

  This function can be used to add MAC address to Atm data base and assign it to particular port.

### A.4.8.3 Detailed Description

Intel® IXP42X product line demo Ethernet Aal5 driver component API.

### A.4.8.4 Define Documentation

#### A.4.8.4.1 #define IX_EAA_NUM_ATM_PORTS 8

Define number of supported atm ports by this application.

User can define them in EthAal5User.h file to overwrite definitions below

#### A.4.8.4.2 #define IX_EAA_NUM_BUFFERS_PER_VC 16

This is the number of buffers per atm port.

User can define them in EthAal5User.h file to overwrite definitions below

### A.4.8.5 Function Documentation

#### A.4.8.5.1 void ixEAAAddMAC (..)

void ixEAAAddMAC (int port, UINT8 mac1, UINT8 mac2, UINT8 mac3, UINT8 mac4, UINT8 mac5, UINT8 mac6)

This function can be used to add MAC address to Atm data base and assign it to particular port.

Valid ports are 0-7

#### A.4.8.5.2 int ixEAAMain (void)

Initialize MAC data base, to contain 0xffs - no valid Mac addresses present.Also initialize QMGR, NpeMh, Eth phys - 100Mbit, FULL DUPLEX, NO AUTONEGOTIATION (User can change those settings accordingly to required configuration), ATM, and UTOPIA interface.

If Linux is used, use interrupt mode - much faster under Linux than polling

If vxWorks is used use poll mode - much faster under vxWorks than interrupts and start background QMgr queues poll

## A.4.9 Ethernet Access Codelet (IxEthAccCodelet) API

Intel® IXP42X product line demo Ethernet Access component API.

### A.4.9.1 Data Structures

- struct IxEthAccCodeletPhyConf

### A.4.9.2 Defines

- #define IX_ETHACC_CODELET_NPE1_MAC {{0x2, 0x0, 0xa, 0xb, 0xc, 0xd}}
  Hard-encoded MAC address for NPE2.
- #define IX_ETHACC_CODELET_NPE2_MAC {{0x2, 0x0, 0xe, 0xf, 0xa, 0xb}}
  Recommended size of receive MBuf pool.
- #define IX_ETHACC_CODELET_RX_MBUF_POOL_SIZE 128
  Recommended size of transmit MBuf pool.
- #define IX_ETHACC_CODELET_TX_MBUF_POOL_SIZE 128
  Number of Ports supported for this demo.
- #define IX_ETHACC_CODELET_MAX_PORT IX_ETH_ACC_NUMBER_OF_PORTS
  Size of MBuf pool.
- #define IX_ETHACC_CODELET_MBUF_POOL_SIZE
  Size of MBuf packet.
- #define IX_ETHACC_CODELET_PCK_SIZE IX_ETHACC_RX_MBUF_MIN_SIZE
  Length of MBuf payload (in bytes).
- #define IX_ETHACC_CODELET_PCK_LEN 1514
  Size of MBuf data pool.
- #define IX_ETHACC_CODELET_MBUF_DATA_POOL_SIZE (IX_ETHACC_CODELET_MBUF_POOL_SIZE * IX_ETHACC_CODELET_PCK_SIZE)
  Size of packets for TxGenRxSink demo.
- #define IX_ETHACC_CODELET_TXGEN_PCK_LEN 60
  Number of packets to generate for the TxGenRxSink demo.
- #define IX_ETHACC_CODELET_TXGEN_PCKS 128
  Verify payload for TxGenRxSink demo - defining this will affect performance.

- #define IX_ETHACC_CODELET_TXGENRXSINK_VERIFY

### A.4.9.3 Functions

- IX_STATUS ixEthAccCodeletInit (void)
  This function is the entry point to the Ethernet Access codelet.
- IX_STATUS ixEthAccCodeletRxSink (void)
  Receive sink demo.
- IX_STATUS ixEthAccCodeletSwLoopback (void)
  Software loopback demo.
- IX_STATUS ixEthAccCodeletTxGenRxSinkLoopback (void)
  Port-to-Port (Tx to Rx Sink) loopback demo.
- IX_STATUS ixEthAccCodeletBridge (void)
  Bridge demo.
- IX_STATUS ixEthAccCodeletDBLearning (void)
  Ethernet MAC address learning demo.

void ixEthAccCodeletShow (void)

### A.4.9.4 Detailed Description

Intel® IXP42X product line demo Ethernet Access component API.

This file contains the interface of the Ethernet Access Codelet.

Demos — The following top-level scenarios are supported:

Demonstrate Ethernet Data and Control plane services by:

- Configuring both ports as a receiver sink from an external source (such as SmartBits*).
- Connecting a port to external test equipment performing a loopback.
- Configuring and performing a software loopback on each of the two ethernet ports.
- Configuring both ports to act as a bridge so that frames received on one port are retransmitted on the other.

Demonstrate Ethernet Management services by:

- Adding and removing static/dynamic entries.
- Calling the maintenance interface (shall be run as a separate background task)
- Calling the show routine to display the MAC address filtering tables.
- All demos will be conducted using different packet sizes and varying source MAC addresses. Where external test equipment is used to transmit data it should verify that data received is the same as that transmitted.
- A list of statistics will be available for all demos and shall be display-able by using the Codelet's show function.

#### MAC Setup

The default MAC setup will be:

- Promiscuous mode enabled (for learning example)
- Frame Check Sequence appended for all frames generated on the Intel XScale core.

#### PHY Setup

This demo uses two PHYs as defined by IX_ETHACC_CODELET_MAX_PHY. The default PHY setup will be:

- 100Mbits,
- full duplex,
- auto-negotiation on.

#### Loopback

In the loopback scenarios the Codelet will offer the facility to perform a loopback.

The external test equipment will create and transmit packets, and ensure that received packets match those transmitted. The Codelet will re-transmit any received packets.

In the scenario where two ports are involved in the loopback the received packets will be passed to the alternative port much like a bridge would do.

#### Test Equipment

The test harness will consist of external test equipment capable of generating Ethernet packets (e.g. SmartBits).

The test equipment must be capable of performing at least the following actions to support the scenarios outlined for the Codelet:

- Send/receive an Ethernet data stream.
- Send/receive frames of different length.
- Detect CRC errors.
- Append FCS.
- Support 100Mbit full duplex mode.

## A.4.9.5 Define Documentation

### A.4.9.5.1 #define IX_ETHACC_CODELET_MBUF_POOL_SIZE

Value:

```
((IX_ETHACC_CODELET_RX_MBUF_POOL_SIZE + IX_ETHACC_CODELET_TX_MBUF_POOL_SIZE) \
  * IX_ETHACC_CODELET_MAX_PORT)
```

Size of MBuf packet.

## A.4.9.6 Function Documentation

### A.4.9.6.1 IX_STATUS ixEthAccCodeletBridge (void)

Bridge demo.

This demo transmits any frames received on one port through the other one, as a bridge would do. An external Ethernet source must be connected to both ports transmitting frames.

#### Returns

- IX_SUCCESS — Bridge demo successfully started.
- IX_FAIL — Error starting bridge demo.

### A.4.9.6.2 IX_STATUS ixEthAccCodeletDBLearning (void)

Ethernet MAC address learning demo.

This function demonstrates the use of the Ethernet MAC learning facility. It adds some static and dynamic entries. Dynamic entries are then aged and verified that they no longer appear in the database.

#### Returns

- IX_SUCCESS — DB Learning demo successfully started
- IX_FAIL — Error starting DB Learning demo.

### A.4.9.6.3 IX_STATUS ixEthAccCodeletInit (void)

This function is the entry point to the Ethernet Access codelet.

It must be called before any of the demos as it initializes the necessary components.

The initialization sequence is as follows:

- Initialize Queue manager
- Start Queue manager dispatcher loop
- Download NPE microcode
- Start NPE message handler
- Start NPEs
- Initialize Ethernet Access component
- Initialize Ethernet ports
- Initialize Ethernet PHYs
- Program MAC addresses
- Set ports to promiscuous mode
- Initialize MBUF pool
- Set the Tx scheduling priority to FIFO_NO_PRIORITY
- At the end of the initialization sequence ports are not enabled. They are enabled as appropriate for each demo that's run.

**Returns**

- IX_SUCCESS — Codelet successfully initialized
- IX_FAIL — Error initializing codelet

#### A.4.9.6.4 IX_STATUS ixEthAccCodeletRxSink (void)

Receive sink demo.

This function sinks received packets as fast as possible on both ports. An external source must be connected to both ports transmitting Ethernet packets.

**Returns**

- IX_SUCCESS — Rx Sink demo successfully started
- IX_FAIL — Error starting Rx Sink demo

#### A.4.9.6.5 IX_STATUS ixEthAccCodeletSwLoopback (void)

Software loopback demo.

This function sets up some callbacks so that any frames received are looped back and retransmitted unmodified on the same port. An external Ethernet source must be connected to both ports transmitting packets.

**Returns**

- IX_SUCCESS - Software Loopback demo successfully started
- IX_FAIL - Software started external Loopback demo

#### A.4.9.6.6 IX_STATUS ixEthAccCodeletTxGenRxSinkLoopback (void)

Port-to-Port (Tx to Rx Sink) loopback demo.

This function performs a software loopback on the two ethernet ports. Frames are generated on the Intel XScale core and sent from port 1 and received on port 2 via a cross-over cable. Received frames on port 2 are put back on the free queue which is shared between the two ports. A verify function ensures that the frames transmitted are the same as those received. A cross-over cable connecting port 1 and port 2 is needed to perform this demo.

**Returns**

- IX_SUCCESS - SW Loopback successfully started
- IX_FAIL - Error starting SW loopback demo

## A.4.10 Fast Path Access Codelet (IxFpathAccCodelet) API

Intel® IXP42X product line demo Fast Path Access component API.

### A.4.10.1 Functions

- void ixFpCodeletEthBDemoStart (void)
  Start the Single VC Ethernet Bridge Fast Path Demo.

- void ixFpCodeletEthBDemoStop (void)
  Stop the Single VC Ethernet Bridge Fast Path Demo.
- void ixFpCodeletRoutedIpDemoStart (void)
  Start the Single VC Routed Ip Fast Path Demo.
- void ixFpCodeletRoutedIpDemoStop (void)
  Stop the Single VC Routed IP Fast Path Demo.
- void ixFpathAccCodeletShow (void)
  Display Demo Config and Statistics Per VC.

### A.4.10.2 Detailed Description

Intel® IXP42X product line demo Fast Path Access component API.

This module contains the implementation of the FPATH Access Codelet.

The following top-level scenarios are supported:

- Provide a Ethernet Bridge fast path Demo on a single VC.
- Configures the system to pass Ethernet frames with a given RFC1483 (LLC/VCMUX) encapsulation on the fast path from the Utopia to a single Ethernet port.
- Provide example classifier and modifier templates for LLC and VCMUX.
- Simple demonstration of the Wan Mac address learning functionality.
- The Utopia port is connected to a piece of test equipment capable of generating encapsulated ethernet Frames.
- The Ethernet port is connected to a piece of test equipment capable of accepting and verifying ethernet Frames.
- The ATM test equipment will inject traffic on the UTOPIA and the Ethernet test equipment will ensure the integrity of the traffic. No packet loss should be observed. The learned Mac address database will be displayed periodically on standard out.
- Since Ethernet Frame sequencing between hosts must be observed all packets are conveyed on the Fast Path. Only errored packets are set to the slow path and at that are dropped.
- No Ethernet to ATM traffic will be supported.
- Provide a Ip forwarding fast path demo on a single VC.
- Configures the system to pass Routed IP packet with a given RFC1483 (LLC/VCMUX) encapsulation on the fast path from the UTOPIA to a single Ethernet port.
- Provide example classifier and modifier templates for Routed IP LLC and VCMUX.
- Simple demonstration of how IP Flow detection and aging can be implemented.
- The UTOPIA port is connected to a piece of test equipment capable of generating encapsulated IP packets.
- The Ethernet port is connected to a piece of test equipment capable of accepting ethernet frames containing IP packets.
- The ATM test equipment will inject traffic on the UTOPIA and the Ethernet test equipment will ensure the integrity of the traffic. No packet loss should be observed.

- Initially packets will flow on the slow path until the demonstration learning and aging function identifies Ip packet flows. As flows are identified and aged the codelet will provide feedback on standard out.
- Ip packet forwarding is provided in the ATM to Ethernet direction only and will not involve an IP stack. Packets travelling on the slow path will be forwarded to a hard coded Ethernet address over the Ethernet port.
- Further scenarios to include other encapsulation types are considered outside the scope of this demonstration Codelet. It will, however, be a design goal of this Codelet to be re-usable in such extended scenarios e.g. NAPT, PPPoe.
- For the current release only one UTOPIA port will be supported. However, the Codelet will be developed in such a way that the scenarios are easily extendible to use up to 12 ports.

### Test Equipment

- **Atm** — The test harness will consist of external test equipment capable of generating RFC1483 encapsulated Ethernet and Ip traffic over ATM. It should provide UTOPIA connectivity and be capable of operating in MPhy, UTOPIA Slave mode.
- **Ethernet** — The Ethernet test equipment should be attached to the IX_ETH_PORT_1. By default both ethernet ports are brought up with auto negotiation enabled. The test equipment should have the capability to count and inspect Ethernet frames and should ideally support 100Meg Ethernet.
- **Buffer Management** — The demo code will use a single buffer pool for supply of all buffers to IxAtmdAcc and IxFpathAcc. The pool will contain a selection of buffer sizes, the dimensions of which are available in IxFpathAccCodeletBufMan.c.
- **Buffers and Caching** — The allocation of buffers uses the MACROS available in IxOsCacheMMU.h. If these routines are populated will calls to the cache driver then buffer memory will be allocated from cache. Special attention is made in the demo code to invalidating and flushing cached memory when manipulating packets within buffers on the datapath.

  More specifically the data portion of buffers manipulated by the Intel XScale core are flushed before transmit and invalidated after receive. Flushing the buffers before transmit ensures the NPE reads and transmits the correct data. Invalidating the buffers after receive ensures the Intel XScale core reads and processes the correct data. In the case of the Codelet, all data is flushed and invalidated as every byte is being written to on transmit and every byte is verified on receive. In a real application flushing or invalidating all the data may not be necessary, only the data that the application has written before transmit or will read after receive. Note, regarding the packetised service, the IxHssAcc component itself takes care of flushing and invalidating mbuf headers. The application needs only to concern itself with the mbuf data.
- **IxFpathAcc** — Buffers will be supplied at initialization to IxFpathAcc to populate the fast path buffer free pool. No further buffers need be supplied to IxFpathAcc there after.
- **IxAtmdAcc** — Buffers will be supplied at initialization to IxFpathAcc to populate the rx free Q for each active VC. Buffers will be further replenished to each rx VC free Q each time a Q becomes empty.

### Notes

Each buffer has space reserved at the beginning to allow header information to be prepended. An example of when this might be used is in the routing demo where IP packets are prepended with ethernet frame headers before being submitted to the Ethernet driver.

**Connection Database** — The demo codelet maintains a simple connection database so that multiple VCs can be enabled simultaneously. Each vc has a VcNumber which identifies it. The codelet is design to support a maximum of 8 VCs. The VcNumber is set as the userId in any per VC callbacks set on IxFpathAcc or IxAtmdAcc.

**System Configuration** — The following is the static configuration of the system for both bridged and routed demos. See IxFpathAccCodeletDefines.h for corresponding definitions for some of the items below.

- UTOPIA Interface Mode = MPhy, Bus Master.
- Utopia PHY Addr = 0,
- ATM Rx Free Q Buffer Replenishment = Triggered on Rx Free Q Empty.
- Ethernet Port = IX_ETH_PORT_1.
- Ethernet Tx Priority = IX_ETH_ACC_TX_DEFAULT_PRIORITY.
- Ethernet PHY Config = 100Meg, Full Duplex, Auto-negotiate.
- Ethernet Port Mac Addr= 00:00:00:00:00:01
- QMgr Dispatch = Executed from task (Priority = IX_OSSL_THREAD_PRI_LOW)

**Endianness** — The demo codelet assumes that IP and MAC addresses are formatted in network byte order. A UINT32 containing the value 0x12345678 represents the IP address 12.34.56.78. A UINT8 array containing the sequence {0x01,0x02,0x03,0x04,0x05,0x06} represents the MAC address 01:02:03:04:05:06.

### Source Code Overview

Brief description of the functionality of each of the src code demo files

- **IxFpathAccCodeletMain.c:**
  - Contains the main codelet initialization routine.
  - Implements the main show routine.
  - Maintains the codelet connection table and statistics.
  - Provides the primary ATM receive callback which strips aal5 PDU encapsulation and forwards the packet to an encapsulation specific handler in either the router or bridge demo files.
  - Demonstrates the following IxFpathAcc API:-
    - ixFpathAccInit()
    - ixFpathAccBufferSupply()
    - ixFpathAccBufferPoolInfoGet()
    - ixFpathAccStatisticsGet()
- **IxFpathAccCodeletTemplates.c: template generation utilities.**
  - Contains the base templates for each the support encapsulation types.
  - Engine to construct, store and print encapsulation specific templates for each connection.
- **IxFpathAccCodeletRoutedDemo.c:** Routed VC demo.
  - Flow learning and aging example.
  - Demonstrates IxFpathAcc API:-

    - ixFpathAccTemplatePairSet()
- **IxFpathAccCodeletBridgeDemo.c:** Bridged VC demo
  - Wan Mac learning example.
  - Demonstrates IxFpathAcc API:-
    - ixFpathAccTemplatePairSet()
- **IxFpathAccCodeletBufMan.c:**
  - Buffer pool creation.
  - Buffer Alloc and Free routines.
- **IxFpathAccCodeletUtils.c:**
  - Configuration of s/w components other than IxFpathAcc.
  - Routines to init and connect/disconnect to/from ATM and UTOPIA drivers
  - Routines to init and connect to the Ethernet Driver
  - Routines to Download Npe Firmware
- **IxFpathAccCodeletFPGA.c:**
  - Fpga stimulus setup - to provide a clock src.

### A.4.10.3 Function Documentation

#### A.4.10.3.1 void ixFpathAccCodeletShow (void)

Display Demo Config and Statistics Per VC.

This function displays the configuration and statistics for each active VC. It also demonstrates the use of the ixFpathAccStatisticsGet API function.

#### A.4.10.3.2 void ixFpCodeletEthBDemoStart (void)

Start the Single VC Ethernet Bridge Fast Path Demo.

This function initializes the s/w components, sets up a single ATM VC and configures the NPEs with a pair of fast path templates which support the transportation of encapsulated ethernet frames on the Fast Path. Incoming Encapsulated Ethernet Frames are classified on the ATM NPE and are directly passed to ETH NPE-B where they are modified and sent over the Ethernet port.

The demo automatically configures the classifier template with the WAN MAC learning opcode so that a MAC database is populated with a log of the last 16 (default) MAC addresses notified by the ATM NPE.

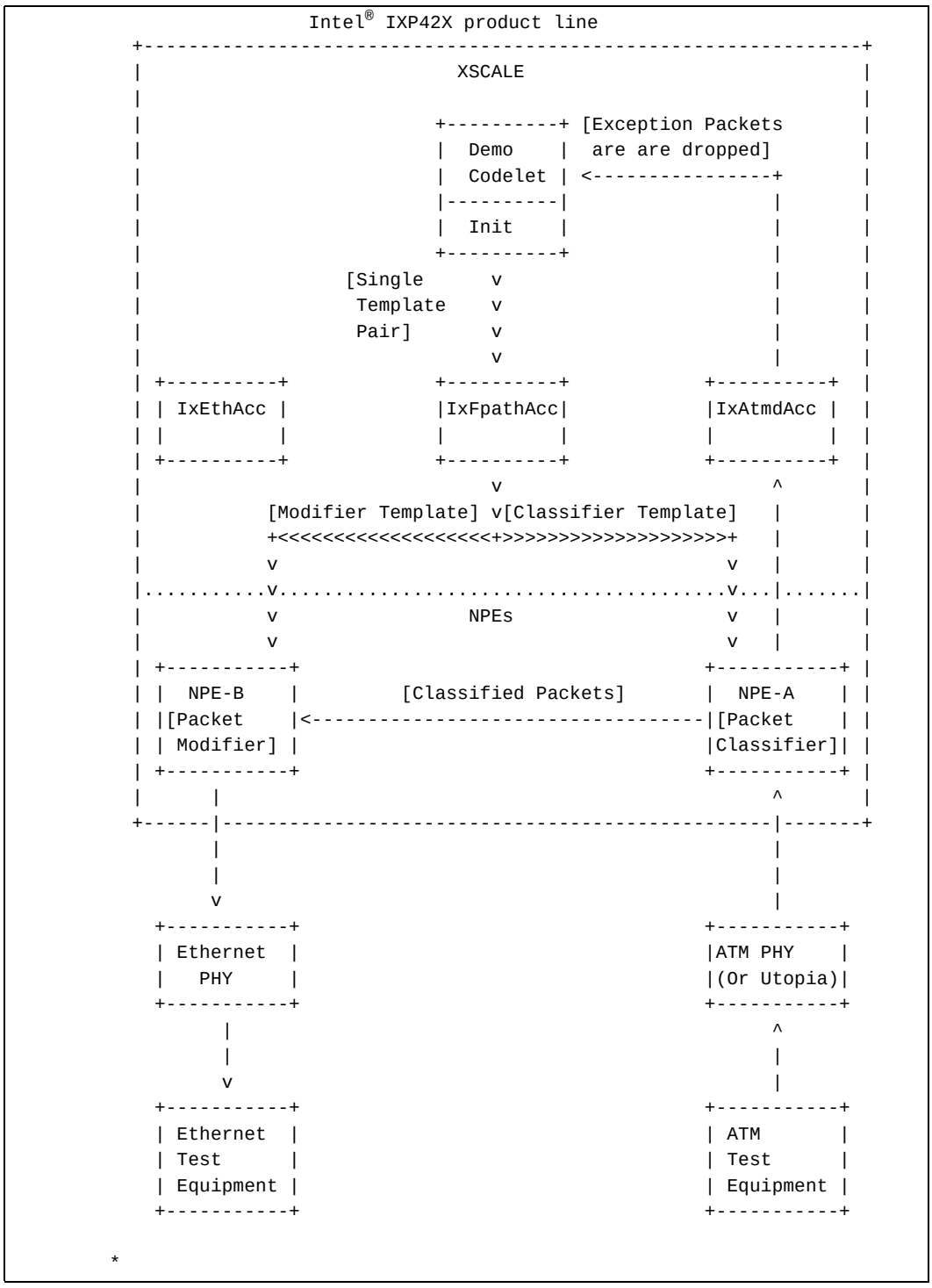
Packets which fail classification are dropped in the codelet and the buffer(s) associated with the packet is recycled to the buffer pool.

Once this function completes, packets can be injected via the ATM equipment and will be received on the Ethernet test equipment.

- **Configuration**
  - VPI = 1
  - VCI = 32

- VcNumber = 0 (codelet connection number)
- Encapsulation = VCMUX
- Max Encapsulated Ethernet Frame Size = 1516.
- Slow Path Packet Policy = All slow path packets dropped.
- Wan Mac Addr database size = 16
- Wan Mac Addr database display rate = 5sec.

- **Bridge Data Flow:**



#### A.4.10.3.3 void ixFpCodeletEthBDemoStop (void)

Stop the Single VC Ethernet Bridge Fast Path Demo.

This function demonstrates the correct procedure for shutting down a fast path VC in conjunction with disconnecting the VC from IxAtmdAcc.

#### A.4.10.3.4 void ixFpCodeletRoutedIpDemoStart (void)

Start the Single VC Routed Ip Fast Path Demo.

This function initializes the s/w components, sets up a single ATM VC and configures the NPEs with a pair of fast path templates which support fast path transportation of encapsulated Ip packets from aal5 to ethernet.

Initially all IP packets will travel on the slow path until such time as a packet flow is determined. The automatic flow learning example function reconfigures the classifier/modifier templates to allow packets for learned destination IP address be forwarded on the fast path. The fast path functionality supports 4 Ip flows per VC and this is demonstrated.

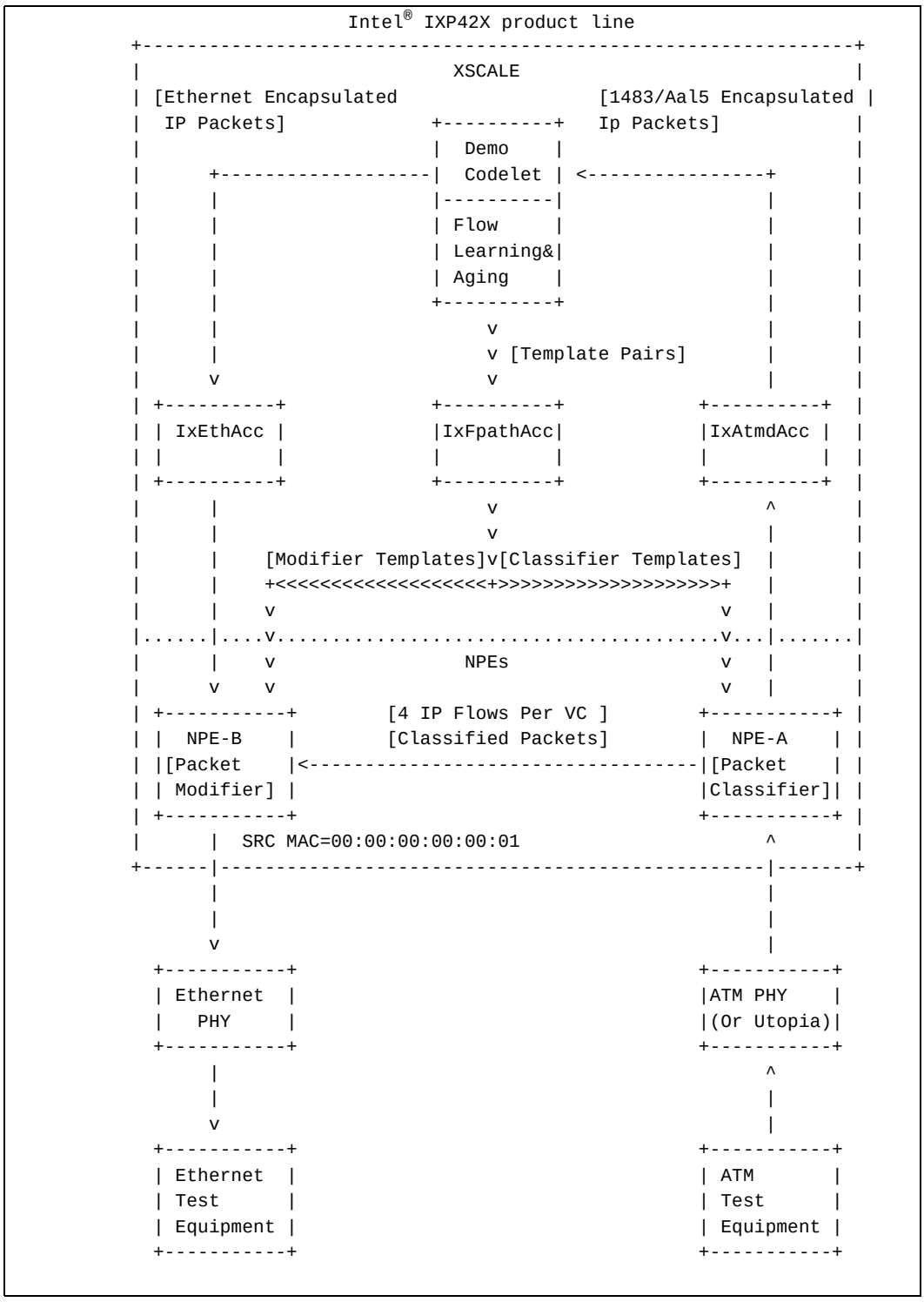
Encapsulated IP packets are classified on the ATM NPE and are directly passed to ETH NPE-B where they are placed in Ethernet Frames and sent over the Ethernet port.

Packets which fail classification continue to the learning function and are subsequently forwarded to the Ethernet driver (IxEthAcc) to a single fixed MAC address.

Once this function completes, packets can be injected via the ATM equipment and will be received on the Ethernet test equipment.

- **Configuration;**
  - VPI = 1
  - VCI = 33
  - VcNumber = 1 (codelet connection number)
  - Encapsulation = LLC/VCMUX (default).
  - Max Encapsulated Ethernet Frame Size = Based on encapsulation.
  - Slow Path Packet Policy = Packets have RFC1483/Aal5 encapsulation removed to IxEthAcc and are reencapsulated in Ethernet frames then forwarded to IxEthAcc.
  - Destination Mac for all IP packets = 00:00:00:00:00:03

- **Routed Data Flow:**



#### A.4.10.3.5 void ixFpCodeletRoutedIpDemoStop (void)

Stop the Single VC Routed IP Fast Path Demo.

This function demonstrates the correct procedure for shutting down a fast path VC in conjunction with disconnecting the VC from IxAtmdAcc.

## A.4.11 HSS Access Codelet (IxHssAccCodelet) API

Intel® IXP42X product line demo HSS Acc component API. The interface for the HSS Access Codelet.

### A.4.11.1 Modules

- Intel® IXP42X product line HSS Access Codelet Packetised (IxHssAccCodeletPkt) API
  The interface for packetised implementation of the HSS Access Codelet.
- Intel® IXP42X product line HSS Access Codelet MBuf Pool (IsHssAccCodeletMbuf) API
  The interface for mbuf pool implementation of the HSS Access Codelet.
- Intel® IXP42X product line HSS Access Codelet Config (IxHssAccCodeletConfig) API
  The interface for config implementation of the HSS Access Codelet.
- Intel® IXP42X product line HSS Access Codelet Common (IxHssAccCodeletCom) API
  The interface for common implementation of the HSS Access Codelet.
- Intel® IXP42X product line HSS Access Codelet Channelised (IxHssAccCodeletChan)
  The interface for channelised implementation of the HSS Access Codelet.

### A.4.11.2 Functions

- void ixHssAccCodeletInit (BOOL runForever)
- void ixHssAccCodeletConfigure (IxHssAccHssPort hssPortId)
- void ixHssAccCodeletPacketisedServiceStart (IxHssAccHssPort hssPortId, IxHssAccHdlcPort hdlcPortId)
- void ixHssAccCodeletPacketisedServiceRun (IxHssAccHssPort hssPortId, IxHssAccHdlcPort hdlcPortId)
- void ixHssAccCodeletPacketisedServiceStop (IxHssAccHssPort hssPortId, IxHssAccHdlcPort hdlcPortId)
- void ixHssAccCodeletPacketisedVerifySet (BOOL verifyOn)
- void ixHssAccCodeletChannelisedServiceStart (IxHssAccHssPort hssPortId)
- void ixHssAccCodeletChannelisedServiceRun (IxHssAccHssPort hssPortId)
- void ixHssAccCodeletChannelisedServiceStop (IxHssAccHssPort hssPortId)
- void ixHssAccCodeletChannelisedVerifySet (BOOL verifyOn)
- void ixHssAccCodeletShow (IxHssAccHssPort hssPortId)
- void ixHssAccCodeletShowReset (IxHssAccHssPort hssPortId)
- BOOL ixHssAccCodeletCodeletLoopbackGet (void)

This function returns the Codelet loopback state.

- void ixHssAccCodeletCodeletLoopbackSet (BOOL codeletLoopback)
- BOOL ixHssAccCodeletHssLoopbackGet (void)
  This function returns the HSS loopback state.
- void ixHssAccCodeletHssLoopbackSet (BOOL hssLoopback)

### A.4.11.3 Detailed Description

Intel® IXP42X product line demo HSS Acc component API. The interface for the HSS Access Codelet.

This module contains the implementation of the HSS Access Codelet.

The following top-level scenarios are supported:

- Test Packetised and Channelised Services, with the Codelet acting as data source/sink and the hardware performing a loopback:
  - Configuring the HSS port for loopback.
  - Connecting the HSS port to external test equipment performing a loopback.
- The Codelet will transmit data and verify that data received is the same as that transmitted.
  Test Packetised and Channelised Services, with the Codelet performing a loopback and the hardware acting as data source/sink:
- Connecting the HSS port to external test equipment acting as data source/sink.
- The external test equipment will transmit data and verify that data received is the same as that transmitted.
- Further scenarios may be conceived such as where traffic can be routed between another port (e.g. Ethernet) and the HSS port to allow extended test scenarios. However these scenarios are considered outside the scope of this Codelet. It will, however, be a design goal of this Codelet to be re-usable in such extended scenarios, where more than one Codelet may be used together in combination.
- For the current release only one HSS port will be supported. However, the Codelet will be developed in such a way that the scenarios are easily extendible to use two HSS ports when the second HSS port is supported.

#### Loopback

- To support the top-level scenarios the Codelet will offer the facility to perform a loopback, while the hardware acts as data source/sink.
  The external test equipment will create and transmit packets, and ensure that received packets match those transmitted. The Codelet will re-transmit any received packets.
  The external test equipment will create and transmit channelised data, and ensure that received channelised data match those transmitted. The Codelet will re-transmit any received channelised data.

#### Data Source/Sink

To support the top-level scenarios the Codelet will offer the facility to act as data source/sink, while the hardware performs a loopback.

The Codelet will create and transmit packets, and ensure that received packets match those transmitted. The external test equipment will re-transmit any received packets.

The Codelet will create and write channelised data to the TX buffer, and ensure that received channelised data in the RX buffer matches that transmitted. The channelised data will consist of a unique, changing bit pattern for each channel. The external test equipment will re-transmit any received channelised data.

### Test Equipment

The test harness will consist of external test equipment capable of generating T1/E1 streams of data (e.g. Abacus*, Fireberd*) and a quad T1/E1 framer to allow the concentration of four individual T1/E1 streams into one data stream and the connection of this stream to the HSS port.

The test equipment must be capable of performing at least the following actions to support the scenarios outlined for the Codelet:

- Create a T1/E1 data stream.
- Send pre-defined channelised data on a specified channel of the T1/E1 data stream.
- Receive and validate channelised data from a specified channel of the T1/E1 data stream.
- Loop back channelised data received over the same channels in the T1/E1 data stream.
- Send both HDLC and RAW packetised data over specified channels of the T1/E1 data stream.
- Receive and validate both HDLC and RAW packetised data from specified channels of the T1/E1 data stream.
- Loop back both HDLC and RAW packetised data over the same channels in the T1/E1 data stream.

### Buffer Management

The packetised service uses mbuf buffers to store data, and chains mbufs together to form large packets. In the transmit direction, mbufs are allocated from a pool on transmit, and returned to the pool on transmit done. For receive, mbufs are allocated from a pool when supplying buffers to the free queue, and returned to the pool on receive.

The channelised service operates quite differently. As voice data is very sensitive to latency using mbufs for transferal of the data between Intel XScale core and NPE is not very appropriate. Instead, circular buffers are used whereby the NPE reads data from a block of SDRAM that the Intel XScale core writes to, and writes data to a block of SDRAM that the Intel XScale core reads from. On receive, the NPE writes directly into a circular buffer that the Intel XScale core subsequently reads the data from. Each channel has its own circular buffer, and all these buffers are stored contiguously. On transmit, the NPE takes a circular list of pointers from the Intel XScale core and transmits the data referenced by these pointers. Each list of pointers contains a pointer for each channel, and the circular list of pointers contains multiple lists stored contiguously. This is to allow the Intel XScale core to transmit voice samples without having to copy data, as only the pointer to the data blocks needs to be written to SDRAM. The NPE lets the Intel XScale core know, in the form of Tx and Rx offsets, where in the blocks of SDRAM it is currently reading from and writing to. This enables the Intel XScale core to coordinate its reading and writing activities to maintain the data flow. The Tx offset lets the Intel XScale core know the list offset into the Tx circular pointer list that the NPE will next use to transmit. The Rx offset lets the Intel XScale core know the byte offset into each channel's Rx circular buffer that the NPE will next receive data into.

### Caching

To improve system performance, caching may be enabled for both the channelised and packetised buffers. To allow for this, buffers need to be flushed before transmit, and invalidated after receive. Flushing the buffers before transmit ensures the NPE reads and transmits the correct data. Invalidating the buffers after receive ensures the Intel XScale core reads and processes the correct data. In the case of the Codelet, all data is flushed and invalidated as every byte is being written to on transmit and every byte is verified on receive. In a real application flushing or invalidating all the data may not be necessary, only the data that the application has written before transmit or will read after receive. Note, regarding the packetised service, the IxHssAcc component itself takes care of flushing and invalidating mbuf headers. The application needs only to concern itself with the mbuf data.

### Data Verification Strategy

For both the packetised and channelised service a changing pattern will be transmitted. When the HSS co-processor is performing a loopback the data received is expected to be the same as that transmitted. The data transmitted carries a byte pattern that begins at a known value and is incremented for each byte. An independent byte pattern is transmitted for each channel of the channelised service, and also for each port of the packetised service. When data is received it is expected to match the pattern that was transmitted. For the channelised service the first non-idle byte received is expected to be the beginning of the byte pattern. For the packetised service, RAW mode clients may receive idle data so this is detected and ignored. Only non-idle data is verified.

## A.4.11.4 Function Documentation

### A.4.11.4.1 void ixHssAccCodeletChannelisedServiceRun (..)

void ixHssAccCodeletChannelisedServiceRun (IxHssAccHssPort hssPortId)

### Parameters

***IxHssAccHssPort*** hssPortId (in) — the HSS port ID (0 or 1).

Transmit traffic via the TX buffer and receive traffic via the RX buffer. No action needs to be taken to transmit or receive data. Once the Channelised Service is started the NPE will automatically transmit data from the TX buffer and receive data into the RX buffer.

The callback routine ixHssAccCodeletChanRxCallback() will be called at the sampling rate to tell the Codelet where data is currently being transmitted from the TX buffer and received into the RX buffer. This callback will maintain statistics of data transmitted and received and number of HSS errors.

When the Codelet is acting as data source/sink the data in the TX buffer will be prepared for transmission. This will involve writing a unique, changing bit pattern to each channel. The data in the RX buffer will be verified to ensure it matches the data transmitted (the hardware is performing a loopback).

For transmitting data, the strategy used will be to write tx data to slots we know the NPE has finished transmitting from. As the NPE transmits data and moves on to the next set of pointers, we will fill in new data in the set of pointers just transmitted. In this way the NPE will always be transmitting new data. Note, this is an artificial scenario as we always have tx data available without any delay. In a real application data should be transmitted when available, while ensuring

that the NPE isn't too far behind (causing latency) or too close behind (trying to transmit when there is no data ready). Also, as we are creating the tx data ourselves we can simply use one large tx buffer and pass pointers to this buffer to the NPE.

For receiving data, the NPE tells us where data is currently being received to. As we want to handle voice data as samples, we wait for a full sample to be received before processing it.

#### A.4.11.4.2 void ixHssAccCodeletChannelisedServiceStart (..)

void ixHssAccCodeletChannelisedServiceStart (IxHssAccHssPort hssPortId)

**Parameters**

***IxHssAccHssPort*** hssPortId (in) — the HSS port ID (0 or 1).

First create both the TX and RX buffers.

For controlling the RX and TX of data, a frequent sampling rate will be used. This allows flexibility in deciding when to RX and TX data. The common configuration for the Channelised Service will be as follows:

- Trigger rate = 1ms
- Bytes per timeslot trigger = 8 (8 bytes @ 1ms => 64Kbps)
- Number of channels = 16
- Bytes per sample = 44 bytes (5.5ms sampling rate)
- These parameters are example values that would be typical of a voice over ATM application. A voice sample of 44 bytes plus an AAL2 header will fit into the 48 byte payload of an ATM cell. A trigger rate of 1ms ensures that we are very responsive to handling voice samples. A trigger rate of 5ms or 6ms would seem more appropriate, but missing a trigger due to busy CPU means a much longer delay before the data gets processed on the next trigger.
- A latency factor is introduced for the RX path to give the client sufficient time to process the data received from the HSS port. For example, the client may wish to retransmit the data over an ATM network. The higher this latency, the longer it will be before data received from the HSS port is overwritten. The RX configuration will be as follows:
- RX latency factor = 4 (=> 3 * 5.5ms before data is over-written)
- RX buffer size per channel = 176 bytes (bytes per sample times latency factor, i.e. 44 * 4)
- RX buffer size for all channels = 2816 bytes (buffer size per channel times number of channels, i.e. 176 * 16)
- A latency factor is introduced for the TX path to absorb the jitter introduced by the network the data is coming from. The HSS port transmits data at a constant rate, however packets don't arrive at a constant rate from a data network. The TX configuration will be as follows:
- TX latency factor = 8 (=> 7 * 5.5ms before TX data runs out)
- TX buffer size per channel = 352 bytes (bytes per sample times latency factor, i.e. 44 * 8)
- TX buffer size for all channels = 5632 bytes (buffer size per channel times number of channels, i.e. 352 * 16)
- For the TX service an array of pointers to samples is also required. This array will contain a pointer to each 44-byte sample in the TX buffer, i.e. 5632 / 44 = 128 pointers (or in other words the number of channels times the latency factor, i.e. 16 * 8).

- Next connect the client to the TX/RX NPE Channelised Service by invoking ixHssAccChanConnect(). The connection parameters will be as follows:
- Bytes per timeslot trigger = 8
- RX circular buffer = RX buffer
- Number of RX bytes per timeslot = 176 (RX buffer size per channel)
- TX pointer list = TX pointer array
- Number of TX pointer lists = 8 (latency factor)
- Number of TX bytes per block = 44 (bytes per sample)
- The client will supply ixHssAccCodeletChanRxCallback() as the IxHssAccChanRxCallback parameter to handle the actual TX/RX of channelised data.
- Finally start the Channelised Service and enable the TX/RX flows by invoking ixHssAccChanPortEnable().
- Note, before starting the service the Codelet will need to prepare the data to be transmitted.

#### A.4.11.4.3 void ixHssAccCodeletChannelisedServiceStop (..)

void ixHssAccCodeletChannelisedServiceStop (IxHssAccHssPort hssPortId)

**Parameters**

***IxHssAccHssPort*** hssPortId (in) — the HSS port ID (0 or 1).

Stop the Channelised Service and disable the TX/RX flows by calling ixHssAccChanPortDisable().

Disconnect the client from the TX/RX NPE Channelised Service by invoking ixHssAccChanDisconnect().

#### A.4.11.4.4 void ixHssAccCodeletChannelisedVerifySet (..)

void ixHssAccCodeletChannelisedVerifySet (BOOL verifyOn)

**Parameters**

***BOOL*** verifyOn (in) — whether to verify data or not.

This function sets the verification of data on or off. Verification of data can be turned off to improve performance.

#### A.4.11.4.5 void ixHssAccCodeletCodeletLoopbackSet (..)

void ixHssAccCodeletCodeletLoopbackSet (BOOL codeletLoopback)

**Parameters**

***BOOL*** codeletLoopback (in) — the Codelet loopback state.

This function sets the Codelet loopback state.

#### A.4.11.4.6 void ixHssAccCodeletConfigure (..)

void ixHssAccCodeletConfigure (IxHssAccHssPort hssPortId)

### Parameters

***IxHssAccHssPort*** hssPortId (in) — the HSS port ID (0 or 1).

For each top-level scenario the HSS port needs to be configured appropriately.

The Codelet will simulate five simultaneous clients, four using the Packetised Service and one using the Channelised Service. Each packetised client will use a subset of the timeslots available in the T1/E1 stream for that client. The channelised client will use timeslots from the remaining available timeslots from each of the four T1/E1 streams. Two packetised clients will use RAW mode while two will use HDLC mode. The channelised client will use some timeslots to carry voice at 64Kbps and other timeslots to carry voice at 56Kbps (employing “bit-stealing”).

The HSS Port needs to be configured to allow the HSS to interface to the particular framer and protocol being used. The configuration options and their values are shown below. Some options are relevant to TX and others to both TX and RX. These values have been initially chosen based on the HSS co-processor default settings. These values will get written to the HSSTXPCR, HSSRXPCR, HSSTXFCR, and HSSRXFCR registers of the HSS co-processor.

- Frame sync type = active low (HSSTXPCR/HSSRXPCR)
- Frame sync output en. = output rising (HSSTXPCR/HSSRXPCR)
- Frame sync clock edge = rising (HSSTXPCR/HSSRXPCR)
- Data clock edge = falling (HSSTXPCR/HSSRXPCR)
- Clock direction = output (HSSTXPCR/HSSRXPCR)
- Frame usage = enabled (HSSTXPCR/HSSRXPCR)
- Data rate = clock rate (HSSTXPCR/HSSRXPCR)
- Data polarity = same (HSSTXPCR/HSSRXPCR)
- Data endianness = lsb endian (HSSTXPCR/HSSRXPCR)
- Drain mode = normal (HSSTXPCR)
- FBit usage = data (HSSTXPCR/HSSRXPCR)
- Data enable = data (HSSTXPCR)
- 56K type = low (HSSTXPCR)
- Unassigned type = low (HSSTXPCR)
- FBit type = fifo (HSSTXPCR)
- 56K endianness = bit 7 unused (HSSTXPCR)
- 56K selection = 32/8 data (HSSTXPCR)
- Frame offset = 0 (HSSTXFCR/HSSRXFCR)
- Frame size = 1024 (4 * 32 * 8) (HSSTXFCR/HSSRXFCR)
- The following configuration parameters are related to the scenarios we are running:
- HssChannelised Number of Channels = 16
- HssPacketised Number of Packet-Pipes = 4
- HssChannelised Idle Pattern = 0x7F
- Loopback = TRUE/FALSE (depends on scenario)

- HssPacketised Pipes 0-3 Idle Patterns = 0x7F7F7F7F
- Clock speed = 8192
- The timeslots of each of the four T1/E1 streams will be assigned as follows:
- 4 timeslots for 64K channelised (8, 15, 24, 31)
- 4 timeslots for 56K channelised (4, 12, 20, 28)
- 16 timeslots for HDLC (1-2, 5-6, 9-10, 13-14, 17-18, 21-22, 25-26, 29-30)
- 6 timeslots unassigned (3, 7, 11, 19, 23, 27)
- The Codelet will provide ixHssAccCodeletLastHssErrorCallback() as the IxHssAccLastErrorCallback during configuration. ixHssAccLastErrorRetrievalInitiate() will be called whenever an increase in the number of HSS errors is detected. This will initiate a callback to the Codelet's ixHssAccCodeletLastHssErrorCallback().

#### A.4.11.4.7 void ixHssAccCodeletHssLoopbackSet (..)

void ixHssAccCodeletHssLoopbackSet (BOOL hssLoopback)

**Parameters**

***BOOL*** hssLoopback (in) — the HSS loopback state.

This function sets the HSS loopback state.

#### A.4.11.4.8 void ixHssAccCodeletInit (..)

void ixHssAccCodeletInit (BOOL runForever)

**Parameters**

***BOOL*** runForever (in) — Flag indicating whether or not to run the services indefinitely.

This function is the launch-point for the codelet. It assumes the environment has been setup. It initializes the IxHssAcc component and then creates a thread to run the IxHssAcc services.

#### A.4.11.4.9 void ixHssAccCodeletPacketisedServiceRun (..)

void ixHssAccCodeletPacketisedServiceRun (IxHssAccHssPort hssPortId, IxHssAccHdlcPort hdlcPortId)

**Parameters**

***IxHssAccHssPort*** hssPortId (in) — the HSS port ID (0 or 1).

***IxHssAccHdlcPort*** hdlcPortId (in) — the port id (0,1,2,3) to run the service on.

Packets will be transmitted by calling ixHssAccPktPortTx(). Buffers will be taken from the TX pool and chained together to create a buffer for sending.

Both the clients using HDLC mode and the clients using RAW mode will send and receive packets of 16Kb in size.

In the transmit-done callback, ixHssAccCodeletPktTxDoneCallback(), the transmitted packets will be returned to the TX buffer pool to be reused. This callback will also maintain statistics of packets transmitted and number of HSS errors.

Packets will be received via the callback routine ixHssAccCodeletPktRxCallback(). Received packets will be returned to the RX buffer pool to be reused. This callback will also maintain statistics of packets received and number of HSS errors.

When operating in loopback mode any received packets will be copied and re-transmitted. The reason for the copying is to maintain the RX buffer pool and TX buffer pool separate. When operating as data source/sink packets will be created and transmitted, and the contents of received packets will be verified against those transmitted (the hardware is performing a loopback).

### A.4.11.4.10 void ixHssAccCodeletPacketisedServiceStart (..)

void ixHssAccCodeletPacketisedServiceStart (IxHssAccHssPort hssPortId, IxHssAccHdlcPort hdlcPortId)

#### Parameters

***IxHssAccHssPort*** hssPortId (in) — the HSS port ID (0 or 1).

***IxHssAccHdlcPort*** hdlcPortId (in) — the port id (0,1,2,3) to start the service on.

First create both TX and RX mbuf pools. Each client will create a pool of 32 mbufs of size 2K for TX and 32 mbufs of size 2K for RX. For sending packets larger than 2K, mbufs will be chained together. Similarly the NPE will chain together mbufs buffers as necessary to receive packets larger than 2K. Both the TX and RX mbufs will be recycled so that the pools just circulate the same mbufs over and over.

Next connect the client to one of the four available HDLC ports by invoking ixHssAccPktPortConnect(). The connection parameters will be specified as follows. Where indicated, the values will get written to the TxCtxt and RxCtxt registers of the HDLC co-processor.

- HDLC framing = TRUE/FALSE (clients 0 and 2 will use HDLC mode, clients 1 and 3 will use RAW mode).
- Raw mode block size = 4096 words (i.e. 16K)
- Raw mode idle pattern = 0x5F5F5F5F
- HDLC idle transmission type = flags (TxCtxt/RxCtxt)
- HDLC data endianness = lsb endian (TxCtxt/RxCtxt
- CRC type (CRC-16 or CRC-32) = CRC-32 (TxCtxt/RxCtxt)
- Number of flags inserted at start of frame = 1 (TxCtxt)
- The method ixHssAccCodeletPktRxCallback() will be supplied as the IxHssAccPktRxCallback to receive packets. A pointer to a structure identifying the client will be passed as the rxUserId parameter.
- The method ixHssAccCodeletPktRxFreeLowCallback() will be supplied as the IxHssAccPktRxFreeLowCallback to provide RX buffers on request. A pointer to a structure identifying the client will be passed as the rxFreeLowUserId.
- The method ixHssAccCodeletPktTxDoneCallback() will be supplied as the IxHssAccPktTxDoneCallback to be passed back a packet after it has been transmitted. A pointer to a structure identifying the client will be passed as the txDoneUserId.

- After connecting to the service, the RX buffers will be passed to the IxHssAcc component by calling ixHssAccPktPortRxFreeReplenish() to make them available for receiving packets into.
- Finally the Packetised Service will be started and the RX flow will be enabled by invoking ixHssAccPktPortEnable().

#### A.4.11.4.11 void ixHssAccCodeletPacketisedServiceStop (..)

void ixHssAccCodeletPacketisedServiceStop (IxHssAccHssPort hssPortId, IxHssAccHdlcPort hdlcPortId)

**Parameters**

***IxHssAccHssPort*** hssPortId (in) — the HSS port ID (0 or 1).

***IxHssAccHdlcPort*** hdlcPortId (in) — the port id (0,1,2,3) to stop the service on.

Stop the Packetised Service and disable the RX flow by calling ixHssAccPktPortDisable().

Disconnect the client from the HDLC port by invoking ixHssAccPktPortDisconnect().

#### A.4.11.4.12 void ixHssAccCodeletPacketisedVerifySet (..)

void ixHssAccCodeletPacketisedVerifySet (BOOL verifyOn)

**Parameters**

***BOOL*** verifyOn (in) — whether to verify data or not.

This function sets the verification of data on or off. Verification of data can be turned off to improve performance.

#### A.4.11.4.13 void ixHssAccCodeletShow (..)

void ixHssAccCodeletShow (IxHssAccHssPort hssPortId)

**Parameters**

***IxHssAccHssPort*** hssPortId (in) — the HSS port ID (0 or 1).

Display statistics for the specified HSS port.

#### A.4.11.4.14 void ixHssAccCodeletShowReset (..)

void ixHssAccCodeletShowReset (IxHssAccHssPort hssPortId)

**Parameters**

***IxHssAccHssPort*** hssPortId (in) — the HSS port ID (0 or 1).

Reset the statistics for the specified HSS port.

## A.4.12 HSS Access Codelet Channelised (IxHssAccCodeletChan)

The interface for channelised implementation of the HSS Access Codelet.

### A.4.12.1 Defines

- #define IX_HSSACC_CODELET_CHAN_BYTES_PER_TS_TRIG (8)
  trigger rate of 1ms.
- #define IX_HSSACC_CODELET_CHAN_NUM_CHANS (16)
  number of channels.
- #define IX_HSSACC_CODELET_CHAN_BYTES_PER_SAMPLE (44)
  bytes per sample of data.
- #define IX_HSSACC_CODELET_CHAN_RX_LATENCY_FACTOR (4)
  RX latency factor.
- #define IX_HSSACC_CODELET_CHAN_RX_BUFSIZE_PERCHAN
  RX buffer size per channel.
- #define IX_HSSACC_CODELET_CHAN_TX_LATENCY_FACTOR (8)
  TX latency factor.
- #define IX_HSSACC_CODELET_CHAN_IDLE_PATTERN (0x7F)
  Channelised idle pattern.

### A.4.12.2 Detailed Description

The interface for channelised implementation of the HSS Access Codelet.

### A.4.12.3 Define Documentation

#### A.4.12.3.1 #define IX_HSSACC_CODELET_CHAN_RX_BUFSIZE_PERCHAN

Value:

```
(IX_HSSACC_CODELET_CHAN_RX_LATENCY_FACTOR * \
     IX_HSSACC_CODELET_CHAN_BYTES_PER_SAMPLE)
```

RX buffer size per channel.

## A.4.13 HSS Access Codelet Common (IxHssAccCodeletCom) API

The interface for common implementation of the HSS Access Codelet.

### A.4.13.1 Data Structures

- struct ChannelisedStats
  Type definition structure for channelised statistics.
- struct ChannelisedStats
  Type definition structure for channelised statistics.
- struct GeneralStats
  Type definition structure for general statistics.
- struct GeneralStats

Type definition structure for general statistics.

- struct IxHssAccCodeletStats
  Type definition structure for HSS Access Codelet statistics.
- struct IxHssAccCodeletStats
  Type definition structure for HSS Access Codelet statistics.
- struct PacketisedStats
  Type definition structure for Packetised statistics.
- struct PacketisedStats
  Type definition structure for Packetised statistics.

### A.4.13.2 Functions

- void ixHssAccCodeletLastHssErrorCallback (unsigned lastHssError, unsigned servicePort)
- void ixHssAccCodeletNumHssErrorsUpdate (IxHssAccHssPort hssPortId, unsigned numHssErrs)
- void ixHssAccCodeletPktErrorsUpdate (IxHssAccHssPort hssPortId, IxHssAccHdlcPort hdlcPortId, IxHssAccPktStatus pktStatus)

### A.4.13.3 Variables

- IxHssAccCodeletStats stats [IX_HSSACC_HSS_PORT_MAX]

### A.4.13.4 Detailed Description

The interface for common implementation of the HSS Access Codelet.

### A.4.13.5 Function Documentation

#### A.4.13.5.1 void ixHssAccCodeletLastHssErrorCallback (..)

void ixHssAccCodeletLastHssErrorCallback (unsigned lastHssError, unsigned servicePort)

**Parameters**

***unsigned*** lastHssError (in) — the last HSS error registered by the NPE.

***unsigned*** servicePort (in) — this is the NPE service port number - 0-3 for Packetised Services, 0 for Channelised Service.

This function is of type IxHssAccLastErrorCallback, the prototype of the clients function to accept notification of the last error.

This function is registered through ixHssAccPortInit(). It will be called in response to a call to ixHssAccLastErrorRetrievalInitiate().

This callback will record the last error, so that errors can be accumulated and later reported.

#### A.4.13.5.2 void ixHssAccCodeletNumHssErrorsUpdate (..)

void ixHssAccCodeletNumHssErrorsUpdate (IxHssAccHssPort hssPortId, unsigned numHssErrs)

**Parameters**

***IxHssAccHssPort*** hssPortId (in) — the HSS port ID (0 or 1).

***unsigned*** numHssErrs (in) — The total number of HSS port errors since initial port configuration.

This function will check the number of HSS errors. If the HSS error count has increased since the last invocation of this function then ixHssAccLastErrorRetrievalInitiate() is invoked to query the last error.

#### A.4.13.5.3 void ixHssAccCodeletPktErrorsUpdate (..)

void ixHssAccCodeletPktErrorsUpdate (IxHssAccHssPort hssPortId, IxHssAccHdlcPort hdlcPortId, IxHssAccPktStatus pktStatus)

**Parameters**

***IxHssAccHssPort*** hssPortId (in) — the HSS port ID (0 or 1).

***IxHssAccHdlcPort*** hdlcPortId (in) — the HDLC port ID (0,1,2,3).

***IxHssAccPktStatus*** pktStatus (in) — Indicates the status of packets passed to the client.

This function will update the packet status error stats.

### A.4.13.6 Variable Documentation

IxHssAccCodeletStats stats[IX_HSSACC_HSS_PORT_MAX]

Codelet statistics

## A.4.14 HSS Access Codelet Config (IxHssAccCodeletConfig) API

The interface for config implementation of the HSS Access Codelet.

### A.4.14.1 Functions

- IxHssAccTdmSlotUsage ixHssAccCodeletChannelisedTimeslotGet (unsigned channelIndex)

### A.4.14.2 Detailed Description

The interface for config implementation of the HSS Access Codelet.

### A.4.14.3 Function Documentation

#### A.4.14.3.1 IxHssAccTdmSlotUsage ixHssAccCodeletChannelisedTimeslotGet (..)

IxHssAccTdmSlotUsage ixHssAccCodeletChannelisedTimeslotGet (unsigned channelIndex)

#### Parameters

***unsigned*** channelIndex (in) — the voice channel index.

This function returns the timeslot assignment for the specified voice channel.

#### Returns

the channel's timeslot assignment.

## A.4.15 HSS Access Codelet MBuf Pool (IsHssAccCodeletMbuf) API

The interface for mbuf pool implementation of the HSS Access Codelet.

### A.4.15.1 Defines

- #define MBUF_SIZE 2048
  preferred netBufLib cluster size is 2K.

### A.4.15.2 Typedefs

- typedef IX_MBUF_POOL IxHssAccCodeletMbufPool
  Pool Id type used to reference a particular mbuf pool.

### A.4.15.3 Functions

IX_MBUF * ixHssAccCodeletMbufGet (IxHssAccCodeletMbufPool *poolId)

- void ixHssAccCodeletMbufChainFree (IX_MBUF *pMbuf)
- void ixHssAccCodeletMbufPoolInit (IxHssAccCodeletMbufPool **poolIdPtr, unsigned numPoolMbufs, unsigned poolMbufSize)

### A.4.15.4 Detailed Description

The interface for mbuf pool implementation of the HSS Access Codelet.

### A.4.15.5 Function Documentation

#### A.4.15.5.1 void ixHssAccCodeletMbufChainFree (..)

void ixHssAccCodeletMbufChainFree (IX_MBUF * pMbuf)

#### Parameters

***IX_MBUF*** *pMbuf (in) — A pointer to the first mbuf in the mbuf chain to be freed

This frees a chain of mbufs back to its memory pool

#### Returns

None

### A.4.15.5.2 void ixHssAccCodeletMbufPoolInit (..)

void ixHssAccCodeletMbufPoolInit (IxHssAccCodeletMbufPool ** poolIdPtr, unsigned numPoolMbufs, unsigned poolMbufSize)

#### Parameters

***IxHssAccCodeletMbufPool*** **poolIdPtr (out) — A pointer to the mbuf pool pointer created

***unsigned*** numPoolMbufs (in) — The number of mbufs the pool should contain

***unsigned*** poolMbufSize (in) — The size, in bytes, of the mbufs in the pool

This creates a memory pool of mbufs. Use ixHssAccCodeletMbufGet() to get an mbuf from the pool, and ixHssAccCodeletMbufFree() or ixHssAccCodeletMbufChainFree() to return mbufs to the pool

#### Returns

None

## A.4.16 HSS Access Codelet Packetised (IxHssAccCodeletPkt) API

The interface for packetised implementation of the HSS Access Codelet.

### A.4.16.1 Defines

- #define IX_HSSACC_CODELET_PKT_NUM_TX_BUFS (32)
  num of TX mbufs per client.
- #define IX_HSSACC_CODELET_PKT_NUM_RX_BUFS (32)
  num of RX mbufs per client.
- #define IX_HSSACC_CODELET_PKT_BUFSIZE (MBUF_SIZE)
  mbuf size.
- #define IX_HSSACC_CODELET_PKT_NUM_BUFS
  TX and RX mbufs per client.
- #define IX_HSSACC_CODELET_PKT_TX_PKTSIZE
  (IX_HSSACC_CODELET_PKT_BUFSIZE * 8)
  TX packet size (i.e.
- #define IX_HSSACC_CODELET_PKT_IDLE_PATTERN (0x7F7F7F7F)
  Packetised idle pattern (tx'ed when service is disabled).
- #define IX_HSSACC_CODELET_PKT_RAW_IDLE_PATTERN (0x5F5F5F5F)
  Raw mode idle pattern (tx'ed when no data - HDLC tx's ones or flags).
- #define IX_HSSACC_CODELET_PKT_RAW_STALE_BYTES_MAX (128)
  Number of stale bytes at start of first raw mode packet which can be safely ignored if they occur.

### A.4.16.2 Detailed Description

The interface for packetised implementation of the HSS Access Codelet.

### A.4.16.3 Define Documentation

#### A.4.16.3.1 #define IX_HSSACC_CODELET_PKT_NUM_BUFS

Value:

```
(IX_HSSACC_CODELET_PKT_NUM_TX_BUFS + \
     IX_HSSACC_CODELET_PKT_NUM_RX_BUFS)
```

TX and RX mbufs per client.

#### A.4.16.3.2 #define IX_HSSACC_CODELET_PKT_TX_PKTSIZE (IX_HSSACC_CODELET_PKT_BUFSIZE * 8)

TX packet size (i.e.
8 mbufs)

## A.4.17 Oam Codelet (IxOamCodelet) API

Intel® IXP42X product line codelet Oam demo component API.

### A.4.17.1 Functions

- IX_STATUS ixOamCodeletInit (unsigned int numPorts)

  This function is used to configure OAM across a number of previously configured ports. N.B. IxQmgr, IxNpeMh, IxAtmSch, IxAtmdAcc, IxAtmm must be initialized before calling this API, e.g. see AtmCodelet:ixAtmCodeletSystemInit().

- IX_STATUS ixOamCodeletOamF5EtePing (IxAtmLogicalPort port, unsigned int vpi, unsigned int vci, unsigned int numCells)

  This function is used to initiate an OAM F5 ETE loopback, i.e. OAM ping, on the specified Port, VPI and VCI for a number of cells. ixOamCodeletInit must be called before calling this function.

- IX_STATUS ixOamCodeletOamF5SegPing (IxAtmLogicalPort port, unsigned int vpi, unsigned int vci, unsigned int numCells)

  This function is used to initiate an OAM F5 Segment loopback, i.e. OAM ping, on the specified Port, VPI and VCI for a number of cells. ixOamCodeletInit must be called before calling this function.

- IX_STATUS ixOamCodeletOamF4EtePing (IxAtmLogicalPort port, unsigned int vpi, unsigned int numCells)

  This function is used to initiate an OAM F4 ETE loopback, i.e. OAM ping, on the specified Port, VPI for a number of cells. F4 flows are VPC based and as such don't require a VCI. ixOamCodeletInit must be called before calling this function.

- IX_STATUS ixOamCodeletOamF4SegPing (IxAtmLogicalPort port, unsigned int vpi, unsigned int numCells)

This function is used to initiate an OAM F4 Segment loopback, i.e. OAM ping, on the specified Port, VPI for a number of cells. F4 flows are VPC based and as such don't require a VCI. ixOamCodeletInit must be called before calling this function.

- void ixOamCodeletShow (void)

  This function is used to dump OAM statistics, i.e. parent and child cell receive and transmit counts etc.

### A.4.17.2 Detailed Description

Intel® IXP42X product line codelet Oam demo component API.

This codelet demonstrates an example implementation of OAM F4 Segment, F4 End-To-End (ETE), F5 Segment and F5 ETE loopback that makes use of the AtmdAcc component, as well as demonstrating how the lower layer IxAtmdAcc component can be used for configuration and control.

#### Features

An interface is provided to configure the access layer to enable OAM traffic.

OAM Loopback responses are sent automatically on receipt of OAM F4 and F5 parent Segment and ETE loopback cells, i.e. externally initiated OAM Loopbacks.

Interfaces are provided to initiate an OAM F4/F5 Segment or ETE Loopback, see ITU-610. One loopback can be initiated at a time, this is to keep the codelet simple, this is independent of sending responses to parent loopback cells received.

An interface is provided to query OAM traffic statistics.

#### ATM Components used by this codelet

IxAtmdAcc. This component is the low level interface by which OAM cells get transmitted to, and received from the UTOPIA bus.

IxAtmm. This component provides ATM port and VC management facilities. This component also contains the configuration of the UTOPIA co-processor.

DemoUtils. This codelet provides VC setup facilities using IxAtmdAcc.

### A.4.17.3 Function Documentation

#### A.4.17.3.1 ixOamCodeletInit (..)

ixOamCodeletInit (unsigned int numPorts)

This function is used to configure OAM across a number of previously configured ports. N.B. IxQmgr, IxNpeMh, IxAtmSch, IxAtmdAcc, IxAtmm must be initialized before calling this API, e.g. see AtmCodelet:ixAtmCodeletSystemInit().

#### Parameters

***unsigned int numPorts*** — The number of ATM ports to configure on. This must be less than or equal to the number of ports configured in e.g. the call to ixAtmCodeletInit().

### Returns

- IX_SUCCESS — Indicates that the Atm Codelet has successfully provisioned the OAM Tx Port channels and the OAM Rx channel.
- IX_FAIL — Some internal error has prevented the ATM codelet from provisioning the channels.

#### A.4.17.3.2 ixOamCodeletOamF4EtePing (..)

ixOamCodeletOamF4EtePing (IxAtmLogicalPort port, unsigned int vpi, unsigned int numCells)

This function is used to initiate an OAM F4 ETE loopback, i.e. OAM ping, on the specified Port, VPI for a number of cells. F4 flows are VPC based and as such don't require a VCI. ixOamCodeletInit must be called before calling this function.

OAM F4 ETE Parent Loopback cells are sent to the F4 connection end point (LLID = all one's).

If an OAM F4 ETE Child Loopback cell has been received in response to a Parent cell sent, a message will be displayed at the WindSh prompt.

The loopback is complete and a subsequent loopback is allowed only when all the loopback parent cells have been sent and either A) a child loopback cell has been received in response to the last cell OR B) 5 seconds have passed since the last cell was sent. N.B. the function blocks until the loopback has completed.

### Parameters

***IxAtmLogicalPort*** — The port to send the OAM loopback cells on. This port must have been configured previously.

***unsigned int VPI*** — The VPI of the channel to send the OAM loopback cells on. An Aal0/Aal5 channel need not be configured previously on the same VPI/VCI.

***unsigned int numCells*** — The number of parent loopback cells to send. Not checked against some max value, only limited by size of type.

### Returns

- IX_SUCCESS — Indicates that the Oam Codelet has successfully performed the OAM ping.
- IX_FAIL — Some internal error has prevented the Oam Codelet performing the OAM ping.

#### A.4.17.3.3 ixOamCodeletOamF4SegPing (..)

ixOamCodeletOamF4SegPing (IxAtmLogicalPort port, unsigned int vpi, unsigned int numCells)

This function is used to initiate an OAM F4 Segment loopback, i.e. OAM ping, on the specified Port, VPI for a number of cells. F4 flows are VPC based and as such don't require a VCI. ixOamCodeletInit must be called before calling this function.

OAM F4 Segment Parent Loopback cells are sent to the F4 connection end point (LLID = all one's).

If an OAM F4 Segment Child Loopback cell has been received in response to a Parent cell sent, a message will be displayed at the WindSh prompt.

The loopback is complete and a subsequent loopback is allowed only when all the loopback parent cells have been sent and either A) a child loopback cell has been received in response to the last cell OR B) 5 seconds have passed since the last cell was sent. N.B. the function blocks until the loopback has completed.

#### Parameters

***IxAtmLogicalPort*** — The port to send the OAM loopback cells on. This port must have been configured previously.

***unsigned int VPI*** — The VPI of the channel to send the OAM loopback cells on. An Aal0/Aal5 channel need not be configured previously on the same VPI/VCI.

***unsigned int numCells*** — The number of parent loopback cells to send. Not checked against some max value, only limited by size of type.

#### Returns

- IX_SUCCESS — Indicates that the Oam Codelet has successfully performed the OAM ping.
- IX_FAIL — Some internal error has prevented the Oam Codelet performing the OAM ping.

### A.4.17.3.4 ixOamCodeletOamF5EtePing (..)

ixOamCodeletOamF5EtePing (IxAtmLogicalPort port, unsigned int vpi, unsigned int vci, unsigned int numCells)

This function is used to initiate an OAM F5 ETE loopback, i.e. OAM ping, on the specified Port, VPI and VCI for a number of cells. ixOamCodeletInit must be called before calling this function.

OAM F5 ETE Parent Loopback cells are sent to the F5 connection end point (LLID = all one's).

If an OAM F5 ETE Child Loopback cell has been received in response to a Parent cell sent, a message will be displayed at the WindSh prompt.

The loopback is complete and a subsequent loopback is allowed only when all the loopback parent cells have been sent and either A) a child loopback cell has been received in response to the last cell OR B) 5 seconds have passed since the last cell was sent. N.B. the function blocks until the loopback has completed.

#### Parameters

***IxAtmLogicalPort*** — The port to send the OAM loopback cells on. This port must have been configured previously.

***unsigned int VPI*** — The VPI of the channel to send the OAM loopback cells on. An Aal0/Aal5 channel need not be configured previously on the same VPI/VCI.

***unsigned int VCI*** — The VCI of the channel to send the OAM loopback cells on. An Aal0/Aal5 channel need not be configured previously on the same VPI/VCI.

***unsigned int numCells*** — The number of parent loopback cells to send. Not checked against some max value, only limited by size of type.

#### Returns

- IX_SUCCESS — Indicates that the Oam Codelet has successfully performed the OAM ping.

- IX_FAIL — Some internal error has prevented the Oam Codelet performing the OAM ping.

#### A.4.17.3.5 ixOamCodeletOamF5SegPing (..)

ixOamCodeletOamF5SegPing (IxAtmLogicalPort port, unsigned int vpi, unsigned int vci, unsigned int numCells)

This function is used to initiate an OAM F5 Segment loopback, i.e. OAM ping, on the specified Port, VPI and VCI for a number of cells. ixOamCodeletInit must be called before calling this function.

OAM F5 Segment Parent Loopback cells are sent to the F5 connection end point (LLID = all one's).

If an OAM F5 Segment Child Loopback cell has been received in response to a Parent cell sent, a message will be displayed at the WindSh prompt.

The loopback is complete and a subsequent loopback is allowed only when all the loopback parent cells have been sent and either A) a child loopback cell has been received in response to the last cell OR B) 5 seconds have passed since the last cell was sent. N.B. the function blocks until the loopback has completed.

##### Parameters

***IxAtmLogicalPort*** — The port to send the OAM loopback cells on. This port must have been configured previously.

***unsigned int VPI*** — The VPI of the channel to send the OAM loopback cells on. An Aal0/Aal5 channel need not be configured previously on the same VPI/VCI.

***unsigned int VCI*** — The VCI of the channel to send the OAM loopback cells on. An Aal0/Aal5 channel need not be configured previously on the same VPI/VCI.

***unsigned int numCells*** — The number of parent loopback cells to send. Not checked against some max value, only limited by size of type.

##### Returns

- IX_SUCCESS — Indicates that the Oam Codelet has successfully performed the OAM ping.
- IX_FAIL — Some internal error has prevented the Oam Codelet performing the OAM ping.

#### A.4.17.3.6 ixOamCodeletShow (void)

This function is used to dump OAM statistics, i.e. parent and child cell receive and transmit counts etc.

##### Parameters

none

##### Returns

void

## A.4.18 UART Access Codelet (IxUARTAccCodelet) API

The interface prototypes for Intel® IXP42X product line UART codelet.

### A.4.18.1 Defines

- #define TRANSLATE_OUT_CHARS
  Map '
  ' to '\r
  ' on output.
- #define TRANSLATE_IN_CHARS
  Map '\r' to '
  ' on input.

### A.4.18.2 Typedefs

- typedef enum _ixUARTAccPort ixUARTAccPort

### A.4.18.3 Enumerations

- enum _ixUARTAccPort { IX_UART_PORT1 = 1, IX_UART_PORT2 = 2 }

### A.4.18.4 Functions

- PUBLIC IX_STATUS ixUARTAccCodeletDevConfig (ixUARTAccPort uartNo)
  Configure a UART device.
- PUBLIC IX_STATUS ixUARTAccCodeletSwLoopback (ixUARTAccPort uartNo)
  Echo characters as they are received. Stops when a ^Z is received.
- PUBLIC int ixUARTAccCodeletWrite (ixUARTAccPort uartNo, char *buf)
  Write to the serial device. This is a blocking call.
- PUBLIC int ixUARTAccCodeletRead (ixUARTAccPort uartNo, char *buf, unsigned int count)
  Read from the serial device. This is a blocking call. The buffer returned is NULL terminated.
- PUBLIC IX_STATUS ixUARTAccCodeletShow (ixUARTAccPort uartNo)
  Display statistics for the specified UART.
- PUBLIC IX_STATUS ixUARTAccCodeletBaudDemo (ixUARTAccPort uartNo)
  Output a test string on the device at supported baud rates.

### A.4.18.5 Detailed Description

The interface prototypes for Intel® IXP42X product line UART codelet.

The following scenarios are supported:

- Device initialization, setting up of baud rate, data size etc.
- Character reception.
- Character transmission.
- S/W loopback function.

You must have a PC with serial port supporting the baud rates here and terminal software to view the data as it is passed to/from the UART.

## A.4.18.6 Function Documentation

### A.4.18.6.1 ixUARTAccCodeletBaudDemo (..)

ixUARTAccCodeletBaudDemo (ixUARTAccPort uartNo)

Output a test string on the device at supported baud rates.

#### Parameters

***uartNo*** — Port number of UART to test.

#### Returns

- IX_SUCCESS — Successfully running baud demo
- IX_FAIL — error running baud demo.

### A.4.18.6.2 IX_STATUS ixUARTAccCodeletDevConfig (..)

IX_STATUS ixUARTAccCodeletDevConfig (ixUARTAccPort uartNo)

Configure a UART device.

#### Parameters

***uartNo*** — Port number of UART to configure

#### Returns

- IX_SUCCESS — UART device successfully initialized
- IX_FAIL — error initializing UART device

### A.4.18.6.3 ixUARTAccCodeletRead (..)

ixUARTAccCodeletRead (ixUARTAccPort uartNo, char * buf, unsigned int count)

Read from the serial device. This is a blocking call. The buffer returned is NULL terminated.

#### Parameters

***uartNo*** — Port number of UART to read from.

***buf*** — buffer to place characters in.

***count*** — number of characters to read.

#### Returns

- int — number of characters actually read or IX_FAIL is an unknown UART is selected.

### A.4.18.6.4 ixUARTAccCodeletShow (..)

ixUARTAccCodeletShow (ixUARTAccPort uartNo)

Display statistics for the specified UART.

#### Parameters

***uartNo*** — Port number of UART to display information for.

#### Returns

- IX_SUCCESS — Successfully display UART Statistics
- IX_FAIL — error displaying UART statistics

### A.4.18.6.5 ixUARTAccCodeletSwLoopback (..)

ixUARTAccCodeletSwLoopback (ixUARTAccPort uartNo)

Echo characters as they are received. Stops when a ^Z is received.

#### Parameters

***pUART*** — Port number of the UART we want to test.

#### Returns

- IX_SUCCESS — SW Loopback successfully started on selected device
- IX_FAIL — error starting SW Loopback

### A.4.18.6.6 ixUARTAccCodeletWrite (..)

ixUARTAccCodeletWrite (ixUARTAccPort uartNo, char * buf)

Write to the serial device. This is a blocking call.

#### Parameters

***uartNo*** — Port number of UART to write to.

***buf*** — buffer containing characters to transmit.

Write to the serial device. This is a blocking call.

#### Returns

- int — number of characters actually written or
- IX_FAIL if an unknown UART is selected.

## A.4.19 USB RNDIS Codelet (IxUSBRNDIS) API

Intel® IXP42X product line codelet USB RNDIS API.

### A.4.19.1 Modules

- Intel® IXP42X product line USB RNDIS Vendor Codelet (IxUSBRNDIS)
  Intel® IXP42X product line codelet for RNDIS Vendor information.
- Intel® IXP42X product line USB RNDIS End Driver Codelet (IxUSBRNDIS) API
  Intel® IXP42X product line codelet for RNDIS End API.

### A.4.19.2 Data Structures

- struct RndisLayerData
- struct RndisLayerStats

### A.4.19.3 Defines

- #define USB_COMMUCATION_DEVICE (0x02)
- #define USB_COMMUNICATION_INTERFACE (0x02)
- #define USB_ABSTRACT_CONTROL_MODEL (0x02)
- #define USB_VENDOR_SPECIFIC_PROTOCOL (0xff)
- #define USB_DATA_INTERFACE (0x0a)
- #define RNDIS_SEND_ENCAPSULATED_COMMAND (0x00)
- #define RNDIS_GET_ENCAPSULATED_RESPONSE (0x01)
- #define RNDIS_RESPONSE_AVAILABLE (0x01)
- #define RNDIS_RESPONSE_AVAILABLE_NOTIFICATION_SIZE (2)
- #define RNDIS_NOTIFICATION_ENDPOINT (ENDPOINT_5)
- #define RNDIS_DATA_IN_ENDPOINT (ENDPOINT_1)
- #define RNDIS_DATA_OUT_ENDPOINT (ENDPOINT_2)
- #define RNDIS_CONTROL_ENDPOINT (ENDPOINT_0)
- #define RNDIS_INITIALIZE (0x2)
- #define RNDIS_HALT (0x3)
- #define RNDIS_QUERY (0x4)
- #define RNDIS_SET (0x5)
- #define RNDIS_RESET (0x6)
- #define RNDIS_INDICATE_STATUS (0x7)
- #define RNDIS_KEEPALIVE (0x8)
- #define RNDIS_COMPLETED (0x80000000)
- #define RNDIS_MESSAGE_TYPE_OFFSET (0)

- #define RNDIS_REQUEST_ID_OFFSET (2)
- #define RNDIS_OID_OFFSET (3)
- #define RNDIS_BUFFER_LENGTH_OFFSET (4)
- #define RNDIS_BUFFER_OFFSET_OFFSET (5)
- #define RNDIS_DATA_OFFSET_OFFSET (2)
- #define RNDIS_DATA_LENGTH_OFFSET (3)
- #define RNDIS_DATA_PAYLOAD_BYTE_OFFSET(UINT32_pPacket) (44)
- #define RNDIS_DATA_PAYLOAD_BYTE_LENGTH(UINT32_pPacket) ((UINT32_pPacket)[RNDIS_DATA_LENGTH_OFFSET])
- #define RNDIS_DATA_PACKET_HEADER_SIZE (44)
- #define RNDIS_PACKET_MSG (0x1)
- #define RNDIS_PAYLOAD_OFFSET (36)
- #define RNDIS_INITIALIZE_COMPLETE_SIZE (52)
- #define RNDIS_QUERY_COMPLETE_HEADER_SIZE (24)
- #define RNDIS_GENERIC_MSG_COMPLETE_SIZE (16)
- #define RNDIS_NUMBER_OF_SUPPORTED_OIDS (26)
- #define RNDIS_MAXIMUM_FRAME_SIZE (1520)
- #define RNDIS_MAXIMUM_TOTAL_SIZE (1558)
- #define RNDIS_USB_SPEED (12000000)
- #define RNDIS_LINK_SPEED (RNDIS_USB_SPEED / 100)
- #define RNDIS_RECEIVE_BLOCK_SIZE (1514)
- #define RNDIS_TRANSMIT_BLOCK_SIZE (64)
- #define RNDIS_MAX_PACKETS_PER_TRANSFER (1)
- #define RNDIS_MAX_HOST_PACKET_SIZE (2048)
- #define RNDIS_PACKET_ALIGNMENT (1)
- #define RNDIS_STATUS_SUCCESS (0)
- #define RNDIS_STATUS_FAILURE (0xc0000001)
- #define RNDIS_STATUS_MEDIA_CONNECT (0x4001000b)
- #define RNDIS_STATUS_MEDIA_DISCONNECT (0x4001000c)
- #define RNDIS_MAJOR_VERSION (1)
- #define RNDIS_MINOR_VERSION (0)
- #define RNDIS_DF_CONNECTIONLESS (1)
- #define RNDIS_DF_CONNECTION_ORIENTED (2)
- #define RNDIS_MEDIUM_802_3 (0)
- #define OID_GEN_SUPPORTED_LIST 0x00010101
- #define OID_GEN_HARDWARE_STATUS 0x00010102

- #define OID_GEN_MEDIA_SUPPORTED 0x00010103
- #define OID_GEN_MEDIA_IN_USE 0x00010104
- #define OID_GEN_MAXIMUM_LOOKAHEAD 0x00010105
- #define OID_GEN_MAXIMUM_FRAME_SIZE 0x00010106
- #define OID_GEN_LINK_SPEED 0x00010107
- #define OID_GEN_TRANSMIT_BUFFER_SPACE 0x00010108
- #define OID_GEN_RECEIVE_BUFFER_SPACE 0x00010109
- #define OID_GEN_TRANSMIT_BLOCK_SIZE 0x0001010A
- #define OID_GEN_RECEIVE_BLOCK_SIZE 0x0001010B
- #define OID_GEN_VENDOR_ID 0x0001010C
- #define OID_GEN_VENDOR_DESCRIPTION 0x0001010D
- #define OID_GEN_CURRENT_PACKET_FILTER 0x0001010E
- #define OID_GEN_CURRENT_LOOKAHEAD 0x0001010F
- #define OID_GEN_DRIVER_VERSION 0x00010110
- #define OID_GEN_MAXIMUM_TOTAL_SIZE 0x00010111
- #define OID_GEN_PROTOCOL_OPTIONS 0x00010112
- #define OID_GEN_MAC_OPTIONS 0x00010113
- #define OID_GEN_MEDIA_CONNECT_STATUS 0x00010114
- #define OID_GEN_MAXIMUM_SEND_PACKETS 0x00010115
- #define OID_GEN_VENDOR_DRIVER_VERSION 0x00010116
- #define OID_GEN_SUPPORTED_GUIDS 0x00010117
- #define OID_GEN_NETWORK_LAYER_ADDRESSES 0x00010118
- #define OID_GEN_TRANSPORT_HEADER_OFFSET 0x00010119
- #define OID_GEN_MEDIA_CAPABILITIES 0x00010201
- #define OID_GEN_PHYSICAL_MEDIUM 0x00010202
- #define OID_GEN_XMIT_OK 0x00020101
- #define OID_GEN_RCV_OK 0x00020102
- #define OID_GEN_XMIT_ERROR 0x00020103
- #define OID_GEN_RCV_ERROR 0x00020104
- #define OID_GEN_RCV_NO_BUFFER 0x00020105
- #define OID_802_3_PERMANENT_ADDRESS 0x01010101
- #define OID_802_3_CURRENT_ADDRESS 0x01010102
- #define OID_802_3_MULTICAST_LIST 0x01010103
- #define OID_802_3_MAXIMUM_LIST_SIZE 0x01010104
- #define OID_802_3_MAC_OPTIONS 0x01010105

- #define NDIS_802_3_MAC_OPTION_PRIORITY 0x00000001
- #define OID_802_3_RCV_ERROR_ALIGNMENT 0x01020101
- #define OID_802_3_XMIT_ONE_COLLISION 0x01020102
- #define OID_802_3_XMIT_MORE_COLLISIONS 0x01020103
- #define OID_802_3_XMIT_DEFERRED 0x01020201
- #define OID_802_3_XMIT_MAX_COLLISIONS 0x01020202
- #define OID_802_3_RCV_OVERRUN 0x01020203
- #define OID_802_3_XMIT_UNDERRUN 0x01020204
- #define OID_802_3_XMIT_HEARTBEAT_FAILURE 0x01020205
- #define OID_802_3_XMIT_TIMES_CRS_LOST 0x01020206
- #define OID_802_3_XMIT_LATE_COLLISIONS 0x01020207
- #define NDIS_PROTOCOL_ID_DEFAULT 0x00
- #define NDIS_PROTOCOL_ID_TCP_IP 0x02
- #define NDIS_PROTOCOL_ID_IPX 0x06
- #define NDIS_PROTOCOL_ID_NBF 0x07
- #define NDIS_PROTOCOL_ID_MAX 0x0F
- #define NDIS_PROTOCOL_ID_MASK 0x0F
- #define NDIS_PACKET_TYPE_DIRECTED 0x0001
- #define NDIS_PACKET_TYPE_MULTICAST 0x0002
- #define NDIS_PACKET_TYPE_ALL_MULTICAST 0x0004
- #define NDIS_PACKET_TYPE_BROADCAST 0x0008
- #define NDIS_PACKET_TYPE_SOURCE_ROUTING 0x0010
- #define NDIS_PACKET_TYPE_PROMISCUOUS 0x0020
- #define NDIS_PACKET_TYPE_SMT 0x0040
- #define NDIS_PACKET_TYPE_ALL_LOCAL 0x0080
- #define NDIS_PACKET_TYPE_MAC_FRAME 0x8000
- #define NDIS_PACKET_TYPE_FUNCTIONAL 0x4000
- #define NDIS_PACKET_TYPE_ALL_FUNCTIONAL 0x2000
- #define NDIS_PACKET_TYPE_GROUP 0x1000
- #define WORD_COUNT(packet) (sizeof(packet) / sizeof((packet)[0]))
- #define MIN(a, b) ((a) < (b) ? (a) : (b))
- #define NUM_ELEM(vector) (sizeof (vector) / sizeof ((vector)[0]))
- #define logMsg if (0) printf
- #define ERROR_TRACE logMsg
- #define ENUM_TRACE logMsg

- #define ENUM_VERBOSE_TRACE  if (0) logMsg
- #define RNDIS_TRACE  logMsg
- #define RNDIS_VERBOSE_TRACE  if (0) logMsg
- #define DUMP_ETHERNET_FRAME(mblk)  {  }
- #define DUMP_IP_FRAME(mblk)  {  }
- #define RNDIS_BUF  struct sk_buff
- #define RNDIS_BUF_LEN(buf)  buf->len
- #define RNDIS_BUF_DATA(buf)  buf->data
- #define RNDIS_SYS_ALIGN_BUF(buf, alignment)  { skb_reserve(buf, alignment); }
- #define RNDIS_SYS_ALLOC_BUF(len)  dev_alloc_skb(len)
- #define RNDIS_SYS_SET_BUF_LEN(buf, len)  { skb_put(buf, len); }
- #define RNDIS_SYS_FREE(buf)
- #define RNDIS_FREE(buf)  { IX_USB_MBLK_FREE(buf); }

### A.4.19.4 Typedefs

- typedef enum _NDIS_HARDWARE_STATUS NDIS_HARDWARE_STATUS
- typedef enum _NDIS_HARDWARE_STATUS * PNDIS_HARDWARE_STATUS
- typedef enum _NDIS_MEDIA_STATE NDIS_MEDIA_STATE
- typedef enum _NDIS_MEDIA_STATE * PNDIS_MEDIA_STATE
- typedef enum _NDIS_MEDIUM NDIS_MEDIUM
- typedef enum _NDIS_MEDIUM * PNDIS_MEDIUM
- typedef enum _NDIS_PHYSICAL_MEDIUM NDIS_PHYSICAL_MEDIUM
- typedef enum _NDIS_PHYSICAL_MEDIUM * PNDIS_PHYSICAL_MEDIUM

### A.4.19.5 Enumerations

- enum _NDIS_HARDWARE_STATUS { NdisHardwareStatusReady, NdisHardwareStatusInitializing, NdisHardwareStatusReset, NdisHardwareStatusClosing, NdisHardwareStatusNotReady }
- enum _NDIS_MEDIA_STATE { NdisMediaStateConnected, NdisMediaStateDisconnected }
- enum _NDIS_MEDIUM { NdisMedium802_3, NdisMedium802_5, NdisMediumFddi, NdisMediumWan, NdisMediumLocalTalk, NdisMediumDix, NdisMediumArcnetRaw, NdisMediumArcnet878_2, NdisMediumAtm, NdisMediumWirelessWan, NdisMediumIrda, NdisMediumBpc, NdisMediumCoWan, NdisMedium1394, NdisMediumMax }
- enum _NDIS_PHYSICAL_MEDIUM { NdisPhysicalMediumReserved0, NdisPhysicalMediumWirelessLan, NdisPhysicalMediumCableModem, NdisPhysicalMediumPhoneLine, NdisPhysicalMediumPowerLine, NdisPhysicalMediumDSL, NdisPhysicalMediumMax }
- enum RndisLayerState { RNDIS_HALTED =  0, RNDIS_INITIALIZED, RNDIS_DATA_INITIALIZED }

### A.4.19.6 Functions

- PUBLIC IX_STATUS ixUSBRNDISSignalEncapsulatedCommand (void)
  Function prototype for signalling encapsulation command.
- PUBLIC IX_STATUS ixUSBRNDISProcessEncapsulatedCommand (IX_USB_MBLK *)
  Function prototype for processing encapsulation command.
- PUBLIC IX_STATUS ixUSBRNDISProcessDataPacket (IX_USB_MBLK *)
  Function prototype for processing data packet.
- PUBLIC IX_STATUS ixUSBRNDISLayerInit (void *pDrvCtrl)
- PUBLIC IX_STATUS ixUSBRNDISInit (void)
- PUBLIC IX_USB_MBLK * ixUSBRNDISCreateMBuf (UINT8 *buffer, UINT32 len)
- PUBLIC IX_STATUS ixUSBRNDISSendDataPacket (RNDIS_BUF *packet)
  Function prototype for sending data packet. It is the hooks for the RNDIS END.
- void ixUSBRNDISIpHdrDump (const char *const mData)
  Function prototype for debugging purpose for dumping IP Header.
- void ixUSBRNDISEthernetHdrDump (const char *const mData, BOOL *nonIpHdrDetected)

### A.4.19.7 Detailed Description

Intel® IXP42X product line codelet USB RNDIS API.

How to use the USB RNDIS codelet:

1. Build a loadable object and load it into vxWorks
2. Start the codelet by typing ixUSBRNDISStart
3. You should see the “USB” network interface in the output generated by the ifShow command.
4. Plug the board into the USB port of a Windows 98/ME/2000 machine and selected the driver provided with the codelet when queried for it.

***Note:*** The IP and MAC addresses of the END driver (therefore the board side of the link) are defined in ixUSBRNDISEnd.h

the MAC address of the RNDIS driver (which will be used by Windows as its own MAC address) is defined in ixUSBRNDIS.h

Currently the END MAC address is 00:00:00:00:00:02 and the RNDIS MAC address is 00:00:00:00:00:01. The IP address of the END is 192.168.1.1, therefore you should use a compatible address for the RNDIS controller on the Windows side (such as 192.168.1.2) and set the END IP address as gateway address for the RNDIS network device, or change them to suitable values.

The codelet was tested with Windows* 2000 only, and telnet and ftp traffic was passed by routing the PC through the Intel® IXP42X product line into a network.

***Warning:*** This codelet is for demonstration purposes only, it should not be considered a fully working application.

## A.4.20 USB RNDIS End Driver Codelet (IxUSBRNDIS) API

Intel® IXP42X product line codelet for RNDIS End API.

### A.4.20.1 Defines

- #define RNDIS_END_INET_ADDR "192.168.1.1"
  RNDIS END driver with inet address.
- #define RNDIS_END_MAC_ADDRESS { 0x00, 0x00, 0x00, 0x00, 0x00, 0x02 }
  RNDIS END driver with MAC address.

### A.4.20.2 Functions

- PUBLIC void rndisInt (void *pDrvCtrl, int packet)
  RNDIS END hook for receiving packets.

### A.4.20.3 Detailed Description

Intel® IXP42X product line codelet for RNDIS End API.

## A.4.21 USB RNDIS Vendor Codelet (IxUSBRNDIS)

Intel® IXP42X product line codelet for RNDIS Vendor information.

### A.4.21.1 Defines

- #define RNDIS_VENDOR_ID (0x8086)
  RNDIS with static Intel Vendor ID.
- #define RNDIS_PRODUCT_ID (0x0001)
  RNDIS with static Product ID.
- #define RNDIS_VENDOR_DESCRIPTION “Intel Corporation”
  RNDIS with vendor description.
- #define RNDIS_MAC_ADDRESS { 0x00, 0x00, 0x00, 0x00, 0x00, 0x01 }
  RNDIS with MAC address.

### A.4.21.2 Detailed Description

Intel® IXP42X product line codelet for RNDIS Vendor information.

# A.5 Intel® IXP42X product line Access-Layer API and Codelet Data Structure Documentation

## A.5.1 Aal5Statistics Struct Reference

Types For Gathering Statistics.

`#include <IxFpathAccCodeletMain_p.h>`

### A.5.1.1 Data Fields

- unsigned int totalPduCount
- unsigned int validPduCount
- unsigned int errorCount

### A.5.1.2 Detailed Description

Types For Gathering Statistics.

## A.5.2 ActiveIpFlow Struct Reference

Struct to hold IP flow information while active.

`#include <IxFpathAccCodeletMain_p.h>`

### A.5.2.1 Data Fields

- BOOL aged
- UINT32 ipAddress
  Flow may be superseded.
- UINT8 macAddress [6]
  Destination IP address.
- int age
  Corresponding Mac Address.

### A.5.2.2 Detailed Description

Struct to hold IP flow information while active.

## A.5.3 ChannelisedStats Struct Reference

Type definition structure for channelised statistics.

`#include <IxHssAccCodeletCom.h>`

### A.5.3.1 Data Fields

- UINT32 txSamples
- UINT32 txBytes
- UINT32 rxSamples
- UINT32 rxBytes
- UINT32 rxIdles
- UINT32 rxVerifyFails
- UINT32 connectFails
- UINT32 portEnableFails
- UINT32 portDisableFails
- UINT32 disconnectFails

### A.5.3.2 Detailed Description

Type definition structure for channelised statistics.

## A.5.4 GeneralStats Struct Reference

Type definition structure for general statistics.

```
#include <IxHssAccCodeletCom.h>
```

### A.5.4.1 Data Fields

- UINT32 portInitFails
- UINT32 errorRetrievalFails
- UINT32 txFrmSyncErrors
- UINT32 rxFrmSyncErrors
- UINT32 txOverRunErrors
- UINT32 rxOverRunErrors
- UINT32 chanSwTxErrors
- UINT32 chanSwRxErrors
- UINT32 pktSwTxErrors
- UINT32 pktSwRxErrors
- UINT32 unrecognisedErrors

### A.5.4.2 Detailed Description

Type definition structure for general statistics.

## A.5.5 ix_s_ossl_thread_main_info_t Struct Reference

TYPENAME: ix_ossl_thread_main_info_t.

```
#include <ix_ossl.h>
```

### A.5.5.1 Data Fields

- ix_ossl_thread_entry_point_t threadMain
- void * threadArg

### A.5.5.2 Detailed Description

#### A.5.5.2.1 TYPENAME: ix_ossl_thread_main_info_t.

### A.5.5.3 Description

This type define thread main info. The struct has two fields: threadMain and threadArg. The threadMain is pointer to thread entry point function. threadArg is a void pointer to a user defined thread entry point argument.

## A.5.6 ix_s_ossl_time_t Struct Reference

#### A.5.6.0.1 TYPENAME: ix_ossl_time_t.

```
#include <ix_ossl.h>
```

### A.5.6.1 Data Fields

- unsigned long tv_sec
- unsigned long tv_nsec

### A.5.6.2 Detailed Description

#### A.5.6.2.1 TYPENAME: ix_ossl_time_t.

**Description**

This type defines OSSL time. The ix_ossl_time_t struct has two fields. 'sec' and 'nsec'. Time value is computed as: sec * 10^9 + nsec

## A.5.7 IxAtmdAccUtopiaConfig Struct Reference

Utopia configuration.

```
#include <IxAtmdAccCtrl.h>
```

### A.5.7.1 Data Fields

- IxAtmdAccUtopiaConfig::UtTxConfig_ utTxConfig
  Tx config Utopia register.
- IxAtmdAccUtopiaConfig::UtTxStatsConfig_ utTxStatsConfig
  Tx stats config Utopia register.
- IxAtmdAccUtopiaConfig::UtTxDefineIdle_ utTxDefineIdle
  Tx idle cell config Utopia register.
- IxAtmdAccUtopiaConfig::UtTxEnableFields_ utTxEnableFields
  Tx enable Utopia register.
- IxAtmdAccUtopiaConfig::UtTxTransTable0_ utTxTransTable0
  Tx translation table.
- IxAtmdAccUtopiaConfig::UtTxTransTable1_ utTxTransTable1
  Tx translation table.
- IxAtmdAccUtopiaConfig::UtTxTransTable2_ utTxTransTable2
  Tx translation table.
- IxAtmdAccUtopiaConfig::UtTxTransTable3_ utTxTransTable3
  Tx translation table.
- IxAtmdAccUtopiaConfig::UtTxTransTable4_ utTxTransTable4
  Tx translation table.
- IxAtmdAccUtopiaConfig::UtTxTransTable5_ utTxTransTable5
  Tx translation table.
- IxAtmdAccUtopiaConfig::UtRxConfig_ utRxConfig
  Rx config Utopia register.
- IxAtmdAccUtopiaConfig::UtRxStatsConfig_ utRxStatsConfig
  Rx stats config Utopia register.
- IxAtmdAccUtopiaConfig::UtRxDefineIdle_ utRxDefineIdle
  Rx idle cell config Utopia register.
- IxAtmdAccUtopiaConfig::UtRxEnableFields_ utRxEnableFields
  Rx enable Utopia register.
- IxAtmdAccUtopiaConfig::UtRxTransTable0_ utRxTransTable0
  Rx translation table.
- IxAtmdAccUtopiaConfig::UtRxTransTable1_ utRxTransTable1
  Rx translation table.
- IxAtmdAccUtopiaConfig::UtRxTransTable2_ utRxTransTable2
  Rx translation table.
- IxAtmdAccUtopiaConfig::UtRxTransTable3_ utRxTransTable3
  Rx translation table.
- IxAtmdAccUtopiaConfig::UtRxTransTable4_ utRxTransTable4

Rx translation table.

- IxAtmdAccUtopiaConfig::UtRxTransTable5_ utRxTransTable5
  Rx translation table.
- IxAtmdAccUtopiaConfig::UtSysConfig_ utSysConfig
  NPE debug config.

### A.5.7.2 Detailed Description

Utopia configuration.

This structure is used to set the UTOPIA parameters

contains the values of Utopia registers, to be set during initialization

contains debug commands for NPE, to be used during development steps

**Note**

The exact description of all parameters is done in the UTOPIA reference documents.

## A.5.8 IxAtmdAccUtopiaConfig::UtRxConfig_ Struct Reference

Utopia Rx config Register.

```
#include <IxAtmdAccCtrl.h>
```

### A.5.8.1 Data Fields

- unsigned int rxInterface:1
  [31] Utopia Receive Interface.
- unsigned int rxMode:1
  [30] Utopia Receive Mode.
- unsigned int rxOctet:1
  [29] Utopia Receive cell transfer protocol.
- unsigned int rxParity:1
  [28] Utopia Receive Parity Checking enable.
- unsigned int rxEvenParity:1
  [27] Utopia Receive Parity Mode
  - 1 - Check for Even Parity
  - 0 - Check for Odd Parity.
- unsigned int rxHEC:1
  [26] RxHEC Header Error Check Mode.
- unsigned int rxCOSET:1
  [25] If enabled the HEC is Exclusive-ORÆed with the value 0x55 before being tested with the received HEC.

- unsigned int rxHECpass:1

  [24] Specifies if the incoming cell HEC byte should be transferred after optional processing to the NPE2 Coprocessor Bus Interface or if it should be discarded.

unsigned int reserved_1:1

[23] These bits are always 0.

- unsigned int rxCellSize:7

  [22:16] Receive cell size.

- unsigned int rxHashEnbGFC:1

  [15] Specifies if the VPI field [11:8]/GFC field should be included in the Hash data input or if the bits should be padded with 1Æb0.

- unsigned int rxPreHash:1

  [14] Enable Pre-hash value generation.

- unsigned int reserved_2:1

  [13] These bits are always 0.

- unsigned int rxAddrRange:5

  [12:8] In ATM master, MPHY mode, this register specifies the upper limit of the PHY polling logical range.

- unsigned int reserved_3:3

  [7-5] These bits are always 0.

- unsigned int rxPHYAddr:5

  [4:0] When configured as a slave in an MPHY system this register specifies the physical address of the PHY.

### A.5.8.2 Detailed Description

Utopia Rx config Register.

### A.5.8.3 Field Documentation

#### A.5.8.3.1 unsigned int IxAtmdAccUtopiaConfig::UtRxConfig_::rxAddrRange

[12:8] In ATM master, MPHY mode, this register specifies the upper limit of the PHY polling logical range.

The number of active PHYs are RxAddrRange + 1.

#### A.5.8.3.2 unsigned int IxAtmdAccUtopiaConfig::UtRxConfig_::rxCellSize

[22:16] Receive cell size.

Configures the receive cell size. Values between 52-64 are valid

#### A.5.8.3.3 unsigned int IxAtmdAccUtopiaConfig::UtRxConfig_::rxCOSET

[25] If enabled the HEC is Exclusive-ORÆed with the value 0x55 before being tested with the received HEC.

- 1 - Enable HEC ExOR with value 0x55.
- 0 - Use generated HEC value.

#### A.5.8.3.4 unsigned int IxAtmdAccUtopiaConfig::UtRxConfig_::rxHashEnbGFC

[15] Specifies if the VPI field [11:8]/GFC field should be included in the Hash data input or if the bits should be padded with 1Æb0.

- 1 - VPI [11:8]/GFC field valid and used in Hash residue calculation.
- 0 - VPI [11:8]/GFC field padded with 1Æb0

#### A.5.8.3.5 unsigned int IxAtmdAccUtopiaConfig::UtRxConfig_::rxHEC

[26] RxHEC Header Error Check Mode.

Enables/disables cell header error checking on the received cell header.

— 1 - HEC checking enabled

— 0 - HEC checking disabled

#### A.5.8.3.6 unsigned int IxAtmdAccUtopiaConfig::UtRxConfig_::rxHECpass

[24] Specifies if the incoming cell HEC byte should be transferred after optional processing to the NPE2 Coprocessor Bus Interface or if it should be discarded.

- 1 - HEC maintained 53-byte/UDC cell sent to NPE2.
- 0 - HEC discarded 52-byte/UDC cell sent to NPE2 coprocessor.

#### A.5.8.3.7 unsigned int IxAtmdAccUtopiaConfig::UtRxConfig_::rxInterface

[31] Utopia Receive Interface.

The following encoding is used to set the UTOPIA Receive interface as ATM master or PHY slave:

- 1 - PHY
- 0 - ATM

#### A.5.8.3.8 unsigned int IxAtmdAccUtopiaConfig::UtRxConfig_::rxMode

[30] Utopia Receive Mode.

The following encoding is used to set the UTOPIA Receive mode to SPHY or MPHY:

- 1 - SPHY
- 0 - MPHY

#### A.5.8.3.9 unsigned int IxAtmdAccUtopiaConfig::UtRxConfig_::rxOctet

[29] Utopia Receive cell transfer protocol.

Used to set the UTOPIA cell transfer protocol to Octet-level handshaking. Note this is only applicable in SPHY mode.

- 1 - Octet-handshaking enabled
- 0 - Cell-handshaking enabled

#### A.5.8.3.10 unsigned int IxAtmdAccUtopiaConfig::UtRxConfig_::rxParity

[28] Utopia Receive Parity Checking enable.

- 1 - Parity checking enabled
- 0 - Parity checking disabled

#### A.5.8.3.11 unsigned int IxAtmdAccUtopiaConfig::UtRxConfig_::rxPreHash

[14] Enable Pre-hash value generation.

Specifies if the incoming cell data should be pre-hashed to allow VPI/VCI header look-up in a hash table.

- 1 - Pre-hashing enabled
- 0 - Pre-hashing disabled

## A.5.9 IxAtmdAccUtopiaConfig::UtRxDefineIdle_ Struct Reference

Utopia Rx idle cells config Register.

`#include <IxAtmdAccCtrl.h>`

### A.5.9.1 Data Fields

- unsigned int vpi:12
  [31:20] ATM VPI [11:0] OR GFC [3:0] and VPI [7:0]

***Note:*** if VCIdleRxGFC is set to 0 the GFC field is ignored in test.

- unsigned int vci:16
  [19:4] ATM VCI [15:0].
- unsigned int pti:3
  [3:1] ATM PTI PTI [2:0]

***Note:*** if VCIdleRxPTI is set to 0 the PTI field is ignored in test.

- unsigned int clp:1
  [0] ATM CLP [0]

***Note:*** if VCIdleRxCLP is set to 0 the CLP field is ignored in test.

### A.5.9.2 Detailed Description

Utopia Rx idle cells config Register.

## A.5.10 IxAtmdAccUtopiaConfig::UtRxEnableFields_ Struct Reference

Utopia Rx enable Register.

`#include <IxAtmdAccCtrl.h>`

### A.5.10.1 Data Fields

- unsigned int defineRxIdleGFC:1

  [31] This register is used to include or exclude the GFC field of the ATM header when testing for Idle cells.

- unsigned int defineRxIdlePTI:1

  [30] This register is used to include or exclude the PTI field of the ATM header when testing for Idle cells.

- unsigned int defineRxIdleCLP:1

  [29] This register is used to include or exclude the CLP field of the ATM header when testing for Idle cells.

- unsigned int phyStatsRxEnb:1

  [28] This register is used to enable or disable ATM statistics gathering based on the specified PHY address as defined in RxStatsConfig register.

- unsigned int vcStatsRxEnb:1

  [27] This register is used to enable or disable ATM statistics gathering based on a specific VPI/VCI address.

- unsigned int vcStatsRxGFC:1

  [26] This register is used to include or exclude the GFC field of the ATM header when ATM VPI/VCI statistics are enabled.

- unsigned int vcStatsRxPTI:1

  [25] This register is used to include or exclude the PTI field of the ATM header when ATM VPI/VCI statistics are enabled.

- unsigned int vcStatsRxCLP:1

  [24] This register is used to include or exclude the CLP field of the ATM header when ATM VPI/VCI statistics are enabled.

- unsigned int discardHecErr:1

  [23] Discard cells with an invalid HEC.

- unsigned int discardParErr:1

  [22] Discard cells containing parity errors.

- unsigned int discardIdle:1

  [21] Discard Idle Cells based on DefineIdle register values

  — 1 - Discard IDLE cells

  — 0 - IDLE cells passed.

- unsigned int enbHecErrCnt:1

  [20] Enable Receive HEC Error Count.

- unsigned int enbParErrCnt:1
  [19] Enable Parity Error Count
  - 1 - Enable count of received cells containing Parity errors
  - 0 - No count is maintained.
- unsigned int enbIdleCellCnt:1
  [18] Enable Receive Idle Cell Count.
- unsigned int enbSizeErrCnt:1
  [17] Enable Receive Size Error Count.
- unsigned int enbRxCellCnt:1
  [16] Enable Receive Valid Cell Count of non-idle/non-error cells.
- unsigned int reserved_1:3
  [15:13] These bits are always 0.
- unsigned int rxCellOvrInt:1
  [12] Enable CBI Utopia Receive Status Condition if the RxCellCount register overflows.
- unsigned int invalidHecOvrInt:1
  [11] Enable CBI Receive Status Condition if the InvalidHecCount register overflows.
- unsigned int invalidParOvrInt:1
  [10] Enable CBI Receive Status Condition if the InvalidParCount register overflows
  - 1 - CBI Receive Condition asserted.
- unsigned int invalidSizeOvrInt:1
  [9] Enable CBI Receive Status Condition if the InvalidSizeCount register overflows.
- unsigned int rxIdleOvrInt:1
  [8] Enable CBI Receive Status Condition if the RxIdleCount overflows.
- unsigned int reserved_2:3
  [7:5] These bits are always 0.
- unsigned int rxAddrMask:5
  [4:0] This register is used as a mask to allow the user to increase the PHY receive address range.

### A.5.10.2 Detailed Description

Utopia Rx enable Register.

### A.5.10.3 Field Documentation

#### A.5.10.3.1 unsigned int IxAtmdAccUtopiaConfig::UtRxEnableFields_::defineRxIdleCLP

[29] This register is used to include or exclude the CLP field of the ATM header when testing for Idle cells.

- 1 - CLP field is valid.
- 0 - CLP field ignored.

#### A.5.10.3.2 unsigned int IxAtmdAccUtopiaConfig::UtRxEnableFields_::defineRxIdleGFC

[31] This register is used to include or exclude the GFC field of the ATM header when testing for Idle cells.

- 1- GFC field is valid.
- 0 - GFC field ignored.

#### A.5.10.3.3 unsigned int IxAtmdAccUtopiaConfig::UtRxEnableFields_::defineRxIdlePTI

[30] This register is used to include or exclude the PTI field of the ATM header when testing for Idle cells.

- 1 - PTI field is valid.
- 0 - PTI field ignored.

#### A.5.10.3.4 unsigned int IxAtmdAccUtopiaConfig::UtRxEnableFields_::discardHecErr

[23] Discard cells with an invalid HEC.

- 1 - Discard cells with HEC errors
- 0 - Cells with HEC errors are passed

#### A.5.10.3.5 unsigned int IxAtmdAccUtopiaConfig::UtRxEnableFields_::discardParErr

[22] Discard cells containing parity errors.

- 1 - Discard cells with parity errors
- 0 - Cells with parity errors are passed

#### A.5.10.3.6 unsigned int IxAtmdAccUtopiaConfig::UtRxEnableFields_::enbHecErrCnt

[20] Enable Receive HEC Error Count.

- 1 - Enable count of received cells containing HEC errors
- 0 - No count is maintained.

#### A.5.10.3.7 unsigned int IxAtmdAccUtopiaConfig::UtRxEnableFields_::enbIdleCellCnt

[18] Enable Receive Idle Cell Count.

- 1 - Enable count of Idle cells received.
- 0 - No count is maintained.

#### A.5.10.3.8 unsigned int IxAtmdAccUtopiaConfig::UtRxEnableFields_::enbRxCellCnt

[16] Enable Receive Valid Cell Count of non-idle/non-error cells.

- 1 - Enable count of valid cells received - non-idle/non-error
- 0 - No count is maintained.

#### A.5.10.3.9 unsigned int IxAtmdAccUtopiaConfig::UtRxEnableFields_::enbSizeErrCnt

[17] Enable Receive Size Error Count.

- 1 - Enable count of received cells of incorrect size
- 0 - No count is maintained.

#### A.5.10.3.10 unsigned int IxAtmdAccUtopiaConfig::UtRxEnableFields_::invalidHecOvrInt

[11] Enable CBI Receive Status Condition if the InvalidHecCount register overflows.

- 1 - CBI Receive Condition asserted.
- 0 - No CBI Receive Condition asserted

#### A.5.10.3.11 unsigned int IxAtmdAccUtopiaConfig::UtRxEnableFields_::invalidParOvrInt

[10] Enable CBI Receive Status Condition if the InvalidParCount register overflows

- 1 - CBI Receive Condition asserted.
- 0 - No CBI Receive Condition asserted

#### A.5.10.3.12 unsigned int IxAtmdAccUtopiaConfig::UtRxEnableFields_::invalidSizeOvrInt

[9] Enable CBI Receive Status Condition if the InvalidSizeCount register overflows.

- 1 - CBI Receive Status Condition asserted. @li¸
- 0 - No CBI Receive Status asserted

#### A.5.10.3.13 unsigned int IxAtmdAccUtopiaConfig::UtRxEnableFields_::phyStatsRxEnb

[28] This register is used to enable or disable ATM statistics gathering based on the specified PHY address as defined in RxStatsConfig register.

- 1 - Enable statistics for specified receive PHY address.
- 0 - Disable statistics for specified receive PHY address.

#### A.5.10.3.14 unsigned int IxAtmdAccUtopiaConfig::UtRxEnableFields_::rxAddrMask

[4:0] This register is used as a mask to allow the user to increase the PHY receive address range.

The register should be programmed with the address-range limit, i.e. if set to 0x3 the address range increases to a maximum of 4 addresses.

#### A.5.10.3.15 unsigned int IxAtmdAccUtopiaConfig::UtRxEnableFields_::rxCellOvrInt

[12] Enable CBI Utopia Receive Status Condition if the RxCellCount register overflows.

- 1 - CBI Receive Status asserted.
- 0 - No CBI Receive Status asserted.

#### A.5.10.3.16 unsigned int IxAtmdAccUtopiaConfig::UtRxEnableFields_::rxIdleOvrInt

[8] Enable CBI Receive Status Condition if the RxIdleCount overflows.

- 1- CBI Receive Condition asserted.
- 0 - No CBI Receive Condition asserted

#### A.5.10.3.17 unsigned int IxAtmdAccUtopiaConfig::UtRxEnableFields_::vcStatsRxCLP

[24] This register is used to include or exclude the CLP field of the ATM header when ATM VPI/VCI statistics are enabled.

- 1 - CLP field is valid.
- 0 - CLP field ignored.

#### A.5.10.3.18 unsigned int IxAtmdAccUtopiaConfig::UtRxEnableFields_::vcStatsRxEnb

[27] This register is used to enable or disable ATM statistics gathering based on a specific VPI/VCI address.

- 1 - Enable statistics for specified VPI/VCI address.
- 0 - Disable statistics for specified VPI/VCI address.

#### A.5.10.3.19 unsigned int IxAtmdAccUtopiaConfig::UtRxEnableFields_::vcStatsRxGFC

[26] This register is used to include or exclude the GFC field of the ATM header when ATM VPI/VCI statistics are enabled.

GFC is only available at the UNI and uses the first 4-bits of the VPI field.

- 1 - GFC field is valid.
- 0 - GFC field ignored.

#### A.5.10.3.20 unsigned int IxAtmdAccUtopiaConfig::UtRxEnableFields_::vcStatsRxPTI

[25] This register is used to include or exclude the PTI field of the ATM header when ATM VPI/VCI statistics are enabled.

- 1 - PTI field is valid.
- 0 - PTI field ignored.

## A.5.11 IxAtmdAccUtopiaConfig::UtRxStatsConfig_ Struct Reference

Utopia Rx stats config Register.

```
#include <IxAtmdAccCtrl.h>
```

### A.5.11.1 Data Fields

- unsigned int vpi:12
  [31:20] ATM VPI VPI [11:0] OR GFC [3:0] and VPI [7:0]

***Note:*** if VCStatsRxGFC is set to 0 the GFC field is ignored in test.

- unsigned int vci:16
  [19:4] VCI [15:0] or PHY Address [4].
- unsigned int pti:3

[3:1] PTI [2:0] or or PHY Address [3:1]

***Note:*** if VCStatsRxPTI is set to 0 the PTI field is ignored in test.

- unsigned int clp:1
  [0] CLP [0] or PHY Address [0]

***Note:*** if VCStatsRxCLP is set to 0 the CLP field is ignored in test.

### A.5.11.2 Detailed Description

Utopia Rx stats config Register.

### A.5.11.3 Field Documentation

#### A.5.11.3.1 unsigned int IxAtmdAccUtopiaConfig::UtRxStatsConfig_::clp

[0] CLP [0] or PHY Address [0]

**Notes**

If VCStatsRxCLP is set to 0 the CLP field is ignored in test.

If VCStatsRxEnb is set to 0 only the PHY port address is used for statistics gathering.

#### A.5.11.3.2 unsigned int IxAtmdAccUtopiaConfig::UtRxStatsConfig_::pti

[3:1] PTI [2:0] or or PHY Address [3:1]

**Notes**

If VCStatsRxPTI is set to 0 the PTI field is ignored in test.

if VCStatsRxEnb is set to 0 only the PHY port address is used for statistics gathering.

## A.5.12 IxAtmdAccUtopiaConfig::UtRxTransTable0_ Struct Reference

Utopia Rx translation table Register.

`#include <IxAtmdAccCtrl.h>`

### A.5.12.1 Data Fields

- unsigned int phy0:5
  [31-27] Rx Mapping value of logical phy 0.
- unsigned int phy1:5
  [26-22] Rx Mapping value of logical phy 1.
- unsigned int phy2:5
  [21-17] Rx Mapping value of logical phy 2.

- unsigned int reserved_1:1
  [16] These bits are always 0.
- unsigned int phy3:5
  [15-11] Rx Mapping value of logical phy 3.
- unsigned int phy4:5
  [10-6] Rx Mapping value of logical phy 4.
- unsigned int phy5:5
  [5-1] Rx Mapping value of logical phy 5.
- unsigned int reserved_2:1
  [0] These bits are always 0.

### A.5.12.2 Detailed Description

Utopia Rx translation table Register.

## A.5.13 IxAtmdAccUtopiaConfig::UtRxTransTable1_ Struct Reference

Utopia Rx translation table Register.

`#include <IxAtmdAccCtrl.h>`

### A.5.13.1 Data Fields

- unsigned int phy6:5
  [31-27] Rx Mapping value of logical phy 6.
- unsigned int phy7:5
  [26-22] Rx Mapping value of logical phy 7.
- unsigned int phy8:5
  [21-17] Rx Mapping value of logical phy 8.
- unsigned int reserved_1:1
  [16-0] These bits are always 0.
- unsigned int phy9:5
  [15-11] Rx Mapping value of logical phy 3.
- unsigned int phy10:5
  [10-6] Rx Mapping value of logical phy 4.
- unsigned int phy11:5
  [5-1] Rx Mapping value of logical phy 5.
- unsigned int reserved_2:1
  [0] These bits are always 0.

### A.5.13.2 Detailed Description

Utopia Rx translation table Register.

## A.5.14 IxAtmdAccUtopiaConfig::UtRxTransTable2_ Struct Reference

Utopia Rx translation table Register.

`#include <IxAtmdAccCtrl.h>`

### A.5.14.1 Data Fields

- unsigned int phy12:5
  [31-27] Rx Mapping value of logical phy 6.
- unsigned int phy13:5
  [26-22] Rx Mapping value of logical phy 7.
- unsigned int phy14:5
  [21-17] Rx Mapping value of logical phy 8.
- unsigned int reserved_1:1
  [16-0] These bits are always 0.
- unsigned int phy15:5
  [15-11] Rx Mapping value of logical phy 3.
- unsigned int phy16:5
  [10-6] Rx Mapping value of logical phy 4.
- unsigned int phy17:5
  [5-1] Rx Mapping value of logical phy 5.
- unsigned int reserved_2:1
  [0] These bits are always 0.

### A.5.14.2 Detailed Description

Utopia Rx translation table Register.

## A.5.15 IxAtmdAccUtopiaConfig::UtRxTransTable3_ Struct Reference

Utopia Rx translation table Register.

`#include <IxAtmdAccCtrl.h>`

### A.5.15.1 Data Fields

- unsigned int phy18:5
  [31-27] Rx Mapping value of logical phy 6.

- unsigned int phy19:5
  [26-22] Rx Mapping value of logical phy 7.
- unsigned int phy20:5
  [21-17] Rx Mapping value of logical phy 8.
- unsigned int reserved_1:1
  [16-0] These bits are always 0.
- unsigned int phy21:5
  [15-11] Rx Mapping value of logical phy 3.
- unsigned int phy22:5
  [10-6] Rx Mapping value of logical phy 4.
- unsigned int phy23:5
  [5-1] Rx Mapping value of logical phy 5.
- unsigned int reserved_2:1
  [0] These bits are always 0.

### A.5.15.2 Detailed Description

Utopia Rx translation table Register.

## A.5.16 IxAtmdAccUtopiaConfig::UtRxTransTable4_ Struct Reference

Utopia Rx translation table Register.

`#include <IxAtmdAccCtrl.h>`

### A.5.16.1 Data Fields

- unsigned int phy24:5
  [31-27] Rx Mapping value of logical phy 6.
- unsigned int phy25:5
  [26-22] Rx Mapping value of logical phy 7.
- unsigned int phy26:5
  [21-17] Rx Mapping value of logical phy 8.
- unsigned int reserved_1:1
  [16-0] These bits are always 0.
- unsigned int phy27:5
  [15-11] Rx Mapping value of logical phy 3.
- unsigned int phy28:5
  [10-6] Rx Mapping value of logical phy 4.
- unsigned int phy29:5
  [5-1] Rx Mapping value of logical phy 5.

- unsigned int reserved_2:1
  [0] These bits are always 0.

### A.5.16.2 Detailed Description

Utopia Rx translation table Register.

## A.5.17 IxAtmdAccUtopiaConfig::UtRxTransTable5_ Struct Reference

Utopia Rx translation table Register.

`#include <IxAtmdAccCtrl.h>`

### A.5.17.1 Data Fields

- unsigned int phy30:5
  [31-27] Rx Mapping value of logical phy 6.
- unsigned int reserved_1:27
  [26-0] These bits are always 0.

### A.5.17.2 Detailed Description

Utopia Rx translation table Register.

## A.5.18 IxAtmdAccUtopiaConfig::UtSysConfig_ Struct Reference

NPE setup Register.

`#include <IxAtmdAccCtrl.h>`

### A.5.18.1 Data Fields

- unsigned int reserved_1:2
  [31-30] These bits are always 0.
- unsigned int txEnbFSM:1
  [29] Enables the operation of the Utopia Transmit FSM
  - 1 - FSM enabled
  - 0 - FSM inactive.
- unsigned int rxEnbFSM:1
  [28] Enables the operation of the Utopia Receive FSM
  - 1 - FSM enabled
  - 0 - FSM inactive.
- unsigned int disablePins:1
  [27] Disable Utopia interface I/O pins forcing the signals to an inactive state.

- unsigned int tstLoop:1
  [26] Test Loop Back Enable.
- unsigned int txReset:1
  [25] Resets the Utopia Coprocessor transmit module to a known state.
- unsigned int rxReset:1
  [24] Resets the Utopia Coprocessor receive module to a known state.
- unsigned int reserved_2:24
  [23-0] These bits are always 0.

### A.5.18.2 Detailed Description

NPE setup Register.

### A.5.18.3 Field Documentation

#### A.5.18.3.1 unsigned int IxAtmdAccUtopiaConfig::UtSysConfig_::disablePins

[27] Disable Utopia interface I/O pins forcing the signals to an inactive state.
Note that this bit is set on reset and must be de-asserted

- 0 - Normal data transfer
- 1 - Utopia interface pins are forced inactive

#### A.5.18.3.2 unsigned int IxAtmdAccUtopiaConfig::UtSysConfig_::rxReset

[24] Resets the Utopia Coprocessor receive module to a known state.

***Note:*** All receive configuration and status registers will be reset to their reset values.

- 0 - Normal operating mode
- 1 - Reset receive modules

#### A.5.18.3.3 unsigned int IxAtmdAccUtopiaConfig::UtSysConfig_::tstLoop

[26] Test Loop Back Enable.

***Note:*** For loop back to function RxMode and Tx Mode must both be set to single PHY mode.

- 0 - Loop back
- 1 - Normal operating mode

#### A.5.18.3.4 unsigned int IxAtmdAccUtopiaConfig::UtSysConfig_::txReset

[25] Resets the Utopia Coprocessor transmit module to a known state.

***Note:*** All transmit configuration and status registers will be reset to their reset values.

- 0 - Normal operating mode¸
- 1 - Reset transmit modules

## A.5.19 IxAtmdAccUtopiaConfig::UtTxConfig_ Struct Reference

Utopia Tx Config Register.

`#include <IxAtmdAccCtrl.h>`

### A.5.19.1 Data Fields

- unsigned int reserved_1:1
  [31] These bits are always 0.
- unsigned int txInterface:1
  [30] Utopia Transmit Interface.
- unsigned int txMode:1
  [29] Utopia Transmit Mode.
- unsigned int txOctet:1
  [28] Utopia Transmit cell transfer protocol.
- unsigned int txParity:1
  [27] Utopia Transmit parity enabled when set.
- unsigned int txEvenParity:1
  [26] Utopia Transmit Parity Mode
  - 1 - Even Parity Generated.
- unsigned int txHEC:1
  [25] Header Error Check Insertion Mode.
- unsigned int txCOSET:1
  [24] If enabled the HEC is Exclusive-ORÆed with the value 0x55 before being presented on the Utopia bus.
- unsigned int reserved_2:1
  [23] These bits are always 0.
- unsigned int txCellSize:7
  [22:16] Transmit expected cell size.
- unsigned int reserved_3:3
  [15:13] These bits are always 0.
- unsigned int txAddrRange:5
  [12:8] When configured as an ATM master in MPHY mode this register specifies the upper limit of the PHY polling logical range.
- unsigned int reserved_4:3
  [7:5] These bits are always 0.
- unsigned int txPHYAddr:5
  [4:0] When configured as a slave in an MPHY system this register specifies the physical address of the PHY.

### A.5.19.2 Detailed Description

Utopia Tx Config Register.

### A.5.19.3 Field Documentation

#### A.5.19.3.1 unsigned int IxAtmdAccUtopiaConfig::UtTxConfig_::txAddrRange

[12:8] When configured as an ATM master in MPHY mode this register specifies the upper limit of the PHY polling logical range.

The number of active PHYs are TxAddrRange + 1.

#### A.5.19.3.2 unsigned int IxAtmdAccUtopiaConfig::UtTxConfig_::txCellSize

[22:16] Transmit expected cell size.

Configures the cell size for the transmit module: Values between 52-64 are valid.

#### A.5.19.3.3 unsigned int IxAtmdAccUtopiaConfig::UtTxConfig_::txCOSET

[24] If enabled the HEC is Exclusive-ORÆed with the value 0x55 before being presented on the Utopia bus.

- 1 - Enable HEC ExOR with value 0x55
- 0 - Use generated HEC value.

#### A.5.19.3.4 unsigned int IxAtmdAccUtopiaConfig::UtTxConfig_::txEvenParity

[26] Utopia Transmit Parity Mode

- 1 - Even Parity Generated.
- 0 - Odd Parity Generated.

#### A.5.19.3.5 unsigned int IxAtmdAccUtopiaConfig::UtTxConfig_::txHEC

[25] Header Error Check Insertion Mode.

Specifies if the transmit cell header check byte is calculated and inserted when set.

- 1 - Generate HEC.
- 0 - Disable HEC generation.

#### A.5.19.3.6 unsigned int IxAtmdAccUtopiaConfig::UtTxConfig_::txInterface

[30] Utopia Transmit Interface.

The following encoding is used to set the UTOPIA Transmit interface as ATM master or PHY slave:

- 1 - PHY
- 0 - ATM

#### A.5.19.3.7 unsigned int IxAtmdAccUtopiaConfig::UtTxConfig_::txMode

[29] Utopia Transmit Mode.

The following encoding is used to set the Utopia Transmit mode to SPHY or MPHY:

- 1 - SPHY
- 0 - MPHY

#### A.5.19.3.8 unsigned int IxAtmdAccUtopiaConfig::UtTxConfig_::txOctet

[28] Utopia Transmit cell transfer protocol.

Used to set the UTOPIA cell transfer protocol to Octet-level handshaking. Note this is only applicable in SPHY mode.

- 1 - Octet-handshaking enabled
- 0 - Cell-handshaking enabled

#### A.5.19.3.9 unsigned int IxAtmdAccUtopiaConfig::UtTxConfig_::txParity

[27] Utopia Transmit parity enabled when set.

TxEvenParity defines the parity format odd/even.

- 1 - Enable Parity generation.
- 0 - ut_op_prty held low.

## A.5.20 IxAtmdAccUtopiaConfig::UtTxDefineIdle_ Struct Reference

Utopia Tx idle cells Register.

```
#include <IxAtmdAccCtrl.h>
```

### A.5.20.1 Data Fields

- unsigned int vpi:12
  [31:20] ATM VPI [11:0] OR GFC [3:0] and VPI [7:0]

***Note:*** if VCIdleTxGFC is set to 0 the GFC field is ignored in test.

- unsigned int vci:16
  [19:4] ATM VCI [15:0].
- unsigned int pti:3
  [3:1] ATM PTI PTI [2:0]

***Note:*** if VCIdleTxPTI is set to 0 the PTI field is ignored in test.

- unsigned int clp:1
  [0] ATM CLP [0]

***Note:*** if VCIdleTxCLP is set to 0 the CLP field is ignored in test.

### A.5.20.2 Detailed Description

Utopia Tx idle cells Register.

## A.5.21 IxAtmdAccUtopiaConfig::UtTxEnableFields_ Struct Reference

Utopia Tx ienable fields Register.

`#include <IxAtmdAccCtrl.h>`

### A.5.21.1 Data Fields

- unsigned int defineTxIdleGFC:1

  [31] This register is used to include or exclude the GFC field of the ATM header when testing for Idle cells.

- unsigned int defineTxIdlePTI:1

  [30] This register is used to include or exclude the PTI field of the ATM header when testing for Idle cells.

- unsigned int defineTxIdleCLP:1

  [29] This register is used to include or exclude the CLP field of the ATM header when testing for Idle cells.

- unsigned int phyStatsTxEnb:1

  [28] This register is used to enable or disable ATM statistics gathering based on the specified PHY address as defined in TxStatsConfig register.

- unsigned int vcStatsTxEnb:1

  [27] This register is used to change the ATM statistics-gathering mode from the specified logical PHY address to a specific VPI/VCI address.

- unsigned int vcStatsTxGFC:1

  [26] This register is used to include or exclude the GFC field of the ATM header when ATM VPI/VCI statistics are enabled.

- unsigned int vcStatsTxPTI:1

  [25] This register is used to include or exclude the PTI field of the ATM header when ATM VPI/VCI statistics are enabled.

- unsigned int vcStatsTxCLP:1

  [24] This register is used to include or exclude the CLP field of the ATM header when ATM VPI/VCI statistics are enabled.

- unsigned int reserved_1:3

  [23-21] These bits are always 0.

- unsigned int txPollStsInt:1

  [20] Enable the assertion of the ucp_tx_poll_sts condition where there is a change in polling status.

- unsigned int txCellOvrInt:1

  [19] Enable TxCellCount overflow CBI Transmit Status condition assertion.

- unsigned int txIdleCellOvrInt:1

[18] Enable TxIdleCellCount overflow Transmit Status Condition

— 1 - If TxIdleCellCountOvr is set assert the Transmit Status Condition

— 0 - No CBI Transmit Status condition assertion.

- unsigned int enbIdleCellCnt:1
  [17] Enable Transmit Idle Cell Count.
- unsigned int enbTxCellCnt:1
  [16] Enable Transmit Valid Cell Count of non-idle/non-error cells

  — 1 - Enable count of valid cells transmitted- non-idle/non-error

  — 0 - No count is maintained.
- unsigned int reserved_2:16
  [15:0] These bits are always 0.

### A.5.21.2 Detailed Description

Utopia Tx ienable fields Register.

### A.5.21.3 Field Documentation

#### A.5.21.3.1 unsigned int IxAtmdAccUtopiaConfig::UtTxEnableFields_::defineTxIdleCLP

[29] This register is used to include or exclude the CLP field of the ATM header when testing for Idle cells.

- 1 - CLP field is valid.
- 0 - CLP field ignored.

#### A.5.21.3.2 unsigned int IxAtmdAccUtopiaConfig::UtTxEnableFields_::defineTxIdleGFC

[31] This register is used to include or exclude the GFC field of the ATM header when testing for Idle cells.

- 1 - GFC field is valid.
- 0 - GFC field ignored.

#### A.5.21.3.3 unsigned int IxAtmdAccUtopiaConfig::UtTxEnableFields_::defineTxIdlePTI

[30] This register is used to include or exclude the PTI field of the ATM header when testing for Idle cells.

- 1 - PTI field is valid
- 0 - PTI field ignored.

#### A.5.21.3.4 unsigned int IxAtmdAccUtopiaConfig::UtTxEnableFields_::enbIdleCellCnt

[17] Enable Transmit Idle Cell Count.

- 1 - Enable count of Idle cells transmitted.
- 0 - No count is maintained.

#### A.5.21.3.5 unsigned int IxAtmdAccUtopiaConfig::UtTxEnableFields_::phyStatsTxEnb

[28] This register is used to enable or disable ATM statistics gathering based on the specified PHY address as defined in TxStatsConfig register.

- 1 - Enable statistics for specified transmit PHY address.
- 0 - Disable statistics for specified transmit PHY address.

#### A.5.21.3.6 unsigned int IxAtmdAccUtopiaConfig::UtTxEnableFields_::txCellOvrInt

[19] Enable TxCellCount overflow CBI Transmit Status condition assertion.

- 1 - If TxCellCountOvr is set assert the Transmit Status Condition.
- 0 - No CBI Transmit Status condition assertion

#### A.5.21.3.7 unsigned int IxAtmdAccUtopiaConfig::UtTxEnableFields_::txPollStsInt

[20] Enable the assertion of the ucp_tx_poll_sts condition where there is a change in polling status.

- 1 - ucp_tx_poll_sts asserted whenever there is a change in status
- 0 - ucp_tx_poll_sts asserted if ANY transmit PHY is available

#### A.5.21.3.8 unsigned int IxAtmdAccUtopiaConfig::UtTxEnableFields_::vcStatsTxCLP

[24] This register is used to include or exclude the CLP field of the ATM header when ATM VPI/VCI statistics are enabled.

- 1 - CLP field is valid
- 0 - CLP field ignored.

#### A.5.21.3.9 unsigned int IxAtmdAccUtopiaConfig::UtTxEnableFields_::vcStatsTxEnb

[27] This register is used to change the ATM statistics-gathering mode from the specified logical PHY address to a specific VPI/VCI address.

- 1 - Enable statistics for specified VPI/VCI address.
- 0 - Disable statistics for specified VPI/VCI address

#### A.5.21.3.10 unsigned int IxAtmdAccUtopiaConfig::UtTxEnableFields_::vcStatsTxGFC

[26] This register is used to include or exclude the GFC field of the ATM header when ATM VPI/VCI statistics are enabled.

GFC is only available at the UNI and uses the first 4-bits of the VPI field.

- 1 - GFC field is valid
- 0 - GFC field ignored.

#### A.5.21.3.11 unsigned int IxAtmdAccUtopiaConfig::UtTxEnableFields_::vcStatsTxPTI

[25] This register is used to include or exclude the PTI field of the ATM header when ATM VPI/VCI statistics are enabled.

- 1 - PTI field is valid
- 0 - PTI field ignored.

### A.5.21.4 IxAtmdAccUtopiaConfig::UtTxStatsConfig_ Struct Reference

Utopia Tx stats Register.

`#include <IxAtmdAccCtrl.h>`

### A.5.21.5 Data Fields

- unsigned int vpi:12
  [31:20] ATM VPI [11:0] OR GFC [3:0] and VPI [7:0]

***Note:*** if VCStatsTxGFC is set to 0 the GFC field is ignored in test.

- unsigned int vci:16
  [19:4] ATM VCI [15:0] or PHY Address[4].
- unsigned int pti:3
  [3:1] ATM PTI [2:0] or PHY Address[3:1]

***Note:*** if VCStatsTxPTI is set to 0 the PTI field is ignored in test.

- unsigned int clp:1
  [0] ATM CLP or PHY Address [0]

***Note:*** if VCStatsTxCLP is set to 0 the CLP field is ignored in test.

### A.5.21.6 Detailed Description

Utopia Tx stats Register.

### A.5.21.7 Field Documentation

#### A.5.21.7.1 unsigned int IxAtmdAccUtopiaConfig::UtTxStatsConfig_::clp

[0] ATM CLP or PHY Address [0]

**Notes**

if VCStatsTxCLP is set to 0 the CLP field is ignored in test.

if VCStatsTxEnb is set to 0 only the transmit PHY port address as defined by this register is used for ATM statistics [4:0].

#### A.5.21.7.2 unsigned int IxAtmdAccUtopiaConfig::UtTxStatsConfig_::pti

[3:1] ATM PTI [2:0] or PHY Address[3:1]

**Notes**

if VCStatsTxPTI is set to 0 the PTI field is ignored in test.

if VCStatsTxEnb is set to 0 only the transmit PHY port address as defined by this register is used for ATM statistics [4:0].

## A.5.22 IxAtmdAccUtopiaConfig::UtTxTransTable0_ Struct Reference

Utopia Tx translation table Register.

`#include <IxAtmdAccCtrl.h>`

### A.5.22.1 Data Fields

- unsigned int phy0:5
  [31-27] Tx Mapping value of logical phy 0.
- unsigned int phy1:5
  [26-22] Tx Mapping value of logical phy 1.
- unsigned int phy2:5
  [21-17] Tx Mapping value of logical phy 2.
- unsigned int reserved_1:1
  [16] These bits are always 0.
- unsigned int phy3:5
  [15-11] Tx Mapping value of logical phy 3.
- unsigned int phy4:5
  [10-6] Tx Mapping value of logical phy 4.
- unsigned int phy5:5
  [5-1] Tx Mapping value of logical phy 5.
- unsigned int reserved_2:1
  [0] These bits are always 0.

### A.5.22.2 Detailed Description

Utopia Tx translation table Register.

## A.5.23 IxAtmdAccUtopiaConfig::UtTxTransTable1_ Struct Reference

Utopia Tx translation table Register.

`#include <IxAtmdAccCtrl.h>`

### A.5.23.1 Data Fields

- unsigned int phy6:5
  [31-27] Tx Mapping value of logical phy 6.
- unsigned int phy7:5
  [26-22] Tx Mapping value of logical phy 7.
- unsigned int phy8:5

[21-17] Tx Mapping value of logical phy 8.

- unsigned int reserved_1:1
  [16-0] These bits are always 0.
- unsigned int phy9:5
  [15-11] Tx Mapping value of logical phy 3.
- unsigned int phy10:5
  [10-6] Tx Mapping value of logical phy 4.
- unsigned int phy11:5
  [5-1] Tx Mapping value of logical phy 5.
- unsigned int reserved_2:1
  [0] These bits are always 0.

### A.5.23.2 Detailed Description

Utopia Tx translation table Register.

## A.5.24 IxAtmdAccUtopiaConfig::UtTxTransTable2_ Struct Reference

Utopia Tx translation table Register.

`#include <IxAtmdAccCtrl.h>`

### A.5.24.1 Data Fields

- unsigned int phy12:5
  [31-27] Tx Mapping value of logical phy 6.
- unsigned int phy13:5
  [26-22] Tx Mapping value of logical phy 7.
- unsigned int phy14:5
  [21-17] Tx Mapping value of logical phy 8.
- unsigned int reserved_1:1
  [16-0] These bits are always 0.
- unsigned int phy15:5
  [15-11] Tx Mapping value of logical phy 3.
- unsigned int phy16:5
  [10-6] Tx Mapping value of logical phy 4.
- unsigned int phy17:5
  [5-1] Tx Mapping value of logical phy 5.
- unsigned int reserved_2:1
  [0] These bits are always 0.

### A.5.24.2 Detailed Description

Utopia Tx translation table Register.

## A.5.25 IxAtmdAccUtopiaConfig::UtTxTransTable3_ Struct Reference

Utopia Tx translation table Register.

`#include <IxAtmdAccCtrl.h>`

### A.5.25.1 Data Fields

- unsigned int phy18:5
  [31-27] Tx Mapping value of logical phy 6.
- unsigned int phy19:5
  [26-22] Tx Mapping value of logical phy 7.
- unsigned int phy20:5
  [21-17] Tx Mapping value of logical phy 8.
- unsigned int reserved_1:1
  [16-0] These bits are always 0.
- unsigned int phy21:5
  [15-11] Tx Mapping value of logical phy 3.
- unsigned int phy22:5
  [10-6] Tx Mapping value of logical phy 4.
- unsigned int phy23:5
  [5-1] Tx Mapping value of logical phy 5.
- unsigned int reserved_2:1
  [0] These bits are always 0.

### A.5.25.2 Detailed Description

Utopia Tx translation table Register.

## A.5.26 IxAtmdAccUtopiaConfig::UtTxTransTable4_ Struct Reference

Utopia Tx translation table Register.

`#include <IxAtmdAccCtrl.h>`

### A.5.26.1 Data Fields

- unsigned int phy24:5
  [31-27] Tx Mapping value of logical phy 6.

- unsigned int phy25:5
  [26-22] Tx Mapping value of logical phy 7.
- unsigned int phy26:5
  [21-17] Tx Mapping value of logical phy 8.
- unsigned int reserved_1:1
  [16-0] These bits are always 0.
- unsigned int phy27:5
  [15-11] Tx Mapping value of logical phy 3.
- unsigned int phy28:5
  [10-6] Tx Mapping value of logical phy 4.
- unsigned int phy29:5
  [5-1] Tx Mapping value of logical phy 5.
- unsigned int reserved_2:1
  [0] These bits are always 0.

### A.5.26.2 Detailed Description

Utopia Tx translation table Register.

## A.5.27 IxAtmdAccUtopiaConfig::UtTxTransTable5_ Struct Reference

Utopia Tx translation table Register.

```
#include <IxAtmdAccCtrl.h>
```

### A.5.27.1 Data Fields

- unsigned int phy30:5
  [31-27] Tx Mapping value of logical phy 6.
- unsigned int reserved_1:27
  [26-0] These bits are always 0.

### A.5.27.2 Detailed Description

Utopia Tx translation table Register.

## A.5.28 IxAtmdAccUtopiaStatus Struct Reference

Utopia status.

```
#include <IxAtmdAccCtrl.h>
```

### A.5.28.1 Data Fields

- unsigned int utTxCellCount
  count of cells transmitted.
- unsigned int utTxIdleCellCount
  count of idle cells transmitted.
- IxAtmdAccUtopiaStatus::UtTxCellConditionStatus_ utTxCellConditionStatus
  Tx cells condition status.
- unsigned int utRxCellCount
  count of cell received.
- unsigned int utRxIdleCellCount
  count of idle cell received.
- unsigned int utRxInvalidHECount
  count of invalid cell received because of HEC errors.
- unsigned int utRxInvalidParCount
  count of invalid cell received because of parity errors.
- unsigned int utRxInvalidSizeCount
  count of invalid cell received because of cell size errors.
- IxAtmdAccUtopiaStatus::UtRxCellConditionStatus_ utRxCellConditionStatus
  Rx cells condition status.

### A.5.28.2 Detailed Description

Utopia status.

This structure is used to set/get the Utopia status parameters
contains debug cell counters, to be accessed during a read operation

**Note**

the exact description of all parameters is done in the Utopia reference documents.

## A.5.29 IxAtmdAccUtopiaStatus::UtRxCellConditionStatus_ Struct Reference

Utopia Rx Status Register.

`#include <IxAtmdAccCtrl.h>`

### A.5.29.1 Data Fields

- unsigned int reserved_1:3
  [31:29] These bits are always 0.
- unsigned int rxCellCountOvr:1
  [28] This bit is set if the RxCellCount register overflows.

- unsigned int invalidHecCountOvr:1
  [27] This bit is set if the InvalidHecCount register overflows.
- unsigned int invalidParCountOvr:1
  [26] This bit is set if the InvalidParCount register overflows.
- unsigned int invalidSizeCountOvr:1
  [25] This bit is set if the InvalidSizeCount register overflows.
- unsigned int rxIdleCountOvr:1
  [24] This bit is set if the RxIdleCount register overflows.
- unsigned int reserved_2:4
  [23:20] These bits are always 0.
- unsigned int rxFIFO2Underflow:1
  [19] This bit is set if 64-byte Receive FIFO2 indicates a FIFO underflow error condition.
- unsigned int rxFIFO1Underflow:1
  [18] This bit is set if 64-byte Receive FIFO1 indicates a FIFO underflow error condition.
- unsigned int rxFIFO2Overflow:1
  [17] This bit is set if 64-byte Receive FIFO2 indicates a FIFO overflow error condition.
- unsigned int rxFIFO1Overflow:1
  [16] This bit is set if 64-byte Receive FIFO1 indicates a FIFO overflow error condition.
- unsigned int reserved_3:16
  [15:0] These bits are always 0.

### A.5.29.2 Detailed Description

Utopia Rx Status Register.

## A.5.30 IxAtmdAccUtopiaStatus::UtTxCellConditionStatus_ Struct Reference

Utopia Tx Status Register.

`#include <IxAtmdAccCtrl.h>`

### A.5.30.1 Data Fields

- unsigned int reserved_1:2
  [31:30] These bits are always 0.
- unsigned int txFIFO2Underflow:1
  [29] This bit is set if 64-byte Transmit FIFO2 indicates a FIFO underflow error condition.
- unsigned int txFIFO1Underflow:1
  [28] This bit is set if 64-byte Transmit FIFO1 indicates a FIFO underflow error condition.
- unsigned int txFIFO2Overflow:1
  [27] This bit is set if 64-byte Transmit FIFO2 indicates a FIFO overflow error condition.

- unsigned int txFIFO1Overflow:1
  [26] This bit is set if 64-byte Transmit FIFO1 indicates a FIFO overflow error condition.
- unsigned int txIdleCellCountOvr:1
  [25] This bit is set if the TxIdleCellCount register overflows.
- unsigned int txCellCountOvr:1
  [24] This bit is set if the TxCellCount register overflows.
- unsigned int reserved_2:24
  [23:0] These bits are always 0.

#### A.5.30.2 Detailed Description

Utopia Tx Status Register.

### A.5.31 IxAtmmPortCfg Struct Reference

Structure contains port-specific information required to initialize IxAtmm, and specifically, the Intel® IXP42X product line UTOPIA Level-2 device.

```
#include <IxAtmm.h>
```

#### A.5.31.1 Data Fields

- unsigned reserved_1:11
  [31:21] Should be zero.
- unsigned UtopiaTxPhyAddr:5
  [20:16] Address of the transmit (Tx) PHY for this port on the 5-bit UTOPIA Level-2 address bus.
- unsigned reserved_2:11
  [15:5] Should be zero.
- unsigned UtopiaRxPhyAddr:5
  [4:0] Address of the receive (Rx) PHY for this port on the 5-bit UTOPIA Level-2 address bus.

#### A.5.31.2 Detailed Description

Structure contains port-specific information required to initialize IxAtmm, and specifically, the Intel® IXP42X product line UTOPIA Level-2 device.

### A.5.32 IxAtmmVc Struct Reference

This structure describes the required attributes of a virtual connection.

```
#include <IxAtmm.h>
```

#### A.5.32.1 Data Fields

- unsigned vpi

VPI value of this virtual connection.

- unsigned vci
  VCI value of this virtual connection.
- IxAtmmVcDirection direction
  VC direction.
- IxAtmTrafficDescriptor trafficDesc
  Traffic descriptor of this virtual connection.

### A.5.32.2 Detailed Description

This structure describes the required attributes of a virtual connection.

### A.5.32.3 Field Documentation

IxAtmTrafficDescriptor IxAtmmVc::trafficDesc

Traffic descriptor of this virtual connection.

This structure is defined by the Intel® IXP42X product line ATM Transmit Scheduler (IxAtmSch) API component.

## A.5.33 IxAtmScheduleTable Struct Reference

This structure defines a schedule table which gives details on which data (from which VCs) should be transmitted for a forthcoming period of time for a particular port and the order in which that data should be transmitted.

```
#include <IxAtmTypes.h>
```

### A.5.33.1 Data Fields

- unsigned tableSize
  Number of entries.
- unsigned totalCellSlots
  Number of cells.
- IxAtmScheduleTableEntry * table
  Pointer to schedule entries.

### A.5.33.2 Detailed Description

This structure defines a schedule table which gives details on which data (from which VCs) should be transmitted for a forthcoming period of time for a particular port and the order in which that data should be transmitted.

The schedule table consists of a series of entries each of which will schedule one or more cells from a particular registered VC. The total number of cells scheduled and the total number of entries in the table are also indicated.

### A.5.33.3 Field Documentation

#### A.5.33.3.1 IxAtmScheduleTableEntry* IxAtmScheduleTable::table

Pointer to schedule entries.

Pointer to an array containing tableSize entries

#### A.5.33.3.2 unsigned IxAtmScheduleTable::tableSize

Number of entries.

Indicates the total number of entries in the table.

#### A.5.33.3.3 unsigned IxAtmScheduleTable::totalCellSlots

Number of cells.

Indicates the total number of ATM cells which are scheduled by all the entries in the table.

## A.5.34 IxAtmScheduleTableEntry Struct Reference

ATM Schedule Table entry.

```
#include <IxAtmTypes.h>
```

### A.5.34.1 Data Fields

- IxAtmConnId connId
  connection Id
- unsigned int numberOfCells
  number of cells to transmit

### A.5.34.2 Detailed Description

ATM Schedule Table entry.

This IxAtmScheduleTableEntry is used by an ATM scheduler to inform IxAtmdAcc about the data to transmit (in term of cells per VC)

This structure defines:

- number of cells to be transmitted (numberOfCells).
- VC connection to be used for transmission (connId).

**Note**

When the connection Id value is IX_ATM_IDLE_CELLS_CONNID, the corresponding number of idle cells will be transmitted to the hardware.

### A.5.34.3 Field Documentation

#### A.5.34.3.1 IxAtmConnId IxAtmScheduleTableEntry::connId

connection Id.

Identifier of VC from which cells are to be transmitted. When this value is IX_ATM_IDLE_CELLS_CONNID, this indicates that the system should transmit the specified number of idle cells. Unknown connIds result in the transmission idle cells.

#### A.5.34.3.2 unsigned int IxAtmScheduleTableEntry::numberOfCells

number of cells to transmit.

The number of contiguous cells to schedule from this VC at this point. The valid range is from 1 to IX_ATM_SCHEDULETABLE_MAXCELLS_PER_ENTRY. This number can swap over mbufs and pdus. OverScheduling results in the transmission of idle cells.

## A.5.35 IxAtmTrafficDescriptor Struct Reference

Structure describing an ATM traffic contract for a Virtual Connection (VC).

```
#include <IxAtmTypes.h>
```

### A.5.35.1 Data Fields

- IxAtmServiceCategory atmService
  ATM service category.
- unsigned pcr
  Peak Cell Rate - cells per second.
- unsigned cdvt
  Cell Delay Variation Tolerance - in nanoseconds.
- unsigned scr
  Sustained Cell Rate - cells per second.
- unsigned mbs
  Max Burst Size - cells.
- unsigned mcr
  Minimum Cell Rate - cells per second.
- unsigned mfs
  Max Frame Size - cells.

### A.5.35.2 Detailed Description

Structure describing an ATM traffic contract for a Virtual Connection (VC).

Structure is used to specify the requested traffic contract for a VC to the IxAtmSch component using the ixAtmSchVcModelSetup interface.

## A.5.36 IxCryptoAccAuthCtx Struct Reference

Structure storing authentication configuration parameters required to perform security functionality.

`#include <IxCryptoAcc.h>`

### A.5.36.1 Data Fields

- IxCryptoAccAuthAlgo authAlgo
  authentication algorithm - MD5, SHA1.
- UINT32 authDigestLen
  Digest length in bytes.
- UINT32 authKeyLen
  Hash key length in bytes.
- IxCryptoAccAuthCtx:: { ... } key
  Hash key.
- UINT8 authKey [IX_CRYPTO_ACC_MAX_AUTH_KEY_LENGTH]
  default hash key array.
- UINT8 sha1Key [IX_CRYPTO_ACC_SHA1_KEY_160]
  SHA1 key.
- UINT8 md5Key [IX_CRYPTO_ACC_MD5_KEY_128]
  MD5 key.

### A.5.36.2 Detailed Description

Structure storing authentication configuration parameters required to perform security functionality.

Structure is used to specify the authentication required for using the Intel® IXP42X product line Security (IxCryptoAcc) API interface.

## A.5.37 IxCryptoAccCipherCtx Struct Reference

Structure storing cipher configuration parameters required to perform security functionality.

`#include <IxCryptoAcc.h>`

### A.5.37.1 Data Fields

- IxCryptoAccCipherAlgo cipherAlgo
  Cipher Algorithm - DES, 3DES, AES.
- IxCryptoAccCipherMode cipherMode
  Cipher mode - ECB, CBC, CTR.
- UINT32 cipherKeyLen
  Cipher key length in bytes.

- IxCryptoAccCipherCtx:: { ... } key
  Cipher key.
- UINT32 cipherBlockLen
  Input plaintext/ciphertext blocks size in bytes.
- UINT32 cipherInitialVectorLen
  Length of IV in bytes.
- UINT32 securityParameterIndex
  SPI is used in generating counter block for AES CTR mode.
- UINT8 cipherKey [IX_CRYPTO_ACC_MAX_CIPHER_KEY_LENGTH]
  default key array.
- UINT8 desKey [IX_CRYPTO_ACC_DES_KEY_64]
  DES key.
- UINT8 tripleDesKey [IX_CRYPTO_ACC_3DES_KEY_192]
  3DES key.
- UINT8 aesKey128 [IX_CRYPTO_ACC_AES_KEY_128]
  AES-128 key.
- UINT8 aesKey192 [IX_CRYPTO_ACC_AES_KEY_192]
  AES-192 key.
- UINT8 aesKey256 [IX_CRYPTO_ACC_AES_KEY_256]
  AES-256 key.

### A.5.37.2 Detailed Description

Structure storing cipher configuration parameters required to perform security functionality.

Structure is used to specify the cipher context required for using the Intel® IXP42X product line Security (IxCryptoAcc) API interface.

## A.5.38 IxCryptoAccCtx Struct Reference

Structure storing configuration parameters required to perform security functionality.

```
#include <IxCryptoAcc.h>
```

### A.5.38.1 Data Fields

- IxCryptoAccOperation operation
  Types of operation.
  - IxCryptoAccCipherCtx cipherCtx
  - IxCryptoAccAuthCtx authCtx
  - BOOL useDifferentSrcAndDestMbufs
    - If TRUE, data read from srcMbuf, results written to destMbuf, if FALSE data read from srcMbuf, written back to srcMbuf.

### A.5.38.2 Detailed Description

Structure storing configuration parameters required to perform security functionality.

Structure is used to specify the context required for using the Intel® IXP42X product line Security (IxCryptoAcc) API interface.

### A.5.38.3 Field Documentation

#### A.5.38.3.1 BOOL IxCryptoAccCtx::useDifferentSrcAndDestMbufs

If TRUE, data read from srcMbuf, results written to destMbuf, if FALSE data read from srcMbuf, written back to srcMbuf.

Default is FALSE. Note that only the crypted/authenticated data that is copied, not the entire source mbuf.

## A.5.39 IxEthAccMacAddr Struct Reference

The IEEE 802.3 Ethernet MAC address structure.

```
#include <IxEthAcc.h>
```

### A.5.39.1 Data Fields

- UINT8 **macAddress** [IX_IEEE803_MAC_ADDRESS_SIZE]

### A.5.39.2 Detailed Description

The IEEE 802.3 Ethernet MAC address structure.

The data should be packed with bytes xx:xx:xx:xx:xx:xx

**Note**

The data must be packed in network byte order.

## A.5.40 IxEthDBMacAddr Struct Reference

The IEEE 802.3 Ethernet MAC address structure.

```
#include <IxEthDB.h>
```

### A.5.40.1 Data Fields

- UINT8 macAddress [IX_IEEE803_MAC_ADDRESS_SIZE]

### A.5.40.2 Detailed Description

The IEEE 802.3 Ethernet MAC address structure.

The data should be packed with bytes xx:xx:xx:xx:xx:xx

**Note**

The data must be packed in network byte order.

## A.5.41 IxEthDBPortDefinition Struct Reference

Port Definition - a structure contains the Port type and capabilities.

```
#include <IxEthDBPortDefs.h>
```

### A.5.41.1 Data Fields

- IxEthDBPortType type
- IxEthDBPortCapability capabilities

### A.5.41.2 Detailed Description

Port Definition - a structure contains the Port type and capabilities.

## A.5.42 IxEthEthObjStats Struct Reference

This struct defines the statistics returned by this component. The component returns MIB2 EthObj variables which should are obtained from the hardware or maintained by this component.

```
#include <IxEthAcc.h>
```

### A.5.42.1 Data Fields

- UINT32 dot3StatsAlignmentErrors
- UINT32 dot3StatsFCSErrors
- UINT32 dot3StatsFrameTooLongs
- UINT32 dot3StatsInternalMacReceiveErrors
- UINT32 LearnedEntryDiscards
- UINT32 dot3StatsSingleCollisionFrames
- UINT32 dot3StatsMultipleCollisionFrames
- UINT32 dot3StatsDeferredTransmissions
- UINT32 dot3StatsLateCollisions
- UINT32 dot3StatsExcessiveCollsions
- UINT32 dot3StatsInternalMacTransmitErrors
- UINT32 dot3StatsCarrierSenseErrors

### A.5.42.2 Detailed Description

This struct defines the statistics returned by this component. The component returns MIB2 EthObj variables which should are obtained from the hardware or maintained by this component.

## A.5.43 IxFpCodeletFastPathConnection Struct Reference

Main structure to hold connection information.

`#include <IxFpathAccCodeletMain_p.h>`

### A.5.43.1 Data Fields

- BOOL inUse
- VcNumber vc
- IxAtmLogicalPort port
- unsigned int vpi
- unsigned int vci
- EncapsulationType encapsulationType
- AtmRxCallback atmRxHandler
  See IxFpathCodeletTemplates_p.h.
- IxAtmNpeRxVcId npeVcId
- LearningIpFlow learningIpFlow [NUM_LEARNING_FLOWS]
  IP Flow Learning and Aging Information.
- ActiveIpFlow activeIpFlow [MAX_FLOWS_PER_VC]
- BOOL ipFlowRequest
- IpFlowRequestData ipFlowRequestData
- Aal5Statistics aal5Stats
  Statistics.
- EtherBridgeStatistics etherBridgeStats
- RoutedIPStatistics routedIpStats

### A.5.43.2 Detailed Description

Main structure to hold connection information.

## A.5.44 IxFpCodeletTemplate Struct Reference

This type describes a classifier or modifier template.

`#include <IxFpathAccCodeletTemplates_p.h>`

### A.5.44.1 Data Fields

UINT8 * data

unsigned int len

### A.5.44.2 Detailed Description

This type describes a classifier or modifier template.

## A.5.45 IxHssAccCodeletStats Struct Reference

Type definition structure for HSS Access Codelet statistics.

```
#include <IxHssAccCodeletCom.h>
```

### A.5.45.1 Data Fields

- GeneralStats gen
- ChannelisedStats chan
- PacketisedStats pkt [IX_HSSACC_HDLC_PORT_MAX]

### A.5.45.2 Detailed Description

Type definition structure for HSS Access Codelet statistics.

## A.5.46 IxHssAccConfigParams Struct Reference

Structure containing HSS configuration parameters.

```
#include <IxHssAcc.h>
```

### A.5.46.1 Data Fields

- IxHssAccPortConfig txPortConfig
  HSS tx port configuration.
- IxHssAccPortConfig rxPortConfig
  HSS rx port configuration.
- unsigned numChannelised
  The number of channelised timeslots (0-32).
- unsigned hssPktChannelCount
  The number of packetised clients (0 - 4).
- UINT8 channelisedIdlePattern
  The byte to be transmitted on channelised service when there is no client data to tx.
- BOOL loopback
  The HSS loopback state.
- unsigned packetisedIdlePattern
  The data to be transmitted on packetised service when there is no client data to tx.
- IxHssAccClkSpeed clkSpeed
  The HSS clock speed.

### A.5.46.2 Detailed Description

Structure containing HSS configuration parameters.

## A.5.47 IxHssAccPktHdlcFraming Struct Reference

This structure contains information required by the NPE to configure the HDLC co-processor.

```
#include <IxHssAcc.h>
```

### A.5.47.1 Data Fields

- IxHssAccPktHdlcIdleType hdlcIdleType
  What to transmit when a HDLC port is idle.
- IxHssAccBitEndian dataEndian
  The HDLC data endianness.
- IxHssAccPktCrcType crcType
  The CRC type to be used for this HDLC port.

### A.5.47.2 Detailed Description

This structure contains information required by the NPE to configure the HDLC co-processor.

## A.5.48 IxHssAccPortConfig Struct Reference

Structure containing HSS port configuration parameters.

```
#include <IxHssAcc.h>
```

### A.5.48.1 Data Fields

- IxHssAccFrmSyncType frmSyncType
  frame sync pulse type (tx/rx).
- IxHssAccFrmSyncEnable frmSyncIO
  how the frame sync pulse is used (tx/rx).
- IxHssAccClkEdge frmSyncClkEdge
  frame sync clock edge type (tx/rx).
- IxHssAccClkEdge dataClkEdge
  data clock edge type (tx/rx).
- IxHssAccClkDir clkDirection
  clock direction (tx/rx).
- IxHssAccFrmPulseUsage frmPulseUsage
  whether to use the frame sync pulse or not (tx/rx).
- IxHssAccDataRate dataRate
  data rate in relation to the clock (tx/rx).

- IxHssAccDataPolarity dataPolarity
  data polarity type (tx/rx).
- IxHssAccBitEndian dataEndianness
  data endianness (tx/rx).
- IxHssAccDrainMode drainMode
  tx pin open drain mode (tx).
- IxHssAccSOFType fBitUsage
  start of frame types (tx/rx).
- IxHssAccDataEnable dataEnable
  whether or not to drive the data pins (tx).
- IxHssAccTxSigType voice56kType
  how to drive the data pins for voice56k type (tx).
- IxHssAccTxSigType unassignedType
  how to drive the data pins for unassigned type (tx).
- IxHssAccFbType fBitType
  how to drive the Fbit (tx).
- IxHssAcc56kEndianness voice56kEndian
  56k data endianness when using the 56k type (tx).
- IxHssAcc56kSel voice56kSel
  56k data transmission type when using the 56k type (tx).
- unsigned frmOffset
  frame pulse offset in bits wrt the first timeslot (0-1023) (tx/rx).
- unsigned maxFrmSize
  frame size in bits (1-1024) (tx/rx).

### A.5.48.2 Detailed Description

Structure containing HSS port configuration parameters.

***Note:*** All of these are used for TX. Only some are specific to RX.

## A.5.49 IxMbufPool Struct Reference

Implementation of buffer pool structure for use with non-VxWorks OS.

```
#include <IxOsBuffPoolMgt.h>
```

### A.5.49.1 Data Fields

- IX_MBUF * nextFreeBuf
  Pointer to the next free mbuf.
- void * mbufMemPtr
  Pointer to the mbuf memory area.

- void * dataMemPtr
  Pointer to the data memory area.
- int bufDataSize
  The size of the data portion of each mbuf.
- int totalBufsInPool
  Total number of mbufs in the pool.
- int freeBufsInPool
  Number of free mbufs currently in the pool.
- int mbufMemSize
  The size of the pool mbuf memory area.
- int dataMemSize
  The size of the pool data memory area.
- char name [IX_MBUF_POOL_NAME_LEN+1]
  Descriptive name for pool.

### A.5.49.2 Detailed Description

Implementation of buffer pool structure for use with non-VxWorks OS.

## A.5.50 IxNpeA_AtmCellHeader Struct Reference

Atm cell header definition.

```
#include <IxNpeA.h>
```

### A.5.50.1 Data Fields

- unsigned int gfc:4
- unsigned int vpi:8
- unsigned int vci:16
- unsigned int pti:3
- unsigned int clp:1

### A.5.50.2 Detailed Description

Atm cell header definition.

## A.5.51 IxNpeA_AtmVcFp Struct Reference

Atm Descriptor structure used for fpath.

```
#include <IxNpeA.h>
```

### A.5.51.1 Data Fields

- IxNpeA_RxAtmVc npe
- UINT32 masterMbufLen
- IX_MBUF * pRootMbuf

### A.5.51.2 Detailed Description

Atm Descriptor structure used for fpath.

## A.5.52 IxNpeA_NpePacketDescriptor Struct Reference

HSS Packetised NpePacket Descriptor Structure.

```
#include <IxNpeA.h>
```

### A.5.52.1 Data Fields

- UINT8 status
- UINT8 errorCount
- UINT8 chainCount
- UINT8 rsvdByte0
- UINT16 packetLength
- UINT16 rsvdShort0
- IX_MBUF * pRootMbuf
- IX_MBUF * pNextMbuf
- unsigned char * pMbufData
- UINT32 mbufLength

### A.5.52.2 Detailed Description

HSS Packetised NpePacket Descriptor Structure.

## A.5.53 IxNpeA_RxAtmVc Struct Reference

Rx Descriptor definition.

```
#include <IxNpeA.h>
```

### A.5.53.1 Data Fields

- IxNpeA_RxAtmVc:: { ... } rxBitField
- IxNpeA_AtmCellHeader atmCellHeader
- UINT32 pSlowPathDesc

- UINT16 currMbufLen
- UINT8 fpIndex
- UINT8 fpMatchIndex
- unsigned char * pFpMbufData
- IX_MBUF * pCurrMbuf
- unsigned char * pCurrMbufData
- IX_MBUF * pNextMbuf
- UINT32 totalLen
- UINT32 aal5CrcResidue
- unsigned int status:1
- unsigned int port:7
- unsigned int vcId:8
- unsigned int currMbufSize:16

### A.5.53.2 Detailed Description

Rx Descriptor definition.

## A.5.54 IxNpeA_TxAtmVc Struct Reference

Tx Descriptor definition.

```
#include <IxNpeA.h>
```

### A.5.54.1 Data Fields

- UINT8 port
- UINT8 aalType
- UINT16 currMbufLen
- IxNpeA_AtmCellHeader atmCellHeader
- IX_MBUF * pCurrMbuf
- unsigned char * pCurrMbufData
- IX_MBUF * pNextMbuf
- UINT32 totalLen
- UINT32 aal5CrcResidue

### A.5.54.2 Detailed Description

Tx Descriptor definition.

## A.5.55 IxNpeDlVersionId Struct Reference

Version Id to identify each version contained in an image.

```
#include <IxNpeDl.h>
```

### A.5.55.1 Data Fields

- IxNpeDlNpeId npeId
  NPE ID.
- IxNpeDlBuildId buildId
  Build ID indicates functionality of version.
- IxNpeDlMajor major
  Major Release Number.
- IxNpeDlMinor minor
  Minor Revision Number.

### A.5.55.2 Detailed Description

Version Id to identify each version contained in an image.

## A.5.56 IxNpeMhMessage Struct Reference

The 2-word message structure to send to and receive from the NPEs.

```
#include <IxNpeMh.h>
```

### A.5.56.1 Data Fields

- UINT32 data [2]
  the actual data of the message.

### A.5.56.2 Detailed Description

The 2-word message structure to send to and receive from the NPEs.

## A.5.57 ixUARTDev Struct Reference

Device descriptor for the UART.

```
#include <IxUART.h>
```

### A.5.57.1 Detailed Description

Device descriptor for the UART.

## A.5.58 ixUARTStats Struct Reference

Statistics for the UART.

`#include <IxUART.h>`

### A.5.58.1 Detailed Description

Statistics for the UART.

## A.5.59 LearningIpFlow Struct Reference

Struct to hold IP flow information while learning.

`#include <IxFpathAccCodeletMain_p.h>`

### A.5.59.1 Data Fields

- BOOL inUse
- UINT32 ipAddress
  Learning is taking place.
- UINT8 macAddress [6]
  Destination IP address.
- unsigned int counter
  Corresponding Mac Address.

### A.5.59.2 Detailed Description

Struct to hold IP flow information while learning.

## A.5.60 PacketisedStats Struct Reference

Type definition structure for Packetised statistics.

`#include <IxHssAccCodeletCom.h>`

### A.5.60.1 Data Fields

- UINT32 txPackets
- UINT32 txBytes
- UINT32 txNoBuffers
- UINT32 rxPackets
- UINT32 rxBytes
- UINT32 rxNoBuffers
- UINT32 rxIdles
- UINT32 rxVerifyFails

- UINT32 connectFails
- UINT32 portEnableFails
- UINT32 txFails
- UINT32 replenishFails
- UINT32 portDisableFails
- UINT32 disconnectFails
- UINT32 txBufsInUse
- UINT32 rxBufsInUse
- UINT32 stopShutdownErrors
- UINT32 hdlcAlignErrors
- UINT32 hdlcFcsErrors
- UINT32 rxQueueEmptyErrors
- UINT32 hdlcMaxSizeErrors
- UINT32 hdlcAbortErrors
- UINT32 disconnectErrors
- UINT32 unrecognisedErrors

### A.5.60.2 Detailed Description

Type definition structure for Packetised statistics.