Intel® IXP400 Software

Programmer's Guide

June 2003

Document Number: 252539-004a

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL® PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT. Intel products are not intended for use in medical, life saving, life sustaining applications.

Intel may make changes to specifications and product descriptions at any time, without notice.

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

Intel® IXP400 Software v1.3 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.

AlertVIEW, AnyPoint, AppChoice, BoardWatch, BunnyPeople, CablePort, Celeron, Chips, CT Connect, CT Media, Dialogic, DM3, EtherExpress, ETOX, FlashFile, i386, i486, i960, iCOMP, InstantIP, Intel, Intel logo, Intel386, Intel486, Intel740, IntelDX2, IntelDX4, IntelSX2, Intel Create & Share, Intel GigaBlade, Intel InBusiness, Intel Inside, Intel Inside logo, Intel NetBurst, Intel NetMerge, Intel NetStructure, Intel Play, Intel Play logo, Intel SingleDriver, Intel SpeedStep, Intel StrataFlash, Intel TeamStation, Intel Xeon, Intel XScale, IPLink, Itanium, LANDesk, LanRover, MCS, MMX, MMX logo, Optimizer logo, OverDrive, Paragon, PC Dads, PC Parents, PDCharm, Pentium, Pentium II Xeon, Pentium III Xeon, Performance at Your Command, RemoteExpress, Shiva, SmartDie, Solutions960, Sound Mark, StorageExpress, The Computer Inside., The Journey Inside, TokenExpress, Trillium, VoiceBrick, 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 17
1.1 Versions Supported by this Document 17
1.2 Hardware Supported by this Release 17
1.3 Intended Audience 17
1.4 How to Use this Document 18
1.5 About the Processors 18
1.6 Related Documents 19
1.7 Acronyms 21

2 Software Architecture Overview 27
2.1 High-Level Overview 27
2.2 Deliverable Model 28
2.3 Operating System Support 29
2.4 Development Tools 29
2.5 Access Library Source Code Documentation 29
2.6 Release Directory Structure 29
2.7 Threading and Locking Policy 31
2.8 Polled and Interrupt Operation 31
2.9 Statistics and MIBs 32
2.10 Network Data Buffer Management 32

3 Software Development Tools 33
3.1 Intel Hardware 33
3.1.1 Intel® IXDP425 / IXCDP1100 Development Platform 33
3.2 Third-Party Software and Hardware 34
3.2.1 Software Stacks for the Processors 34
3.2.2 Hardware and Software Tools for the Processors 35
3.2.3 Software for the Intel® IXDP425 / IXCDP1100 Development Platform 36
3.2.4 Open Source Software and Tools 38
3.2.5 Intel Communications Alliance 38

4 Access-Layer Components: ATM Driver Access (IxAtmdAcc) API 39
4.1 IxAtmdAcc Component Features 39
4.2 Configuration Services 41
4.2.1 UTOPIA Port-Configuration Service 41
4.2.2 ATM Traffic-Shaping Services 41
4.2.3 VC-Configuration Services 42
4.3 Transmission Services 43
4.3.1 Scheduled Transmission 43
4.3.1.1 Schedule Table Description 45
4.3.2 Transmission Triggers (Tx-Low Notification) 46
4.3.2.1 Transmit-Done Processing 46
4.3.2.2 Transmit Disconnect 48
4.3.3 Receive Services 49
4.3.3.1 Receive Triggers (Rx-Free-Low Notification) 50

4.3.3.2 Receive Processing 50
4.3.3.3 Receive Disconnect 52
4.3.4 Buffer Management 53
4.3.4.1 Buffer Allocation 53
4.3.4.2 Buffer Contents 53
4.3.4.3 Buffer-Size Constraints 55
4.3.4.4 Buffer-Chaining Constraints 55
4.3.5 Error Handling 55
4.3.5.1 API-Usage Errors 55
4.3.5.2 Real-Time Errors 55

5 Access-Layer Components: ATM Manager (IxAtmm) API 57

5.1 IxAtmm Overview 57
5.2 IxAtmm Component Features 57
5.3 UTOPIA Level-2 Port Initialization 58
5.4 ATM-Port Management Service Model 58
5.5 Tx/Rx Control Configuration 61
5.6 Dependencies 63
5.7 Error Handling 63
5.8 Management Interfaces 63
5.9 Memory Requirements 63
5.10 Performance 64

6 Access-Layer Components: ATM Transmit Scheduler (IxAtmSch) API 65

6.1 Overview 65
6.2 IxAtmSch Component Features 65
6.3 Connection Admission Control (CAC) Function 67
6.4 Scheduling and Traffic Shaping 68
6.4.1 Schedule Table 68
6.4.1.1 Minimum Cells Value (minCellsToSchedule) 69
6.4.1.2 Maximum Cells Value (maxCells) 69
6.4.2 Schedule Service Model 69
6.4.3 Timing and Idle Cells 70
6.5 Dependencies 70
6.6 Error Handling 71
6.7 Memory Requirements 71
6.7.1 Code Size 71
6.7.2 Data Memory 71
6.8 Performance 71
6.8.1 Latency 72

7 Access-Layer Components: Security (IxCryptoAcc)API 73

7.1 Overview 73
7.1.1 Authentication Mode 74
7.1.2 Buffer Management 76
7.1.3 Error Handling 76
7.1.4 Endianness 76
7.1.5 Import and Export Regulations of Cryptographic Technology 76

7.1.6 NPE-Based Security Hardware Accelerator and IxCryptoAcc Scope 76
7.1.7 IxCryptoAcc Component Architecture 78
7.2 Functional Interface 79
7.2.1 Overview 79
7.2.2 API 81
7.2.3 Basic API Flow 81
7.2.4 CCD Update Flow 82
7.2.5 Assumptions, Restrictions, and Dependencies 83
7.2.6 Dependencies 84
7.3 Use Models 85
7.3.1 IxCryptoAcc Data Flow Overview 85
7.3.1.1 Description 85
7.3.1.2 Preconditions 85
7.3.1.3 Sequence Details 86
7.3.2 Reference ESP Dataflow 87
7.3.3 Reference AH Dataflow 88
7.4 Memory Requirements 89
7.4.1 Data Memory 89
7.5 Overview of NPE Support for Security 90
7.5.1 NPE Firmware Versioning 90
7.5.2 Supported Encryption and Authentication Algorithms 90
7.5.3 Cipher Modes 91
7.5.4 Electronic Code Book (ECB) 91
7.5.5 Cipher Block Chaining (CBC) 92
7.5.6 Counter Mode (CTR) 92
7.5.7 Detailed Cryptographic NPE Data Flow 92
7.6 Security Hardware Accelerator Demonstration 94

8 Access-Layer Components: DMA Access Driver (IxDmaAcc) API 95

8.1 Overview 95
8.2 Features 95
8.3 Assumptions 95
8.4 Dependencies 96
8.5 DMA Access-Layer API 96
8.5.1 IxDmaAccDescriptorManager 98
8.6 Parameters Description 98
8.6.1 Source Address 98
8.6.2 Destination Address 99
8.6.3 Transfer Mode 99
8.6.4 Transfer Width 99
8.6.5 Addressing Modes 99
8.6.6 Transfer Length 100
8.6.7 Supported Modes 100
8.7 Data Flow 102
8.8 Control Flow 102
8.8.1 DMA Initialization 103
8.8.2 DMA Configuration and Data Transfer 104
8.9 Restrictions of the DMA Transfer 105
8.10 Error Handling 106

8.11 Little Endian 106

9 Access-Layer Components: Ethernet Access (IxEthAcc) API 107

9.1 What's New 107
9.2 IxEthAcc Overview 107
9.3 Ethernet Access Layers: Architectural Overview 108
9.3.1 Role of the Ethernet NPE Firmware 108
9.3.2 Queue Manager 108
9.3.3 Learning/Filtering Database 109
9.3.4 MAC/PHY Configuration 109
9.4 Ethernet Access Layers: Component Features 109
9.5 Data Plane 110
9.5.1 Port Initialization 111
9.5.2 Ethernet Frame Transmission 111
9.5.2.1 Transmission Flow 111
9.5.2.2 Transmit Buffer Management and Priority 112
9.5.2.3 Additional Ethernet Transmission Information 114
9.5.3 Ethernet Frame Reception 114
9.5.3.1 Receive Flow 115
9.5.3.2 Receive Buffer Management 115
9.5.3.3 Additional Receive Path Information 117
9.5.4 Data-Plane Endianness 118
9.5.5 External Memory Requirements 118
9.6 Control Path 119
9.6.1 Ethernet MAC Control 121
9.6.1.1 MAC Duplex Settings 121
9.6.1.2 MII I/O 121
9.6.1.3 Frame Check Sequence 121
9.6.1.4 Frame Padding 121
9.6.1.5 MAC Filtering 121
9.7 Management Information 122

10 Access-Layer Components: Ethernet Database (IxEthDB) API 125

10.1 Overview 125
10.2 What's New 125
10.3 Address Filtering Theory 125
10.4 MAC Address-Learning Database Features 127
10.4.1 EthDB Initialization 130
10.4.2 Promiscuous-Mode Requirement 130
10.4.3 Port Definitions 131
10.4.4 Selective Port Disabling 131
10.4.5 Static Entries 132
10.4.6 Database Maintenance 132
10.4.7 Database Elements 133
10.4.8 Algorithms Used by the Ethernet Learning Tree 133

11 Access-Layer Components: Ethernet PHY (IxEthMii) API 135

11.1 Overview 135

11.2 What's New....135
11.3 Features....135
11.4 Supported PHYs....135
11.5 Dependencies....136

12 Access-Layer Components: Feature Control (IxFeatureCtrl) API....137

12.1 What's New....137
12.2 Overview....137
12.3 Hardware Feature Control....137
12.4 Software Configuration....139
12.5 Dependencies....139

13 Access-Layer Components: Fast-Path Access (IxFpathAcc) API....141

13.1 What's New....141
13.2 Functional Description....142
13.3 Enabling/Modifying/Disabling Fast Path for a VC....143
13.4 Automatic Packet Operations....143
13.5 Application Support and MAC Address Service....143
13.6 Fast-Path Buffers....144
13.7 Fast-Path Dependency Diagram....144
13.8 Error Handling....145
13.9 NPE Fpath Classifier/Modifier Configuration....145
13.9.1 Fast-Path Classifier Template Definition....145
13.9.2 Fast-Path Modifier Template Definition....147
13.9.3 Sample Templates....149
13.10 Little Endian....149

14 Access-Layer Components: HSS-Access (IxHssAcc) API....151

14.1 Overview....151
14.2 Features....151
14.3 HSS and HDLC Coprocessors Operation....152
14.4 Control and Data Flow....153
14.5 Functional Interface Description....154
14.6 Key Assumptions....155
14.7 Programming Procedures....156
14.8 Initialization and Configuration....156
14.9 Buffer Allocation Data-Flow Overview....159
14.9.1 Data Flow in Packetized Service....159
14.9.2 Data Flow in Channelized Service....162
14.10 HSS Channelized Operation....165
14.10.1 Channelized Connect and Enable....165
14.10.2 Channelized Tx/Rx....167
14.10.2.1 CallBack....167
14.10.2.2 Polled....168
14.10.3 Channelized Disconnect....169
14.11 HSS Packetized Operation....170
14.11.1 Packetized Connect and Enable....170

14.11.2 Packetized Tx 172
14.11.3 Packetized Rx 173
14.11.4 Packetized Disconnect 175

15 Access-Layer Components: NPE-Downloader (IxNpeDl) API 177

15.1 What's New 177
15.2 Overview 177
15.3 Microcode Images 177
15.4 Standard Usage Example 178
15.5 Custom Usage Example 179
15.6 IxNpeDl Uninitialization 179
15.7 Note About Deprecated APIs 179

16 Access-Layer Components: NPE Message Handler (IxNpeMh) API 181

16.1 Overview 181
16.2 Initializing the IxNpeMh 182
16.2.1 Interrupt-Driven Operation 182
16.2.2 Polled Operation 182
16.3 Uninitializing IxNpeMh 182
16.4 Sending Messages from an Intel® XScale™ Core Software Client to an NPE 183
16.4.1 Sending an NPE Message 183
16.4.2 Sending an NPE Message with Response 184
16.5 Receiving Unsolicited Messages from an NPE to Client Software 185
16.6 Dependencies 187
16.7 Error Handling 187

17 Access-Layer Components: Performance Profiling (IxPerfProfAcc) API 189

17.1 Overview 189
17.2 Intel® XScale™ Core PMU 189
17.2.1 Counter Buffer Overflow 191
17.3 Internal Bus PMU 191
17.4 Idle-Cycle Counter Utilities (Abbreviated to Xcycle) 192
17.5 Dependencies 192
17.6 Error Handling 193
17.7 Interrupt Handling 193
17.8 Threading 193
17.9 Using the API 193
17.9.1 API Usage for Intel® XScale™ Core PMU 194
17.9.1.1 Event and Clock Counting 194
17.9.1.2 Time-Based Sampling 196
17.9.1.3 Event-Based Sampling 198
17.9.1.4 Using Intel® XScale™ Core PMU to Determine Cache Efficiency 199
17.9.2 Internal Bus PMU 200
17.9.2.1 Using the Internal Bus PMU Utility to Monitor Read/Write Activity on the North Bus 201
17.9.3 Xcycle (Idlecycle Counter) 203

18 Access-Layer Components: Queue Manager (IxQMgr) API 205

18.1 Overview 205
18.2 Features and Hardware Interface 206
18.3 IxQMgr Initialization and Uninitialization 207
18.4 Queue Configuration 207
18.5 Queue Identifiers 208
18.6 Configuration Values 208
18.7 Dispatcher 208
18.8 Threading 211
18.9 Dependencies 211

19 Access-Layer Components: UART-Access (IxUARTAcc) API 213

19.1 Interface Description 213
19.2 UART / OS Dependencies 214
19.2.1 FIFO Versus Polled Mode 214
19.3 Dependencies 215

20 Access-Layer Components: USB Access (ixUSB) API 217

20.1 Overview 217
20.2 USB Controller Background 217
20.2.1 Packet Formats 218
20.2.2 Transaction Formats 219
20.3 ixUSB API Interfaces 222
20.3.1 ixUSB Setup Requests 222
20.3.1.1 Configuration 224
20.3.1.2 Frame Synchronization 225
20.3.2 ixUSB Send and Receive Requests 225
20.3.3 ixUSB Endpoint Stall Feature 225
20.3.4 ixUSB Error Handling 226
20.4 USB Data Flow 227
20.5 USB Dependencies 228

21 Codelets 229

21.1 Overview 229
21.2 ATM Codelet (IxAtmCodelet) 229
21.3 Crypto Access Codelet (IxCryptoAccCodelet) 229
21.4 DMA Access Codelet (IxDmaAccCodelet) 230
21.5 Ethernet Aal5 Codelet (IxEthAal5App) 230
21.6 Ethernet Access Codelet (IxEthAccCodelet) 230
21.7 Fast Path Access Codelet (IxFpathAccCodelet) 231
21.8 HSS Access Codelet (IxHssAccCodelet) 232
21.9 Performance Profiling Codelet (IxPerfProfAccCodelet) 232
21.10 USB RNDIS Codelet (IxUSBRNDIS) 232

22 Operating System Abstraction Layers 233

22.1 Overview 233
22.2 What's New 234

22.3 ixOsServices 234
22.3.1 Mutual Exclusion Services 234
22.3.2 Trace Services 234
22.3.3 Memory Services 235
22.3.4 Timer Services 235
22.3.5 ixOsServices Functions 235
22.4 OSSL 236
22.4.1 Thread Management 236
22.4.2 Semaphores 236
22.4.3 Mutual Exclusion 237
22.4.4 Semaphores Versus Mutexes 237
22.4.5 Timers 237
22.4.6 Memory Management 237
22.4.7 Message Logging 237
22.4.8 OSSL Functions 238
22.5 Software Component Dependencies on Operating System Services 239

23 ADSL Driver for the Intel® IXDP425 / IXCDP1100 Development Platform 241

23.1 What's New 241
23.2 Device Support 241
23.3 ADSL Driver Overview 241
23.3.1 Controlling STMicroelectronics* ADSL Modem Chipset Through CTRL-E 242
23.4 ADSL API 242
23.5 ADSL Line Open/Close Overview 243
23.6 Limitations and Constraints 244

A Application Programming Interfaces 245

A.1 Access-Layer API and Codelet Module Index 245
A.2 IXP425 Operating System Services Library (IxOSSL) API 246
A.3 IXP425 ADSL Driver API 268
A.4 IXP425 Assertion Macros (IxAssert) API 278
A.5 IXP425 ATM Driver Access (IxAtmdAcc) API 278
A.6 IXP425 ATM Driver Access (IxAtmdAcc) Control API 297
A.7 IXP425 ATM Manager (IxAtmm) API 317
A.8 IXP425 ATM Transmit Scheduler (IxAtmSch) API 329
A.9 IXP425 ATM Types (IxAtmTypes) 336
A.10 IXP425 Security (IxCryptoAcc) API 341
A.11 IXP425 DMA Types (IxDmaTypes) 356
A.12 IXP425 DMA Access Driver (IxDmaAcc) API 359
A.13 IXP425 Ethernet Access (IxEthAcc) API 361
A.14 IXP425 Ethernet Access Fast Path (IxEthAccFpathDep) API 391
A.15 IXP425 Ethernet Database (IxEthDB) API 392
A.16 IXP425 Ethernet Database Port Definitions (IxEthDBPortDefs) 401
A.17 IXP425 Ethernet PHY Access (IxEthMii) API 403
A.18 IXP425 Ethernet NPE (IxEthNpe) API 407
A.19 IXP425 Feature Control (IxFeatureCtrl) API 418
A.20 IXP425 Fast Path Access (IxFpathAcc) API 431
A.21 IXP425 HSS Access (IxHssAcc) API 444
A.22 IXP425 NPE-A (IxNpeA) API 465
A.23 IXP425 NPE-Downloader (IxNpeDl) API 481

A.24 IXP425 NPE Message Handler (IxNpeMh) API 501
A.25 IXP425 OS Memory Buffer Management (IxOsBuffMgt) API 509
A.26 IXP425 OS Memory Buffer Pool Management (IxOsBuffPoolMgt) API 510
A.27 IXP425 OS Cache MMU (IxOsCacheMMU) API 521
A.28 IXP425 OS Services (IxOsServices) API 527
A.29 IXP425 Performance Profiling (IxPerfProfAcc) API 539
A.30 IXP425 Queue Manager (IxQMgr) API 567
A.31 IXP425 Timer Control (IxTimerCtrl) API 593
A.32 IXP425 Types (IxTypes) 597
A.33 IXP425 UART Access (IxUARTAcc) API 598
A.34 IXP400 Version ID (IxVersionId) 605
A.35 IXP425 USB Driver Public API 606

B Codelet Reference 625

B.1 Codelet Reference Index 625
B.2 IXP425 ATM Codelet (IxAtmCodelet) API 626
B.3 IXP425 DMA Access Codelet (IxDmaAccCodelet) API 632
B.4 IXP425 Ethenet Aal5 (IxEthAal5App) API 633
B.5 IXP425 Ethernet Access Codelet (IxEthAccCodelet) API 641
B.6 IXP425 Fast Path Access Codelet (IxFpathAccCodelet) API 646
B.7 IXP425 HSS Access Codelet (IxHssAccCodelet) API 654
B.8 IXP425 PerfProf Access Codelet 656
B.9 IXP425 Timers (IxTimersCodelet) API 659
B.10 IXP425 USB RNDIS Codelet (IxUSBRNDIS) API 663
B.11 IXP425 USB RNDIS End Driver Codelet (IxUSBRNDIS) API 666
B.12 IXP425 USB RNDIS Vendor Codelet (IxUSBRNDIS) 668
B.13 IXP425 Crypto Access Codelet 668

C API and Codelet Data Structures 677

C.1 Data Structure Index 677
C.2 Aal5Statistics Struct Reference 682
C.3 ActiveIpFlow Struct Reference 682
C.4 ChannelisedStats Struct Reference 683
C.5 GeneralStats Struct Reference 683
C.6 ix_ossl_thread_main_info_t Struct Reference 684
C.7 ix_ossl_time_t Struct Reference 685
C.8 IxAtmCodeletStats Struct Reference 685
C.9 IxAtmdAccUtopiaConfig Struct Reference 686
C.10 IxAtmdAccUtopiaConfig::UtRxConfig_ Struct Reference 688
C.11 IxAtmdAccUtopiaConfig::UtRxDefineIdle_ Struct Reference 691
C.12 IxAtmdAccUtopiaConfig::UtRxEnableFields_ Struct Reference 692
C.13 IxAtmdAccUtopiaConfig::UtRxStatsConfig_ Struct Reference 697
C.14 IxAtmdAccUtopiaConfig::UtRxTransTable0_ Struct Reference 698
C.15 IxAtmdAccUtopiaConfig::UtRxTransTable1_ Struct Reference 699
C.16 IxAtmdAccUtopiaConfig::UtRxTransTable2_ Struct Reference 699
C.17 IxAtmdAccUtopiaConfig::UtRxTransTable3_ Struct Reference 700
C.18 IxAtmdAccUtopiaConfig::UtRxTransTable4_ Struct Reference 701
C.19 IxAtmdAccUtopiaConfig::UtRxTransTable5_ Struct Reference 702
C.20 IxAtmdAccUtopiaConfig::UtSysConfig_ Struct Reference 702
C.21 IxAtmdAccUtopiaConfig::UtTxConfig_ Struct Reference 704

C.22 IxAtmdAccUtopiaConfig::UtTxDefineIdle_ Struct Reference 707
C.23 IxAtmdAccUtopiaConfig::UtTxEnableFields_ Struct Reference 708
C.24 IxAtmdAccUtopiaConfig::UtTxStatsConfig_ Struct Reference 711
C.25 IxAtmdAccUtopiaConfig::UtTxTransTable0_ Struct Reference 712
C.26 IxAtmdAccUtopiaConfig::UtTxTransTable1_ Struct Reference 713
C.27 IxAtmdAccUtopiaConfig::UtTxTransTable2_ Struct Reference 714
C.28 IxAtmdAccUtopiaConfig::UtTxTransTable3_ Struct Reference 715
C.29 IxAtmdAccUtopiaConfig::UtTxTransTable4_ Struct Reference 715
C.30 IxAtmdAccUtopiaConfig::UtTxTransTable5_ Struct Reference 716
C.31 IxAtmdAccUtopiaStatus Struct Reference 717
C.32 IxAtmdAccUtopiaStatus::UtRxCellConditionStatus_ Struct Reference 718
C.33 IxAtmdAccUtopiaStatus::UtTxCellConditionStatus_ Struct Reference 719
C.34 IxAtmmPortCfg Struct Reference 720
C.35 IxAtmmVc Struct Reference 720
C.36 IxAtmScheduleTable Struct Reference 721
C.37 IxAtmScheduleTableEntry Struct Reference 722
C.38 IxAtmTrafficDescriptor Struct Reference 723
C.39 IxCryptoAccAuthCtx Struct Reference 724
C.40 IxCryptoAccCipherCtx Struct Reference 725
C.41 IxCryptoAccCtx Struct Reference 727
C.42 IxEthAccMacAddr Struct Reference 728
C.43 IxEthDBMacAddr Struct Reference 728
C.44 IxEthDBPortDefinition Struct Reference 729
C.45 IxEthEthObjStats Struct Reference 729
C.46 IxFpCodeletFastPathConnection Struct Reference 730
C.47 IxFpCodeletTemplate Struct Reference 731
C.48 IxHssAccCodeletStats Struct Reference 732
C.49 IxHssAccConfigParams Struct Reference 732
C.50 IxHssAccPktHdlcFraming Struct Reference 733
C.51 IxHssAccPortConfig Struct Reference 734
C.52 IxMbufPool Struct Reference 735
C.53 IxNpeA_AtmVcFp Struct Reference 736
C.54 IxNpeA_NpePacketDescriptor Struct Reference 736
C.55 IxNpeA_RxAtmVc Struct Reference 738
C.56 IxNpeA_TxAtmVc Struct Reference 739
C.57 IxNpeDlImageId Struct Reference 739
C.58 IxNpeMhMessage Struct Reference 740
C.59 IxOamITU610Cell Struct Reference 741
C.60 IxOamITU610GenericPayload Struct Reference 741
C.61 IxOamITU610LbPayload Struct Reference 742
C.62 IxOamITU610Payload Union Reference 742
C.63 IxPerfProfAccBusPmuResults Struct Reference 743
C.64 IxPerfProfAccXcycleResults Struct Reference 743
C.65 IxPerfProfAccXscalePmuEvtCnt Struct Reference 744
C.66 IxPerfProfAccXscalePmuResults Struct Reference 744
C.67 IxPerfProfAccXscalePmuSamplePcProfile Struct Reference 745
C.68 IxQMgrQInlinedReadWriteInfo Struct Reference 746
C.69 ixUARTDev Struct Reference 747
C.70 ixUARTStats Struct Reference 747
C.71 LearningIpFlow Struct Reference 748

C.72 PacketisedStats Struct Reference 748
C.73 USBDevice Struct Reference 750
C.74 USBDeviceCounters Struct Reference 750
C.75 USBSetupPacket Struct Reference 751

Figures

1 Intel® IXP400 Software v.1.3 Architecture Block Diagram 28
2 Buffer Transmission for a Scheduled Port 44
3 IxAtmdAccScheduleTable Structure and Order Of ATM Cell 46
4 Tx Done Recycling — Using a Threshold Level 47
5 Tx Done Recycling — Using a Polling Mechanism 48
6 Tx Disconnect 49
7 Rx Using a Threshold Level 51
8 RX Using a Polling Mechanism 52
9 Rx Disconnect 53
10 Services Provided by Ixatmm 60
11 Configuration of Traffic Control Mechanism 62
12 Component Dependencies of IxAtmm 63
13 Multiple VCs for Each Port, Multiplexed onto Single Line by the ATM Scheduler 68
14 Translation of IxAtmScheduleTable Structure to ATM Tx Cell Ordering 69
15 Generalized IPSec Stack Architecture 74
16 ESP Packet Structure 75
17 Authentication Header 75
18 Security Coprocessor and IPSec Stack Scope 77
19 IxCrypto Component Interface Layering 78
20 Detailed IxCryptoAcc Component Architecture 79
21 Basic IxCryptoAcc API Call Process Flow 82
22 IxCryptoAcc API Call Process Flow for CCD (SA) Updates 83
23 IxCryptoAcc Component Dependencies 84
24 IxCrytoAcc Execution Model 86
25 ESP Data Flow 88
26 AH Data Flow 89
27 NPE IxCryptoAcc Component Interface 92
28 Detailed NPE-to-Intel® XScale™ Core Cryptographic Data Flow 93
29 ixDmaAcc Dependencies 96
30 IxDmaAcc Component Overview 97
31 IxDmaAcc Control Flow 103
32 IxDMAcc Initialization 103
33 DMA Transfer Operation 104
34 Ethernet Access Layers - Block Diagram 110
35 Ethernet Transmit Frame API Overview 111
36 Ethernet Transmit Frame Data Buffer Flow 113
37 Ethernet Receive Frame API Overview 115
38 Ethernet Receive Plane Data Buffer Flow 117
39 MBuf Fields Written by Intel® XScale™ Core (left) and NPE (right) on Ethernet Rx Path 118
40 MBuf Fields Written by Intel® XScale™ Core (left) and NPE (right) on Ethernet Tx Path 119

41 IxEthAcc and Secondary Components 120
42 Example Network Diagram for MAC Learning 126
43 Ethernet Receive Frame API Overview 129
44 Examples of Node Insertion Not Requiring Re-Balancing 132
45 Examples of Node Insertion Requiring Re-Balancing 133
46 Hashing 134
47 Data and Control Flow for Fast Path 142
48 Fast-Path Dependency Diagram 144
49 T1 Tx Signal Format 153
50 HSS/HDLC Access Overview 154
51 HssAccess Initialization 159
52 HSS Packetized Receive Buffering 161
53 HSS Packetized Transmit Buffering 162
54 HSS Channelized Receive Operation 164
55 HSS Channelized Transmit Operation 165
56 hsschannelizedAccess — Connect 167
57 hssChannelizedAccess — Tx/Rx 168
58 hsschannelizedAccess — Disconnect 169
59 hsspacketizedAccess — Connect 171
60 hsspacketizedAccess — Tx 173
61 hsspacketizedAccess — HDLC Rx (Call-Back Implementation) 175
62 hsspacketizedAccess — Disconnect 176
63 Message from Intel® XScale™ Core Software Client to an NPE 184
64 Message with Response from Intel® XScale™ Core Software Client to an NPE 185
65 Receiving Unsolicited Messages from NPE to Software Client 186
66 ixNpeMh Component Dependencies 187
67 IxPerfProfAcc Dependencies 192
68 PerfProf Access Layer Component 194
69 Display Performance Counters 196
70 Display Clock Counter 197
71 Display Xcycle Measurement 204
72 AHB Queue Manager Hardware Block 206
73 AHB Queue Manager Dispatcher Operation 209
74 Example Code for Polled or Interrupt Driven Dispatcher Operation 210
75 UART Services Models 215
76 STALL on IN Transactions 225
77 STALL on OUT Transactions 226
78 USB Dependencies 228
79 Intel® IXP400 Software v.1.3 Operating System Services Layers 233
80 STMicroelectronics* ADSL Chipset on the Intel® IXDP425 / IXCDP1100 Development Platform 242
81 Example of ADSL Line Open Call Sequence 243
82 IxAtmCodelet sub-components 630
83 Payload Format for Routed PDUs 636
84 Payload Format for Bridged Ethernet/802.3 PDUs 637
85 EthAal5 Codelet Data Flow 638
86 Configuration Example 638
87 Configuration Example Number 2 639
88 Bridge Data Flow 651
89 Routed Data Flow 653

Tables

1 Product Line Features 19
2 Intel® IXDP425 / IXCDP1100 Development Platform 33
3 Jungo* Software Technologies Application Stack for the Intel® IXP4XX Product Line of Network Processors and IXC1100 Control Plane Processors 34
4 Intoto* Application Stack for the Processors 34
5 MAJIC-MX* Probe for the Processors 35
6 Raven* and mpDemon* for the Processors 35
7 visionPROBE* and visionICE* for the Processors 36
8 MontaVista* Linux* Distribution and Support for the Intel® IXDP425 / IXCDP1100 Development Platform 36
9 Wind River* Tornado* and VxWorks* Distribution and Support for the Intel® IXDP425 / IXCDP1100 Development Platform 37
10 Red Hat* Boot-Loader v. 1.92 Software for the Intel® IXDP425 / IXCDP1100 Development Platform 37
11 GNUPro* Tools for the Intel® IXDP425 / IXCDP1100 Development Platform 37
12 Linux* Kernel 38
13 GNU Compiler 38
14 Ix_mbuf Fields Required for Transmission 54
15 Ix_mbuf Fields of Available Buffers for Reception 54
16 Ix_mbuf Fields Modified During Reception 54
17 Real-Time Errors 56
18 Supported Traffic Types 66
19 IxAtmSch Data Memory Usage 71
20 SA Tunnel (Cryptographic Context) Memory Requirements 90
21 Supported Encryption Algorithms 91
22 Supported Authentication Algorithms 91
23 DMA Modes Supported for Addressing Mode of Incremental Source Address and Incremental Destination Address 100
24 DMA Modes supported for Addressing Mode of Incremental Source Address and Fixed Destination Address 101
25 DMA Modes supported for Addressing Mode of Fixed Source Address and Incremental Destination Address 102
26 Managed Objects for Ethernet Receive 123
27 Managed Objects for Ethernet Transmit 123
28 PHYs Supported by IxEthMii 136
29 Product ID Values 137
30 Feature Control Register Values 138
31 Classifier Instruction Format in the Classifier Template 146
32 Opcode Operations in TempCls 146
33 Per-VC TempMdf 148
34 Source/Type Template 149
35 NPE Images 178
36 AHB Queue Manager Configuration Attributes 208
37 IN, OUT, and SETUP Token Packet Format 218
38 SOF Token Packet Format 218
39 Data Packet Format 219
40 Handshake Packet Format 219

41 Bulk Transaction Formats 220
42 Isochronous Transaction Formats 220
43 Control Transaction Formats, Set-Up Stage 221
44 Control Transaction Formats 221
45 Interrupt Transaction Formats 221
46 API interfaces Available for Access Layer 222
47 USBSetupPacket 223
48 Host-Device Request Summary 223
49 Detailed Error Codes 227
50 Intel® IXP400 Software v.1.3 ixOsServices Function Descriptions 235
51 Intel® IXP400 Software v.1.3 ixOSSL Function Descriptions 238
52 Access Layer Component OS Service Dependencies 239

Revision History

Date	Revision	Description
June 2004	004a	Updated brand names and document titles.
September 2003	004	Made two minor corrections.
August 2003	003	Updated manual for IXP400 Software Version 1.3.
February 2003	002	Removed “Intel Confidential” classification.
February 2003	001	Initial release of document.

Introduction 1

This chapter contains important information to help you in getting learning about and using the Intel® IXP400 Software v1.3.

1.1 Versions Supported by this Document

This programmer's guide is intended to be used in conjunction with the IXP400 software v1.3. For subsequent versions after software release 1.3, refer to their accompanying release notes for information about the proper documentation sources to be used.

Previous versions of programmers guides for earlier IXP400 software releases can be found on the following Web site:

http://developer.intel.com/design/network/products/npfamily/docs/ixp4xx.htm.

To identify your version of software:

- Open the file `ixp425_xscale_sw/src/include/IxVersionId.h`
- Check the value of IX_VERSION_ID

1.2 Hardware Supported by this Release

The Intel® IXP400 Software v.1.3 release supports the all members of the Intel® IXP42X Product Line of Network Processors and IXC1100 Control Plane Processor.

1.3 Intended Audience

This document describes the software release 1.3 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 the IXP42X product line.

1.4 How to Use this Document

This programmer's guide is generally organized as follows:

Chapters	Description
Chapters 1 through 3	Introduce the Intel® IXP42X Product Line of Network Processors and IXC1100 Control Plane Processor and Intel® IXP400 Software, including an overview of the software architecture and development tools.
Chapters 4 through 20	Provide functional descriptions of the various access layer components.
Chapter 21 and 22	Describe the codelets (example applications) and operation system abstraction layers.
Appendix A, B, and C	Provide source-code reference for the major APIs and codelets.

For the developer interested in a limited number of specific features of the IXP400 software, a recommended reading procedure would be:

1. Read Chapters 1 and 2 to get a general knowledge of the products' software and hardware architecture.
2. Read the chapters on the specific access layer component(s) of interest.

   Note: Many of the access layer components have dependencies on other components — particularly on IxNpeDl, IxNpeMh, and IxQmgr. For that reason, developers also should review those chapters.
3. Review the codelet reference material in Appendix B for the codelets that exercise the feature of interest.
4. Refer to the API reference guide in Appendix A and browse the source code.

1.5 About the Processors

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 wire-speed 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 has 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 hardware platforms 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.

Table 1 shows the features available on various processors in the IXP42X product line.

Table 1. Product Line Features

	Intel® IXP425 Network Processor B0 Step	Intel® IXP422 Network Processor	Intel® IXP421 Network Processor	Intel® IXP420 Network Processor	Intel® IXC1100 Control Plane Processor
Processor Speed (MHz)	266 / 400 / 533	266	266	266	266 / 400 / 533
UTOPIA 2	X (24 devices)		X (4 devices)		
GPIO	X	X	X	X	X
UART 0/1	X	X	X	X	X
HSS 0	X		X		
HSS 1	X		X		
MII 0	X	X	X	X	X
MII 1	X	X		X	X
USB	X	X	X	X	X
PCI	X	X	X	X	X
Expansion Bus	16-bit, 66-MHz	16-bit, 66-MHz	16-bit, 66-MHz	16-bit, 66-MHz	16-bit, 66-MHz
SDRAM	32-bit, 133-MHz	32-bit, 133-MHz	32-bit, 133-MHz	32-bit, 133-MHz	32-bit, 133-MHz
AES / DES / DES3	X	X			
Multi-Channel HDLC	8		8		
SHA-1 / MD-5	X	X			
Commercial Temperature	X	X	X	X	X
Extended Temperature	X				X

1.6 Related Documents

The Intel documents listed below are available from your field representative or from the following Web site:

http://www.intel.com/design/network/products/npfamily/docs/ixp4xx.htm

Document Title	Document #
Intel® IXP400 Software Version 1.2.2 Software Specification Update	273795
Intel® IXP42X Product Line of Network Processors and IXC1100 Control Plane Processor Developer's Manual	252480
Intel® IXP42X Product Line of Network Processors and IXC1100 Control Plane Processor Datasheet	252479
Intel® IXP42X Product Line of Network Processors and IXC1100 Control Plane Processor Specification Update	252702
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	

1.7 Acronyms

AAL	ATM Adaptation Layer
ABR	Available Bit Rate
ACK	Acknowledge Packet
ADSL	Asymmetric Digital Subscriber Line
AES	Advanced Encryption Standard
AH	Authentication Header (RFC 2402)
AHB	Advanced High-Performance Bus
AL	Adaptation Layer
APB	Advanced Peripheral Bus
API	Application Programming Interface
AQM	AHB Queue Manager
ATM	Asynchronous Transfer Mode
ATU-C	ADSL Termination Unit — Central Office
ATU-R	ADSL Termination Unit — Remote
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
DMT	Discrete Multi-Tone
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 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

This page intentionally left blank.

Software Architecture Overview 2

2.1 High-Level Overview

The primary design principles of the Intel® IXP400 Software v.1.3 architecture are to enable the Intel® IXP42X Product Line of Network Processors and IXC1100 Control Plane Processor hardware in a manner which allows maximum flexibility. Intel® IXP400 Software v.1.3 consists of a collection of software components specific to the IXP42X product line and their reference boards.

This section discusses the software architecture of this product, as shown in "Intel® IXP400 Software v.1.3 Architecture Block Diagram" on page 28

The **NPE Firmware** consists of one or more loadable and executable NPE instruction files that implement the NPE functionality behind the software release 1.3 library. The NPEs are RISC processors embedded in the main processor that are surrounded by multiple co-processor components. The co-processors provide specific hardware services (e.g. Ethernet processing and MAC interfaces, cryptographic processing, etc.). The NPE instruction files are incorporated into the software release 1.3 library at build time, and the library includes a NPE downloader component that provides NPE code version selection and downloading services. A variety of NPE firmware files are provided, enabling different combinations of services.

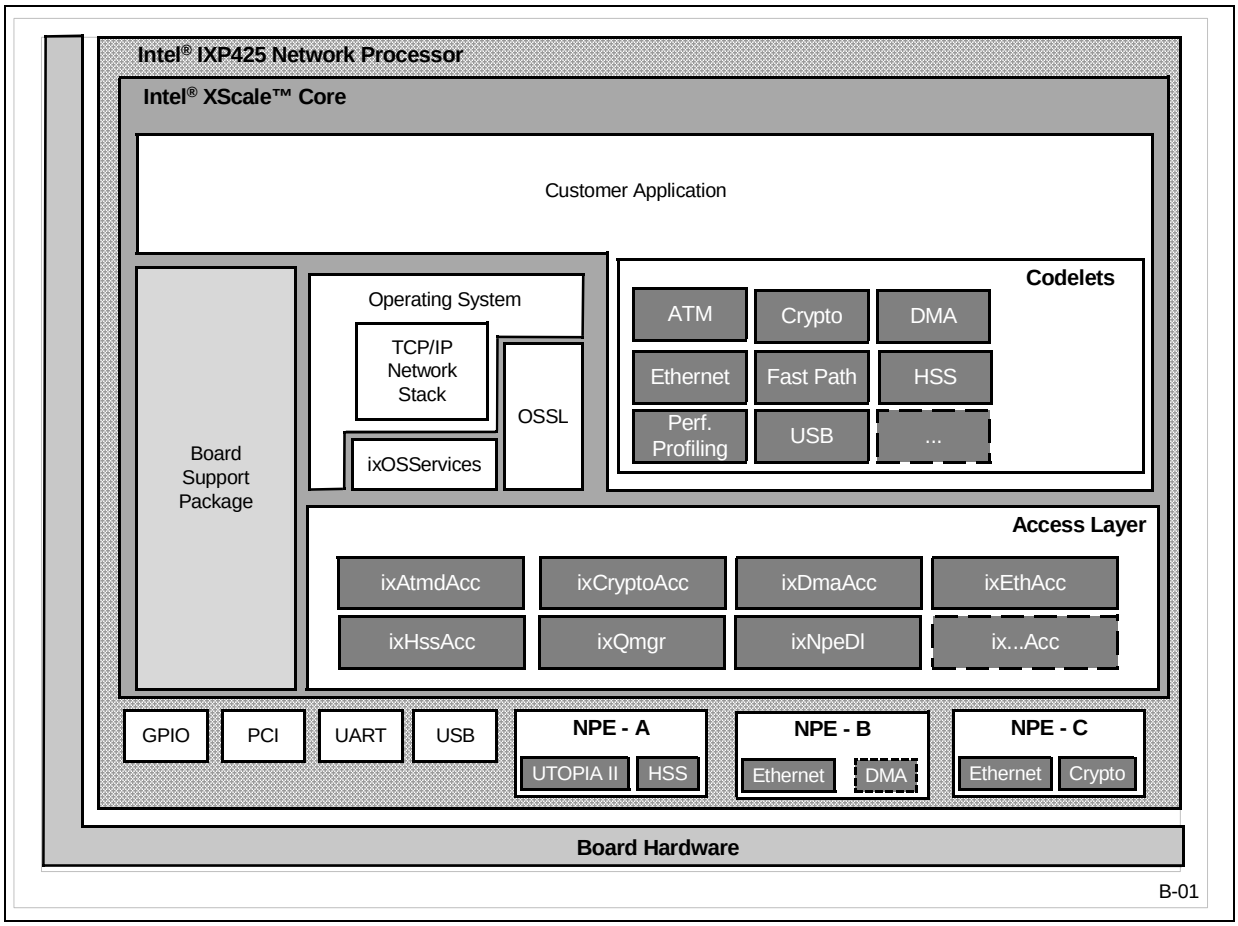
The **Access Layer** provides a software interface which gives customer code access to the underlying capabilities of the IXP42X product line. This layer is made up of a set of software components (Access Layer Components), which clients can use to configure, control and communicate with the hardware. Specifically, most access layer components provide an API interface to specific NPE-hosted hardware capabilities, such as AAL0 and AAL5 on UTOPIA, Cryptography, Ethernet, HSS, or DMA. The remaining access layer components provide an API interface to peripherals on the processors (e.g. UART and USB) or features of the Intel Xscale core (e.g. Product ID Registers or Performance Monitoring Unit)

The example **Codelets** are narrowly focused example applications that show how to use each service or function provided by the Intel XScale core library and the underlying hardware. Many 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 **ixOsServices Layer** provides a very thin set of abstracted OS services. Access Layer components abstract their OS dependency to this module.

Figure 1. Intel® IXP400 Software v.1.3 Architecture Block Diagram



2.2 Deliverable Model

Intel® IXP400 Software v.1.3 consists of these elements:

- Intel® IXP400 Software v.1.3 library
- Complete documentation and source code for library components
- NPE firmware images
- 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 software release 1.3 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 software release 1.3 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 software release 1.3's software 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. A thin abstraction layer is provided for some operating services (timers, mutexs, semiphores, thread management), which can be readily modified to support additional operating systems. This enables the devices to be compatible with multiple operating systems and allows customers the flexibility to port the IXP42X product line 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 board-support packages (BSPs). Tools include Wind River* Tornado* 2.2.1 for the VxWorks 5.5 real-time operating system, and the complete GNU* Linux* development suite.

2.5 Access Library Source Code Documentation

The access library source code uses a commenting style that supports the Doxygen* source code documentation system. Doxygen is an open-source tool, that reads appropriately commented source code and produces hyper-linked documentation of the APIs suitable for on-line browsing (HTML).

The documentation output is typically multiple HTML files, but Doxygen can be configured to produce LaTeX*, RTF (Rich Text Format*), PostScript, hyper-linked PDF, compressed HTML, and Unix* man pages. Doxygen is available for Linux, Windows* and other operating systems.

For more information, see: http://www.doxygen.org.

The Intel® IXP400 Software compressed file contains the HTML source code documentation at `ixp425_xscale_sw\doc\index.html`.

2.6 Release Directory Structure

The software release 1.3 includes the following directory structure:

```
\---ixp425_xscale_sw
    +---buildUtils  (setting environment vars. in VxWorks and Linux
    +---doc  (HTML API Reference)
```

```
\---src (contains access-layer and codelet source code)
    +---adsl
    +---atmdAcc
    +---atmm
    +---atmsch
    +---codelets (sub-directory for codelet source)
    |   +---atm
    |   +---cryptoAcc
    |   +---dmaAcc
    |   +---ethAal5App
    |   +---ethAcc
    |   +---fpathAcc
    |   +---hssAcc
    |   +---perfProfAcc
    |   +---timers
    |   \---usb
    |       +---drivers
    |       \---include
    +---cryptoAcc
    +---dmaAcc
    +---ethAcc
    |   \---include
    +---ethDB
    |   \---include
    +---ethMii
    +---featureCtrl
    +---flashUpgrade (utility for burning flash)
    +---fpathAcc
    +---hssAcc
```

```
|   \---include
+---include  (header location for top-level public modules)
+---linux (OS Services abstractions for Linux)
+---npeDl
|   \---include
+---npeMh
|   \---include
+---osServices
+---ossl
+---perfProfAcc
+---qmgr
+---timerCtrl
|   \---include
+---uartAcc
|   \---include
\---usb
    \---include
```

2.7 Threading and Locking Policy

The software release 1.3 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.

2.8 Polled and Interrupt Operation

It is possible to use access-layer components in a polled and 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.

2.9 Statistics and MIBs

The software release 1.3 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.

2.10 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 software release 1.3, the access-driver software layer follows a simple buffer-management principle: all buffers used between access-layer software and clients above the access layer software must be allocated and freed by the clients. The client passes a buffer to an access-layer 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 callback function to return the buffer, or may put the buffer back on a free queue for the client to poll. The access-layer components utilize similar buffer management techniques when communicating with the NPEs.

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 software release 1.3 uses two buffer formats for all network data:

- mbuf
- raw buffer

These two formats are assumed by software release 1.3'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 software release 1.3 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 (that is, 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 / IXCDP1100 Development Platform

Table 2. Intel® IXDP425 / IXCDP1100 Development Platform

Product	Intel® IXDP425 / IXCDP1100 Development Platform
Description	The IXDP425 / IXCDP1100 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://www.intel.com/design/network/products/npfamily/docs/ixp4xx.htm
Software Release URL	http://www.intel.com/design/network/products/npfamily/ixp425swr1.htm
Support	Intel

3.2 Third-Party Software and Hardware

3.2.1 Software Stacks for the Processors

Table 3. Jungo* Software Technologies Application Stack for the Intel® IXP42X product line

Product	**OpenRG* for the Intel® IXP42X Product Line of Network Processors and IXC1100 Control Plane Processor**
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: • Home networking (HomePNA, wireless LAN - 802.11b, Bluetooth*) • Network security (Stateful Packet Inspection) • Virtual Private Networking (VPN) • Remote management (Web- and SNMP-based) • Remote-update capabilities • Supports industry-leading standards such as CableHome 1.0 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 4. Intoto* Application Stack for the Processors

Product	**IGateway* for the Intel® IXP42X Product Line of Network Processors and IXC1100 Control Plane Processor**
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 Processors

Table 5. MAJIC-MX* Probe for the 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. • 10/100Base-T Ethernet interface • Supports the Intel XScale® Microarchitecture on-chip trace • Supports on-chip hardware breakpoints • Unlimited software breakpoints • 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 6. Raven* and mpDemon* for the Processors

Products	Raven* mpDemon*
Description	**Raven** 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. **mpDemon** 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. • 10BaseT Ethernet interface • IEEE-1284 parallel port supporting Nibble, Compatible, EPP, and ECP modes • RS-232 DCE port supporting baud rate up to 115 Kbaud • Support for programming target Lattice, and Xilinx devices • Target flash programming with our host application
Product Brief URL	• Raven http://www.ocdemon.net/raven_ds.pdf • mpDemon http://www.ocdemon.net/mp_demon.pdf
Manufacturer	Macraigor Systems LLC (http://www.ocdemon.net)
Support	Hardware Support Only Contact Macraigor Systems LLC

Table 7. visionPROBE* and visionICE* for the Processors

Product	visionPROBE* II visionICE* II
Description	**visionPROBE II** The visionPROBE BDM/JTAG debugging cable is at the entry level to the Wind River Systems* family of hardware-assisted debugging solutions for embedded microprocessor systems. • High-speed parallel download • RTOS awareness • On-chip debug target control • Flash memory programming • External triggering • Source-level debugging **visionICE II** 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. • High-speed Ethernet connection • RTOS awareness • On-chip debug target control • Flash memory programming • External triggering • Source-level debugging
Product Brief URL	• visionPROBE II http://www.windriver.com/products/visionprobe2/index.html • visionICE II 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 / IXCDP1100 Development Platform

Table 8. MontaVista* Linux* Distribution and Support for the Intel® IXDP425 / IXCDP1100 Development Platform

Product	MontaVista* Support for the Intel® IXDP425 / IXCDP1100 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. For big-endian application development
Product Brief URL	http://www.mvista.com/pro/downloads/ixp425.html
Manufacturer	MontaVista (http://www.mvista.com)
Support	MontaVista

Table 9. Wind River* Tornado* and VxWorks* Distribution and Support for the Intel® IXDP425 / IXCDP1100 Development Platform

Product	Tornado 2.2.1 Evaluation Kit for the Intel® IXDP425 / IXCDP1100 Development Platform
Description	Tornado 2.2.1 for Intel XScale® Core, VxWorks 5.5 (60-day evaluation copy) BSP Both for Big Endian and Little Endian application development
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 10. Red Hat* Boot-Loader v. 1.92 Software for the Intel® IXDP425 / IXCDP1100 Development Platform

Product	Red Hat Boot-Loader v. 1.92 Software for the Intel® IXDP425 / IXCDP1100 Development Platform
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 11. GNUPro* Tools for the Intel® IXDP425 / IXCDP1100 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. • gcc Highly optimized ANSI-C compiler • g++ ANSI-tracking C++ compiler • gdb Source and assembly-level debugger • gas GNU assembler • ld GNU linker • Insight Graphical user interface for GDB • Source-Navigator Source code comprehension tool • 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 12. 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/ • Intel XScale core-specific Kernel Patches: ftp://source.mvista.com/pub/ds-patches/2.4/

Table 13. GNU Compiler

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

3.2.5 Intel Communications Alliance

The Intel® Communications Alliance is a community of communications and embedded developers and solutions providers who share a common vision on the convergence of computing technologies, and are committed to the development of modular, standards based building blocks, platforms, and solutions based on Intel technologies, processors, products, and services.

The following link provides the detail.

http://developer.intel.com/design/network/ica/

Access-Layer Components: ATM Driver Access (IxAtmdAcc) API 4

This chapter describes the Intel® IXP400 Software v.1.3's "ATM Driver-Access" access-layer component.

The ATM access-driver component is the IxAtmdAcc software component and provides a unified interface to AAL transmit and receive hardware. The software release 1.3 supports AAL-5, AAL-0, and OAM. 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 chapter describes the configuration, control, and transmit/receive flow of ATM PDU data through the IxAtmdAcc component.

The general principle of improving performance by avoiding unnecessary copying of data 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 software release 1.3, 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.

4.1 IxAtmdAcc Component Features

The services offered by the ixAtmdAcc component are:

- Supports the configuration and activation of up to 12 ports on the UTOPIA Level 2 interface.
- Supports 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 consist of single or chained ix_mbufs.)
- Supports 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 consist of single or chained ix_mbufs.)
- Support 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 consist of single or chained ix_mbufs.)
- Supports 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 consist of single or chained ix_mbufs.)
- Supports 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.

- Supports setting or viewing the CLP for AAL-5 CPCS SARed PDUs.
- Supports setting the transmit CLP CUP in all cells of an AAL-0-48 PDU.
- Supports the client setting 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. (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 consist of single or chained ix_mbufs.)
- Supports delivery of AAL-0 PDU containing 48-byte cells (with good HEC) — received on a particular port and VC.
- Supports delivery of AAL-0 PDU containing 52-byte cells — received on a particular port and VC.
- Supports 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.
- Allows the client to determine the port on which the PDU was received, for all client service types.
- Supports viewing the receive CLP of an AAL-0-48 PDU (logical *or* of the CLP value in each cell contained in the PDU).
- Allows 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.
- Supports up to 32 VCC channels for transmit services and up to 32 channels for AAL-0/ AAL-5 receive services. One client per channel is supported.
- Supports one dedicated OAM transmit channel (OAM-VC) per port. This channel supports transmission of OAM cells on any VC.
- Supports one dedicated OAM receive channel (OAM-VC) for all ports. This channel supports reception of OAM cells from any port on any VC.
- Provides 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 — through which the client decides where and when to invoke the functions of the interface.
- Supports fast-path-exception packet processing.
- Supports use in a complete user environment, a complete-interrupt environment, or a mixture of both.

  This is done by providing the control over the Rx and TxDone dispatch functions and 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.

4.2 Configuration Services

IxAtmdAcc supports three configuration services:

- UTOPIA port configuration
- ATM traffic shaping
- VC configuration

4.2.1 UTOPIA Port-Configuration Service

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

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

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 does not itself 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 submits 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.

4.2.3 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 and 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 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 of using a threshold mechanism provided by IxAtmdAcc or polling 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.

4.3 Transmission Services

The IxAtmdAcc 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 0-65,535 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 cells will be segmented into 52-byte cells, HEC added, and they will be 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, therefore, is beyond the scope of this document.

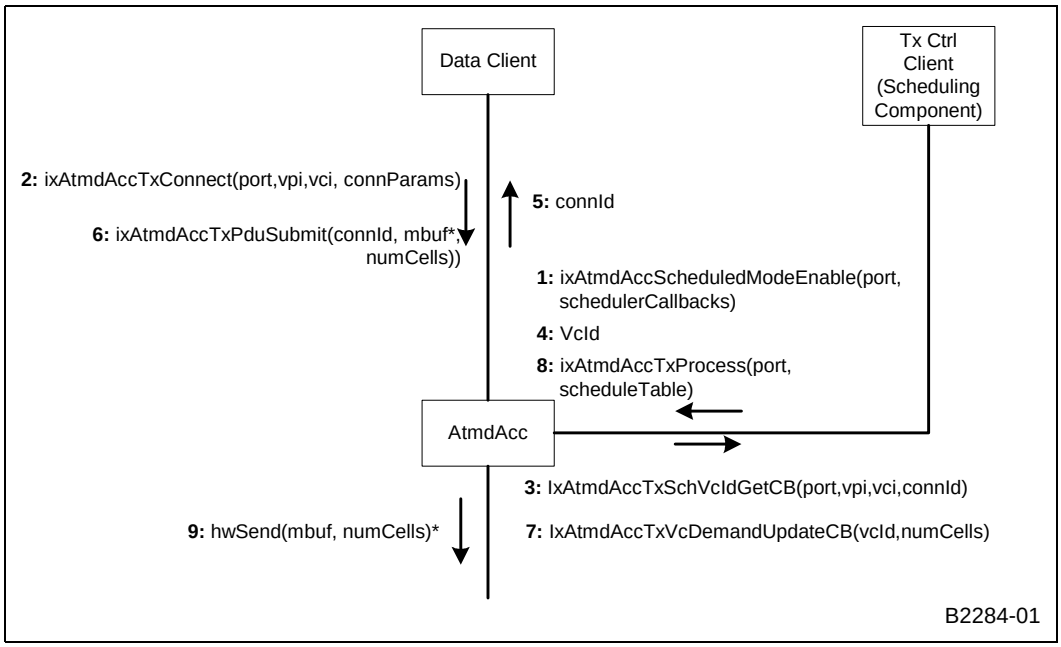
In the case of OAM, a PDU containing OAM cells for any port, VPI, or VCI must be submitted for transmission on the dedicated OAM-VC for that port. This is true regardless of whether an AAL-5/AAL-0-48/AAL-0-52 transmit service connection exists for the given VPI or VCI. The dedicated OAM-VC will be scheduled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

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 2 shows VC connection and buffer transmission for a scheduled port.

Figure 2. 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().

   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 — 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.

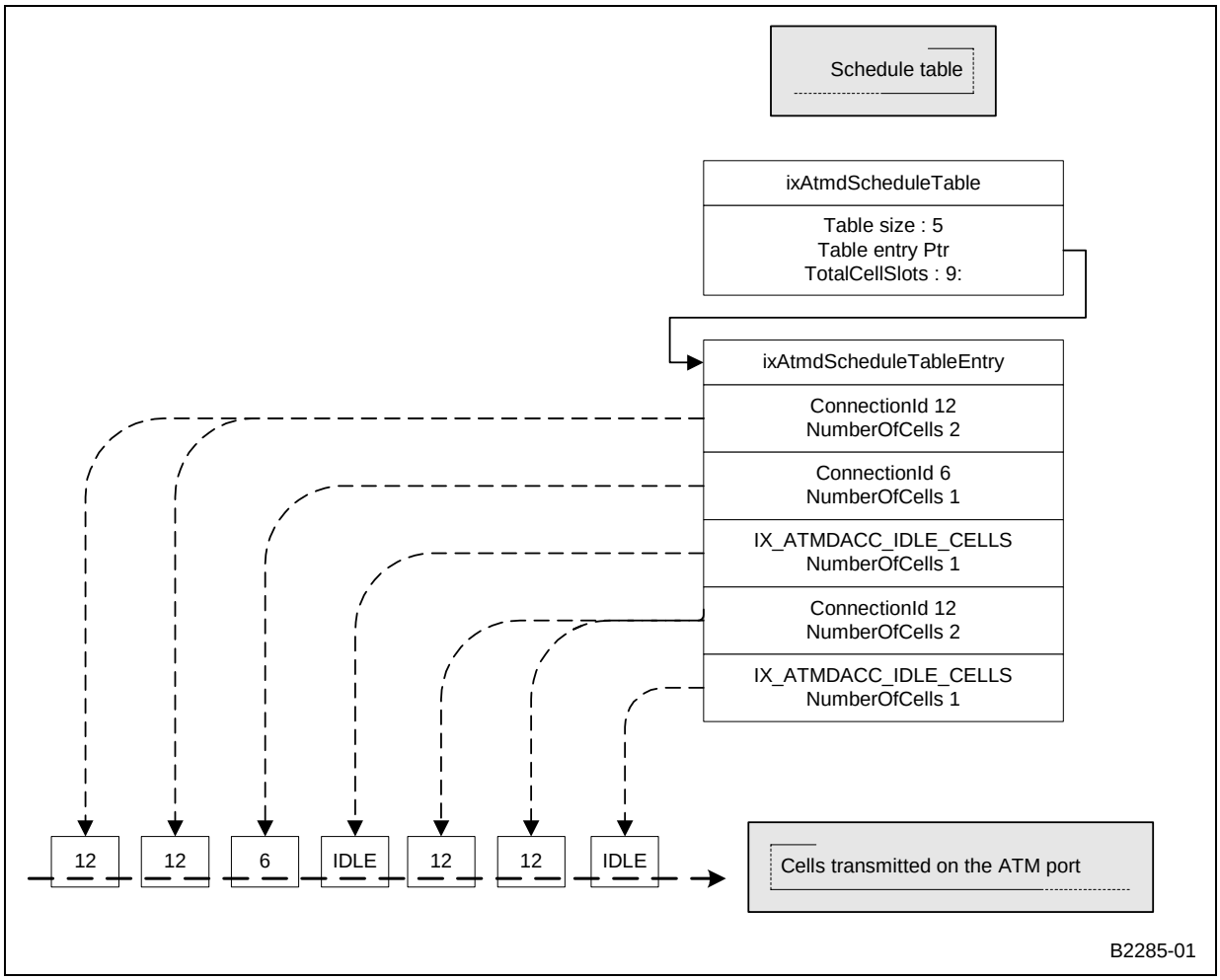
4.3.1.1 Schedule Table Description

IxAtmdAcc 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 3 shows how this table is translated into an ordered sequence of cells transmitted to one ATM port.

Figure 3. IxAtmdAccScheduleTable Structure and Order Of ATM Cell



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 client's discretion and hence 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 eight cells.

4.3.2.1 Transmit-Done Processing

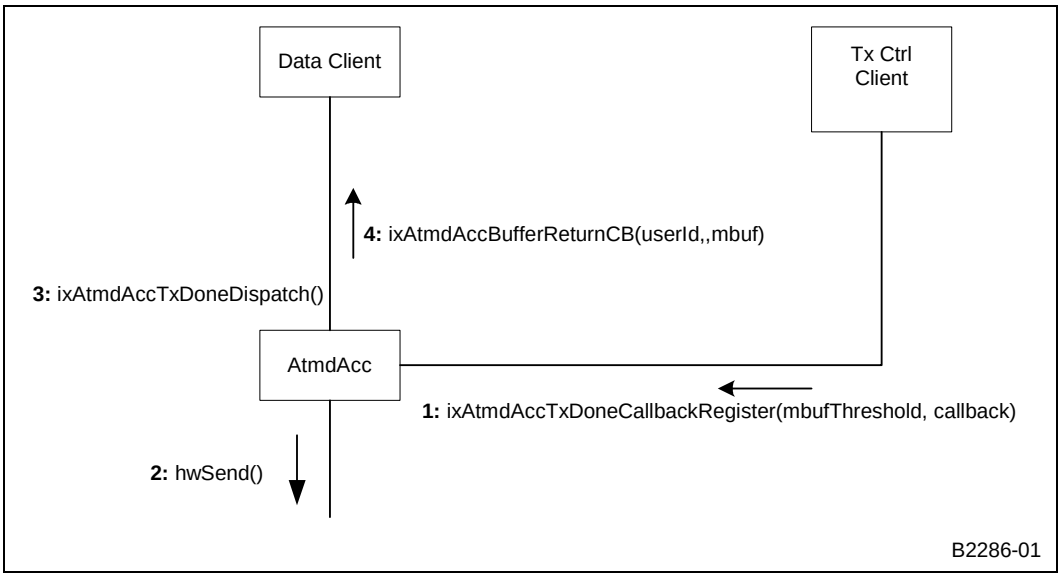
When buffers have been sent on a port, they are placed in a single, transmit-complete stream, which is common to all ports. IxAtmdAcc does not autonomously process this stream — the client, instead, deciding when and how many buffers will be processed.

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 it 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 4.

Figure 4. Tx Done Recycling — Using a Threshold Level



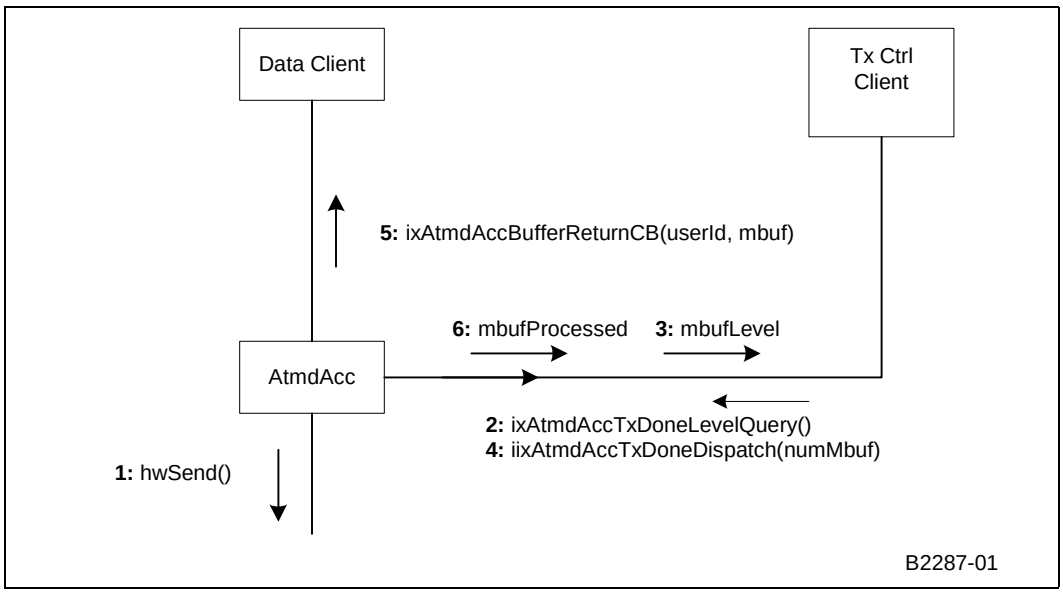
1. The control client wants to use the threshold services to process the transmitted buffers. The ixAtmdAccTxDoneCallbackRegister() function is called to set a buffer threshold level and register a callback. IxAtmdAcc provides the function ixAtmdAccTxDoneDispatch() to be used by the control client. This 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. Sometime 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 IxAtmdAcc. 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 5.

Figure 5. 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 which this buffer belongs. The corresponding data client's TxDoneCallback function — as registered during a TxVcConnect — is invoked with the ix_mbuf.

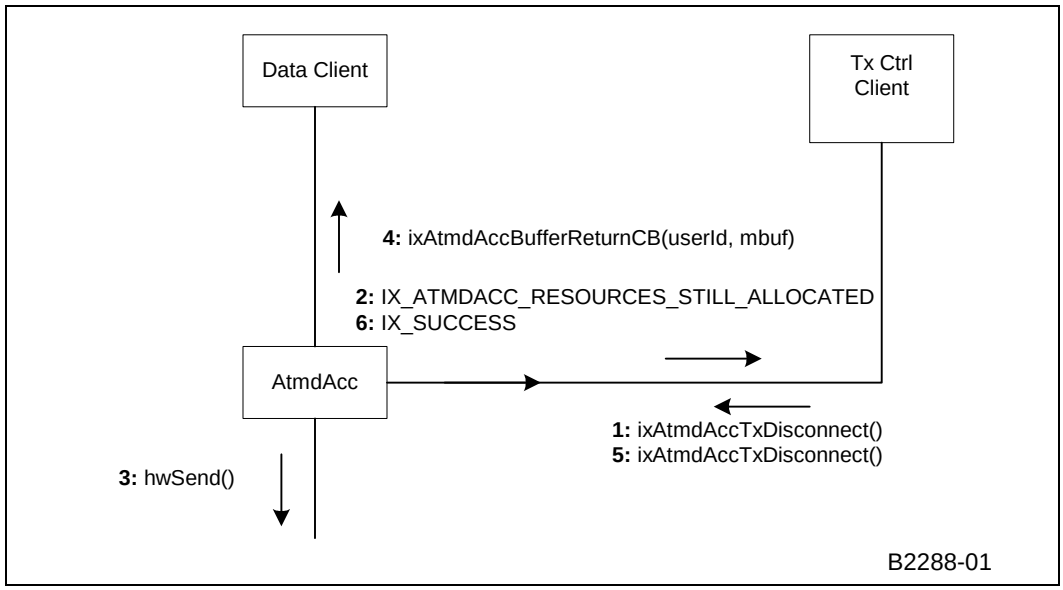
   This TxDoneCallback function is likely to free or recycle the chained ix_mbufs.

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.

4.3.2.2 Transmit Disconnect

Before a client disconnects from a VC, all resources must have been recycled, as shown in Figure 6. 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 6. 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.

4.3.3 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 to 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 (such as 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.

4.3.3.1 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.

4.3.3.2 Receive Processing

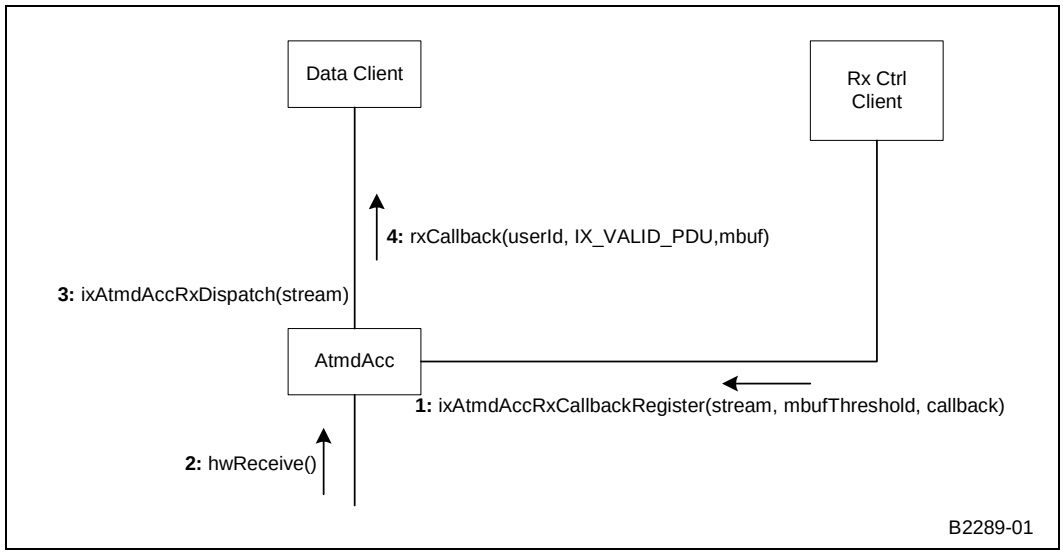
When buffers have been received on a port, they are placed in one of two Rx streams common to the VCs sharing this resource as decided by the client when establishing a connection. IxAtmdAcc does not autonomously process this stream, but instead the client decides when and how many buffers will be processed.

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 7.

Figure 7. 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 iterates through the received buffers and retrieve the connId associated with each buffer.

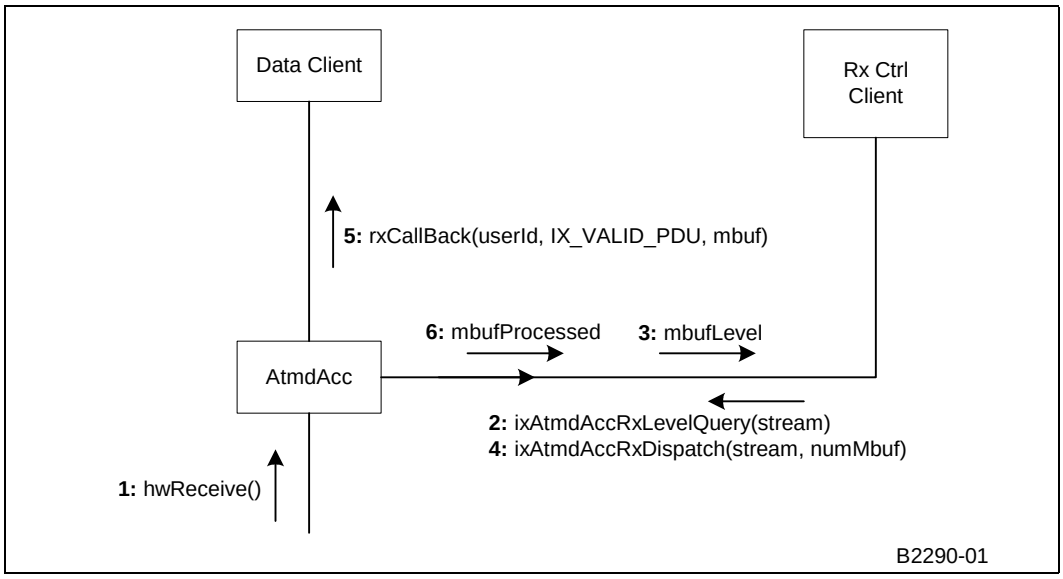
4. Based on connId, ixAtmdAccRxDispatch identified the data client to whom this buffer belongs. The corresponding data client's RxCallback function — as registered during a RxVcConnect — is invoked with the first ix_mbuf of a PDU.

   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 8.

Figure 8. RX Using a Polling Mechanism



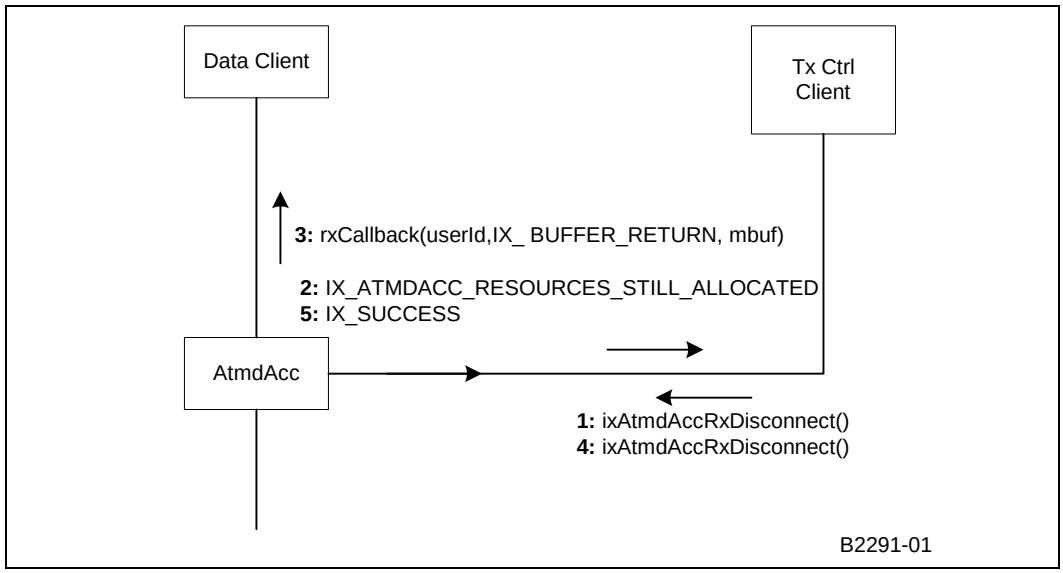
1. Cells are now received over the UTOPIA interface and a complete PDU is now available.
2,3. The control client does not want to use the threshold services. Therefore, the client can optionally query the current number of PDUs already received in one of the receive streams, using the ixAtmdAccRxLevelQuery() function.
4. The control client asks IxAtmdAcc to process an amount of PDUs from one of the streams using the function ixAtmdAccTxDoneDispatch().
5. IxAtmdAcc gets the requested number of PDUs from the underlying hardware. Based on connId, ixAtmdAccRxDispatch() identifies the data clients to which 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.

4.3.3.3 Receive Disconnect

Before a client disconnects from a VC, all resources must have been recycled as shown in Figure 9.

Figure 9. 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.

4.3.4 Buffer Management

The IxAtmdAcc Interface is based on BSD4.4 ix_mbufs.

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 the disconnect steps, pending buffers will be released by the IxAtmDAcc component using the callback functions provided by the client, on a per-VC basis.

4.3.4.2 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 14. These fields will not be changed during the Tx process.

Table 14. 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 15. They are set by the client which wants to provide available buffers to IxAtmdAcc Rx service.

Table 15. 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 16.

Table 16. 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 16. Ix_mbuf Fields Modified During Reception (Sheet 2 of 2)

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

4.3.4.3 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.

4.3.4.4 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.

4.3.5 Error Handling

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, using 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.

4.3.5.2 Real-Time Errors

Errors may occur during real-time traffic. Table 17 shows the different possible errors and the way to resolve them.

Table 17. 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. • AAL-5 CRC errors • PDU length invalid • Cells missing • PDUs missing	• Use the replenish function more often • 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. • Change the TxDone threshold.
IxAtmdAccPduSubmit() reports IX_ATMD_OVERLOADED	System is unable to transmit a PDU.	• Increase the scheduler-transmit speed. • Slow down the submitted traffic.
Rx overflow	• Inbound traffic is dropped. • AAL-5 CRC errors • PDU length invalid	Poll the Rx streams more often.

Access-Layer Components: ATM Manager (IxAtmm) API 5

This chapter describes the Intel® IXP400 Software v.1.3's "ATM Manager API" access-layer component.

IxAtmm is an example IXP400 software component. The phrase "Atmm" stands for "ATM Management." The chapter 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

5.1 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 the IXP42X product line.

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.

5.2 IxAtmm Component Features

The ixAtmm component 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 services include:

- Configuring and tracking the usage of the (physical) ATM ports on IXP42X product line.

  In software release 1.3, 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.

- Initializing 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.

- Ensuring 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.

- Determining the policy for processing transmission buffers recycled from the hardware.

In the IXP400 software, 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.

- Controlling the 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. For example, as soon a buffer is available in the stream, it will be serviced.

  — The low-priority stream will be serviced on a timer basis.

- Allowing clients to register VCCs (Virtual Channel Connections) on all serving ATM ports 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.

- Enabling query for the ATM port and registered VCC information on the port.
- Allowing the client to modify the port rate of any registered port after initialization.

5.3 UTOPIA Level-2 Port Initialization

IxAtmm is responsible for the initial configuration of the IXP42X product line'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 that wish a greater level of control over the UTOPIA device should modify and recompile the IxAtmm component with the new static configuration. Alternately, they can use the interface provided by the IxAtmdAcc component.

5.4 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, from the local ATM port to another device across the ATM network.

Initially, a port is “bare” or “empty.” A VC must be attached (registered) 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, where necessary, to obtain network admission of a VCC. 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 10. Services Provided by Ixatmm

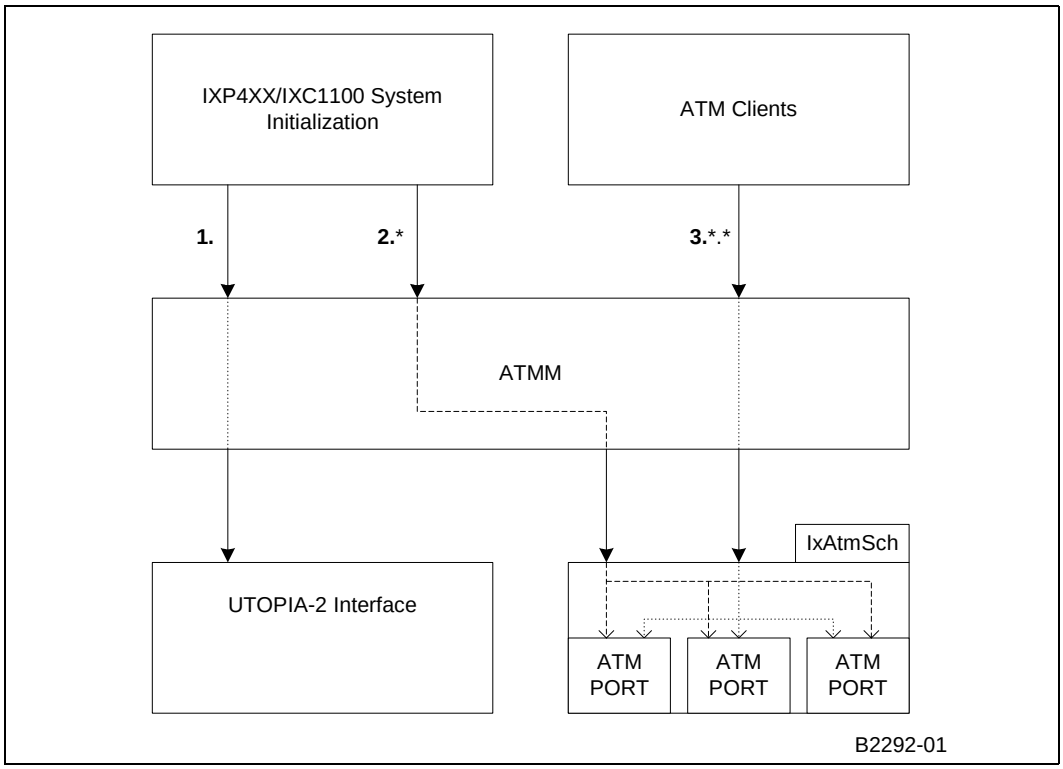


Figure 10 shows the main services provided by the IxAtmm component. In this diagram, the three services outlined are:

- IXP42X product line 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.
- Once the link is established for each active port 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's admission refusal being 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'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.

5.5 Tx/Rx Control Configuration

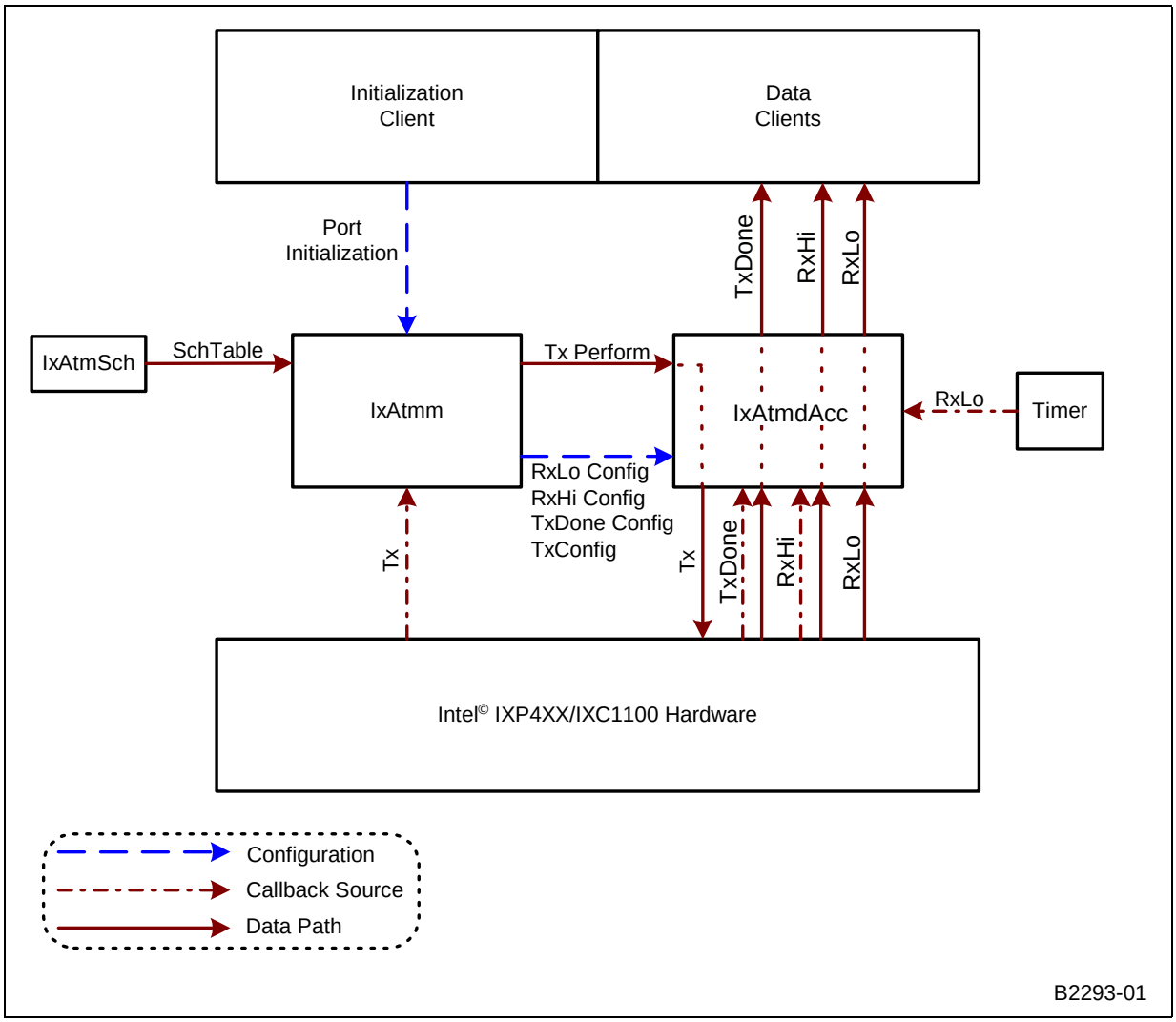
The IxAtmm application is responsible for the configuration of the mechanism by which the lower-layer services will drive transmit and receive of traffic to and from the IXP42X product line' hardware. This configuration is achieved through the IxAtmdAcc component interface. Configuration of these services 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' 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' 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 returns these buffers to the appropriate clients and are then invoked automatically.
- **High-Priority Receive** — Data received on the any high-priority receive channel (such as voice traffic) is required to be supplied to the client in a timely manner. 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 (for example, data traffic) 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. IxAtmm will configure the IXP42X product line processor such that the receive processing of low-priority data will be handled according to 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.

The IxAtmm component 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, all data buffers in the system — once configured — will pass directly through IxAtmdAcc to the appropriate clients. No data traffic will pass through the IxAtmm component at any stage.

Figure 11. Configuration of Traffic Control Mechanism

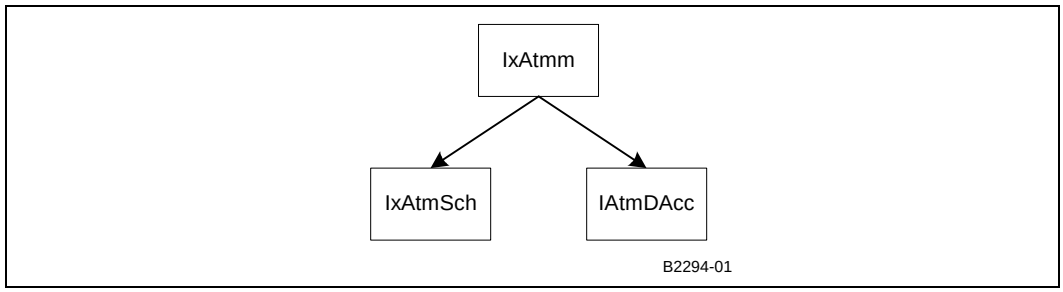


Only transmit traffic — which has already been queued by the client with IxAtmdAcc when the request for more traffic is made — will be scheduled and sent to the hardware. (That is, no callback to the data client will be made in the context of the transmit processing.) IxAtmdAcc makes IxAtmSch aware of 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.

5.6 Dependencies

Figure 12. Component Dependencies of IxAtmm



IxAtmm configures the IXP42X product line'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.

5.7 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 the IXP42X product line's standard error-reporting techniques.

The established state of the IxAtmm component (registered ports, VCs, etc.) is not affected by the occurrence of any error.

5.8 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.

5.9 Memory Requirements

IxAtmm code is approximately 26 Kbytes in size.

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

5.10 Performance

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

Access-Layer Components: ATM Transmit Scheduler (IxAtmSch) API 6

This chapter describes the Intel® IXP400 Software v.1.3's "ATM Transmit Scheduler" (IxAtmSch) access-layer component.

6.1 Overview

IxAtmSch is an "example" software release 1.3 component, an ATM scheduler component supporting ATM transmit services on IXP42X product line processors.

The chapter describes these details of the IxAtmSch component:

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

IxAtmSch is a simplified scheduler with limited capabilities. See Table 18 on page 66 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's ATM transmission-control schedulers.

6.2 IxAtmSch Component Features

The IxAtmSch component is provided as a demonstration ATM scheduler for use in the IXP42X product line's 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's ATM software stack. It is possible (within the complete IXP400 software architecture) to replace this scheduler with one of a different design. If replaced, this component still is valuable as a model of the interfaces that the replacement scheduler 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's chip and software stack, IxAtmSch models them. Because of this, there is no limit to how many ATM ports it can model and schedule — given enough run-time computational resources.

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

In order to use IxAtmSch services, a client first must ask IxAtmSch to establish the model for an ATM port. Virtual connections then can 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 IxAtmSch component's two basic services 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. The IxAtmSch component admits a VC based only on the port capacity, current-port usage, and required-traffic parameters.

The current IXP42X product line's 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 18 shows the ATM service categories that are supported in the current scheduler model.

Table 18. 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.

6.3 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 that 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

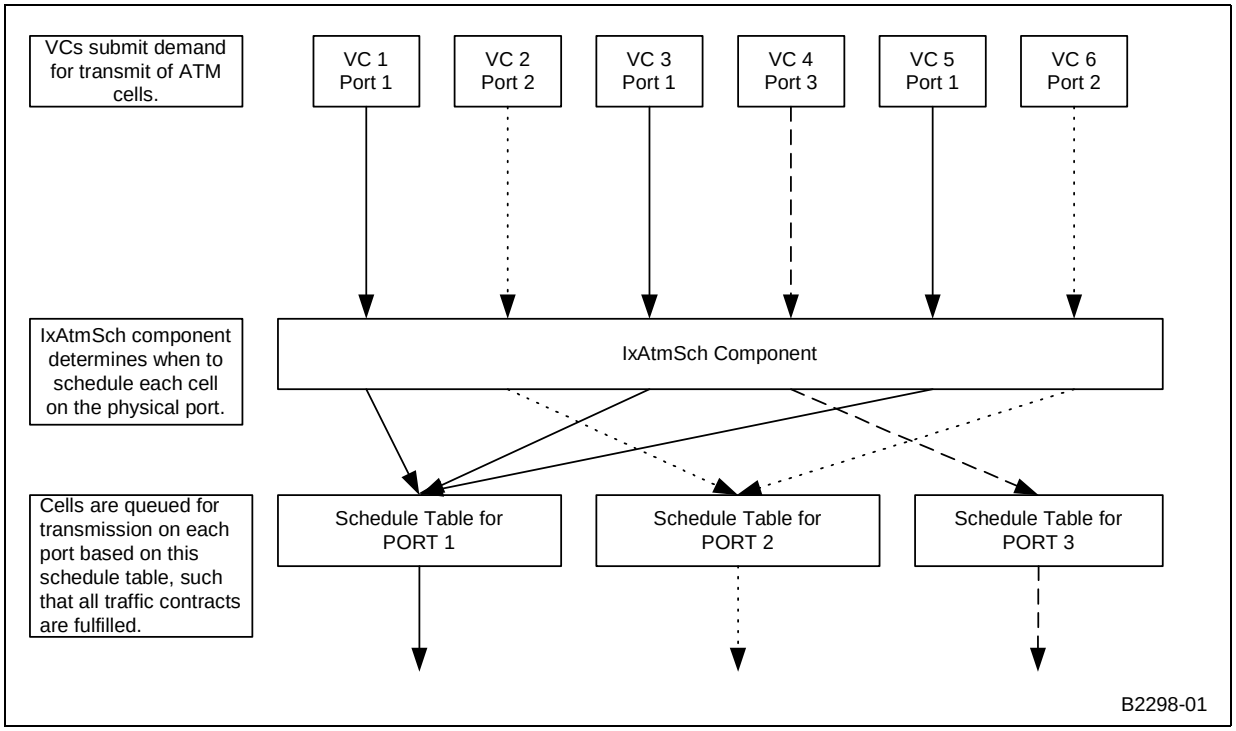
IxAtmSch works on a first-come-first-served basis. For example, if three existing CBR VCs on the ATM port each use one-fourth of the port’s capacity (PCR = R/4), the fourth CBR VCC asking for 1/3 of the port capacity (PCR = R/3) will be rejected. IxAtmSch issues a globally unique VCC ID for each accepted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

For non-CBR real time VCs — 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 IxAtmSch during any subsequent calls. The combination of port and VC ID values will uniquely identify any VC in the IXP42X product line device to the IxAtmSch component.

6.4 Scheduling and Traffic Shaping

Figure 13. Multiple VCs for Each Port, Multiplexed onto Single Line by the ATM Scheduler



6.4.1 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 — will 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.

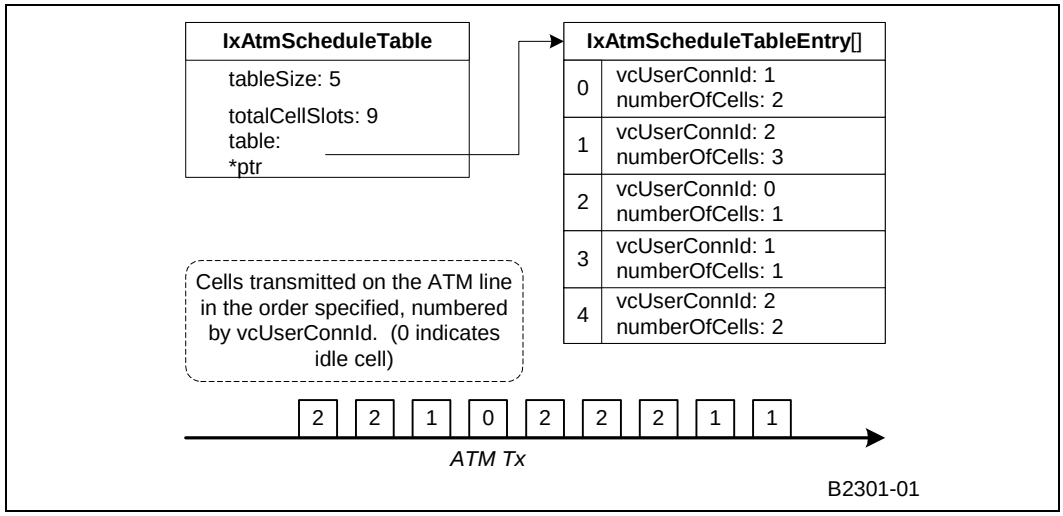
When the client calls the schedule function, the scheduler assumes that all previously scheduled cells on this port have been transmitted and that it may overwrite the previous schedule table with the new table. The client, therefore, must not be dependent on the integrity of the previous table when a request is made for a new schedule table. Additionally, the client should ensure that 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, 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 14 shows how this table is translated into an ordered sequence of cells transmitted to the ATM port.

Figure 14. Translation of IxAtmScheduleTable Structure to ATM Tx Cell Ordering



6.4.1.1 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, 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.

6.4.1.2 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.

6.4.2 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.

6.4.3 Timing and Idle Cells

IxAtmSch does not rely on a hardware clock for timing. Instead, the component 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. (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.

6.5 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.

6.6 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 error-reporting techniques.

6.7 Memory Requirements

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

6.7.1 Code Size

The ixAtmSch code size is approximately 35 Kbytes.

6.7.2 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 19.

Table 19. 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

6.8 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 IXP42X product line processors.

6.8.1 Latency

The transmit latency introduced by the IxAtmSch component into the overall transmit path of 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.

Access-Layer Components: Security (IxCryptoAcc)API 7

This chapter describes the Intel® IXP400 Software v.1.3's "Security API" access-layer component.

7.1 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. Alternately, the IPSec clients 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 Ethernet NPE B of the IXP42X product line processor.

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) hence total key size is 192
  - — 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*)
- Authentication algorithms:
  - — HMAC-SHA1 (512-bit data block size, from 20-byte to 64-byte key size)
  - — HMAC-MD5 (512-bit data block size, from 16-byte to 64-byte 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 15. Generalized IPSec Stack Architecture

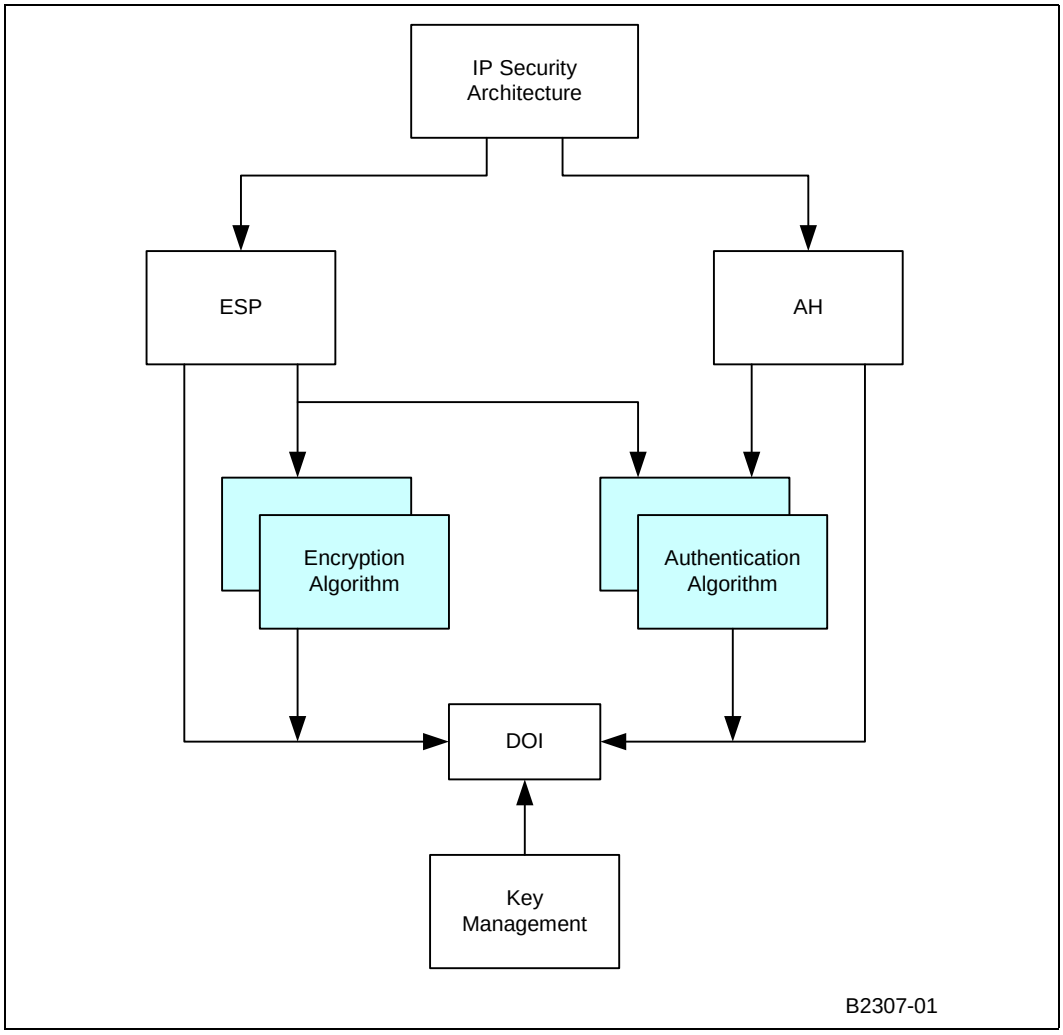


Figure 15 shows the relationship of encryption and authentication algorithm within the IPSec protocol.

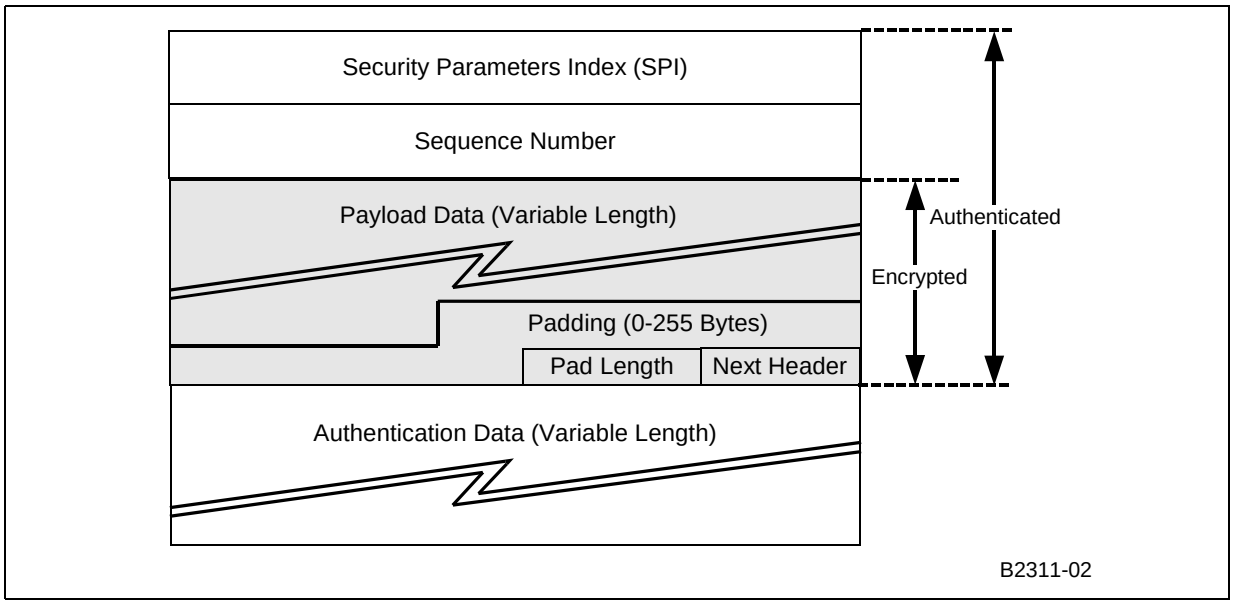
7.1.1 Authentication Mode

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.

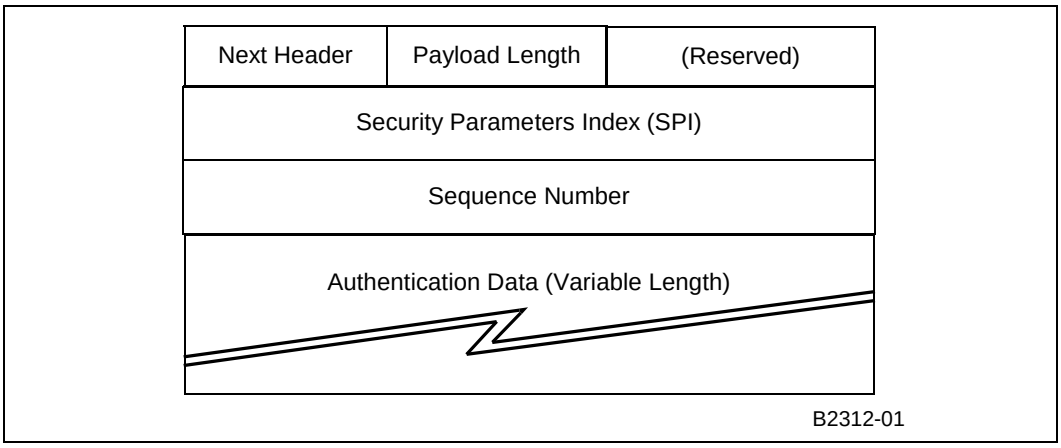
Figure 16. ESP Packet Structure



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 17. Authentication Header



7.1.2 Buffer 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.

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.

7.1.3 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.

7.1.4 Endianness

All the data structures passed into the component are defined in host's byte order. The mode supported by this component is both big endian and little endian.

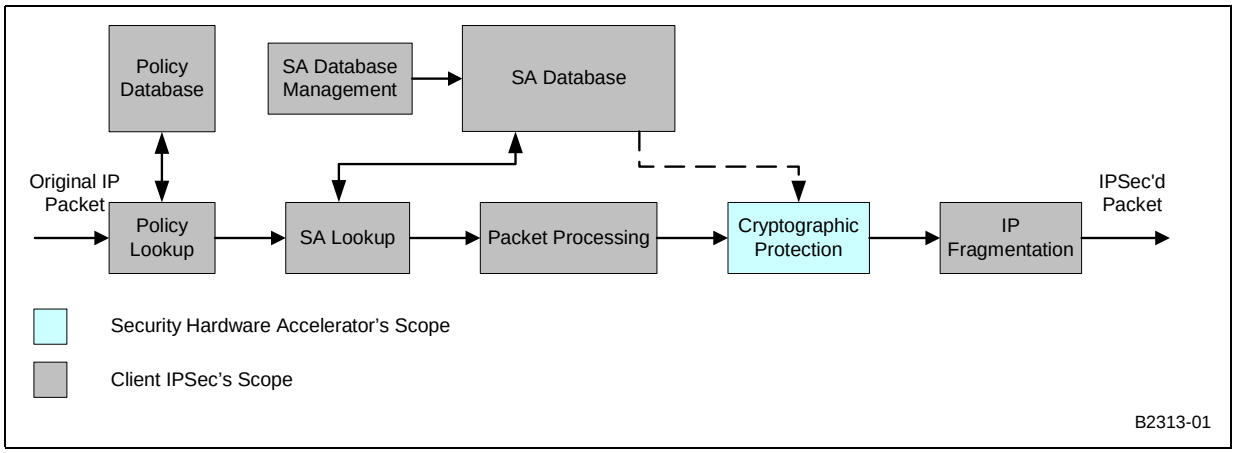
7.1.5 Import and Export Regulations of Cryptographic Technology

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.

7.1.6 NPE-Based Security Hardware Accelerator and IxCryptoAcc Scope

When deploying security-related applications, the generalized architecture in Figure 18 is used. The figure shows the scope and the roles played by the NPE and the IxCryptoAcc component in an IPSec application.

Figure 18. 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.

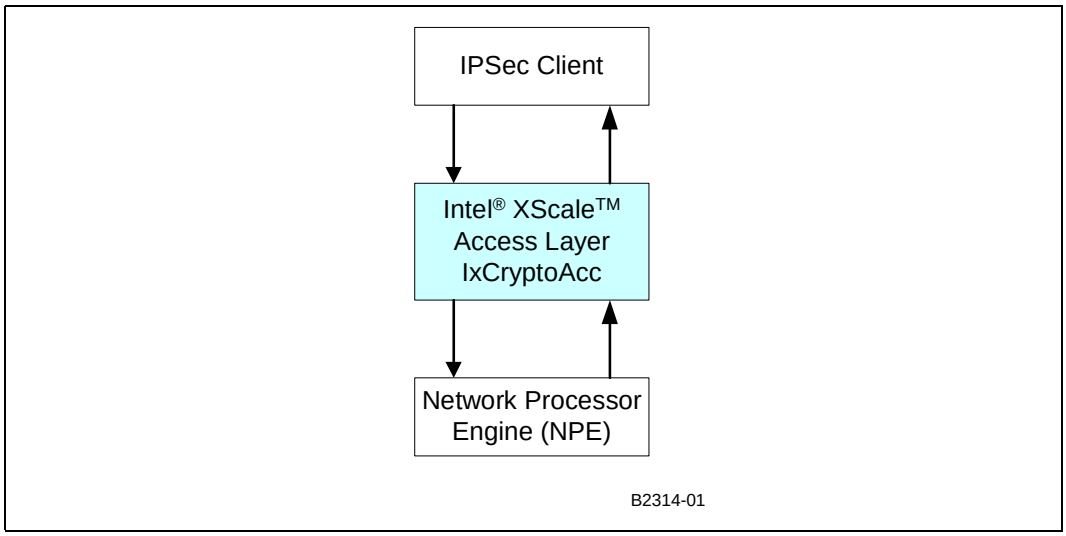
The NPE does not have access to the Security Association (SA) database. The information in the SA database — that is required for the cryptographic protection — will be passed in, via the component API, to the NPE (Cryptographic Protection Block).

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 19. 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.

7.1.7 IxCryptoAcc Component Architecture

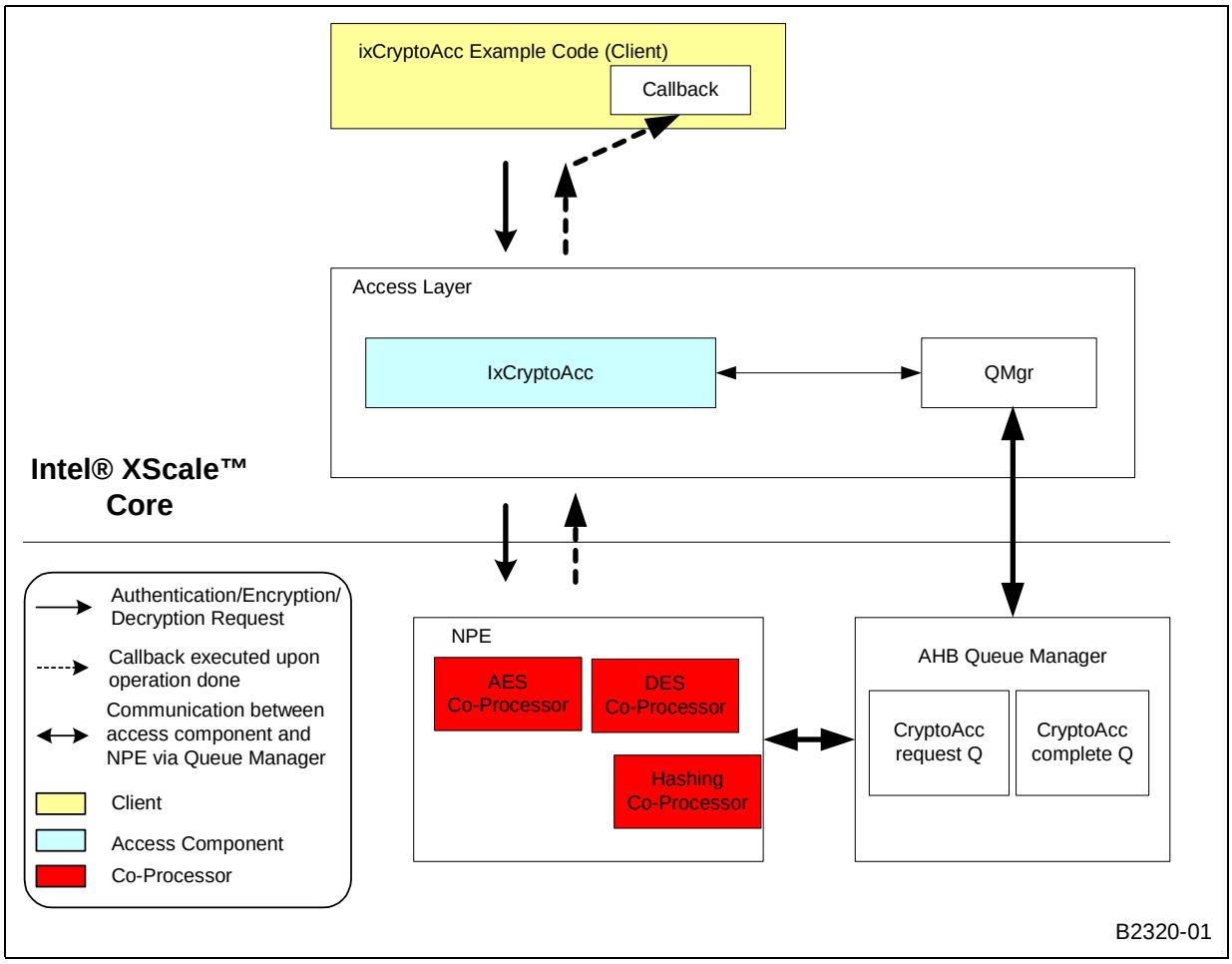
Packets for encryption/decryption and authentication are prepared by the client and passed to the IxCryptoAcc component via the component API. The client component will call IxCryptoAcc for cryptographic services. IxCryptoAcc invokes the NPE to gather the data from SDRAM, with appropriate key information,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

The IxCryptoAcc depends on Qmgr component to configure and use the hardware queues to access the NPE. When the Queue Manager 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.

The resulting data is stored back into the SDRAM.

The architecture is shown in Figure 20.

Figure 20. Detailed IxCryptoAcc Component Architecture



7.2 Functional Interface

7.2.1 Overview

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

- Initialization
- Registration (creation) of a security context
- Unregistration (freeing) of a security context
- Stopping the crypto services
- Displaying the IxCryptoAcc access component's statistics

The complete services include:

- Initialization — Initializes the IxCryptoAcc access component.
- Hashing the authentication key — This will hash the authentication key into L key length (L= 20 bytes for SHA1, L = 16 bytes for MD5), if key length greater than 64 bytes (authentication data block size).The hashing function must be called if the key length greater than 64 bytes, otherwise the key length will be rejected in crypto context allocation.
- 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.

  Note: Authentication keys with a length less than 64 bytes but greater than L (L= 20 bytes for SHA1, L = 16 bytes for MD5) is used directly in crypto context allocation (registration). No further hashing is needed
- 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

7.2.2 API

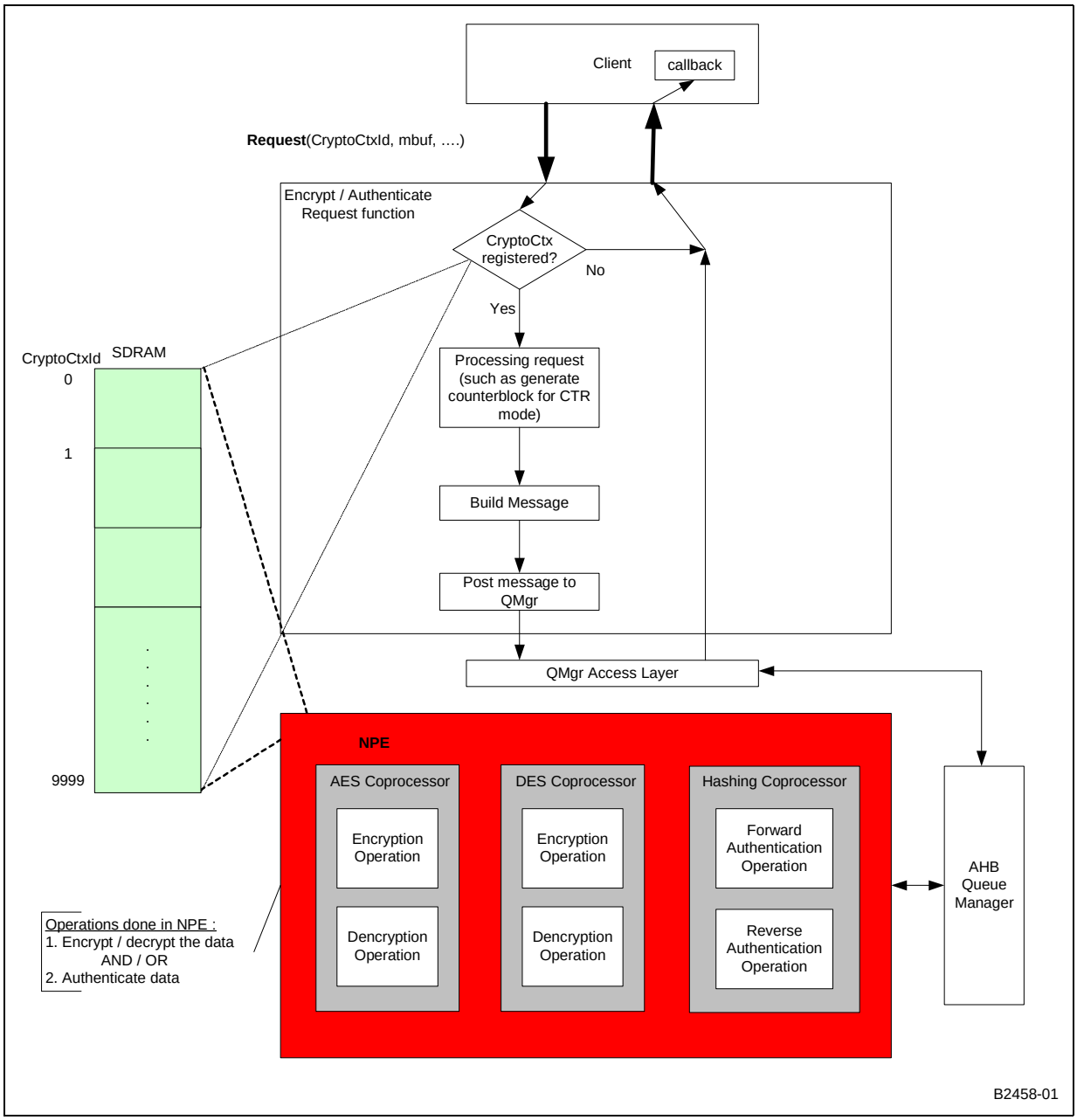
Details of the full API are provided in Appendix A, “Application Programming Interfaces.”

7.2.3 Basic API Flow

The flow of the calls through the API are detailed in Figure 21.

- Client initiates a cryptographic request and checks for a valid cryptographic descriptor.
- The Intel® XScale™ core enqueues a pointer to the cryptographic descriptor onto the CryptoReq Queue in the Q-manager.
- The NPE monitors the empty flag for CryptoReq and reads the cryptographic descriptor pointer.
- The NPE reads the cryptographic structure from SDRAM.
- The NPE reads data from the source mbuf and performs encryption.
- The NPE writes the encrypted data to destination mbuf.
- The Intel XScale core is alerted that the task is completed.

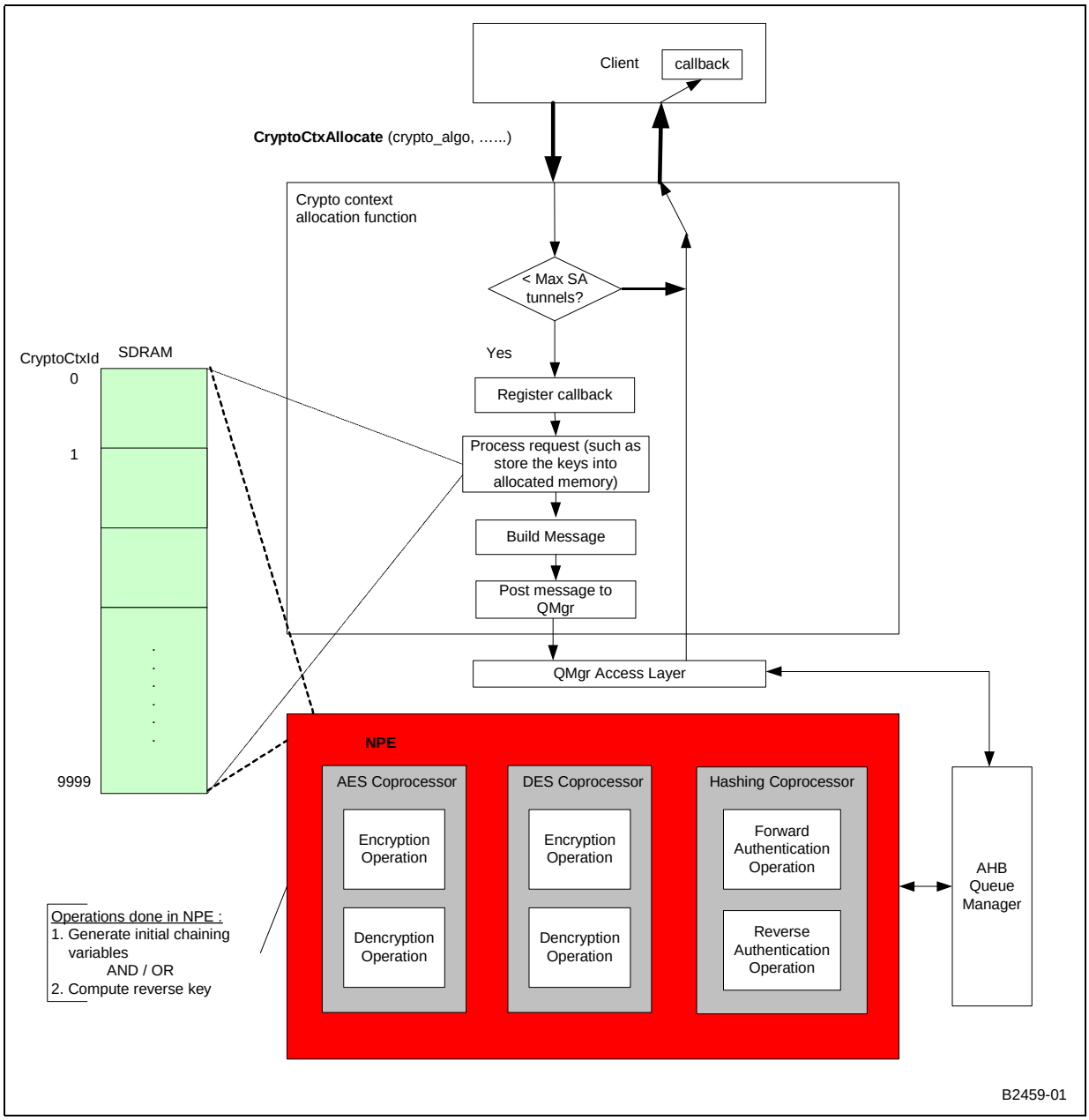
Figure 21. Basic IxCryptoAcc API Call Process Flow



7.2.4 CCD Update Flow

Figure 22 shows the flow through the IxCryptoAcc API when adding to the CCD.

Figure 22. IxCryptoAcc API Call Process Flow for CCD (SA) Updates



7.2.5 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.

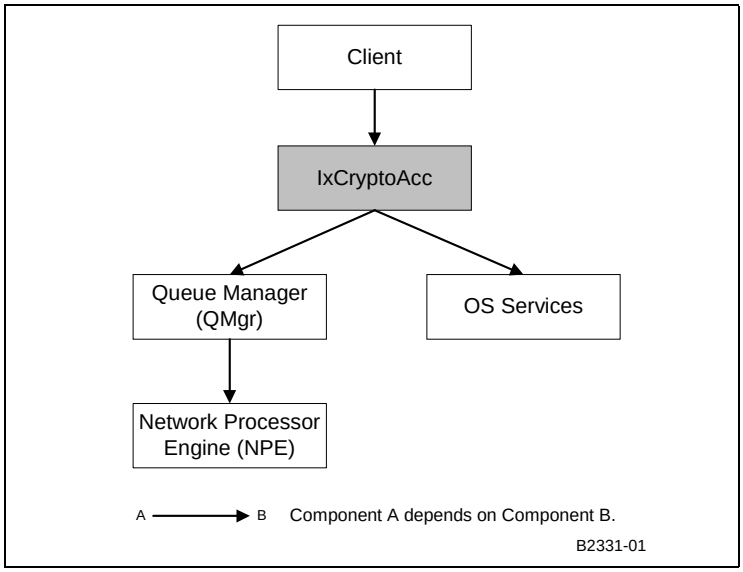
  If a field in the IP header is modified during transit, the value of the field is set to 0 for purposes of the ICV computation. These fields are considered as mutable fields (RFC 2402).
- Client pads the IP datagram to be a multiple of the cipher block size, using ESP trailer for encryption (RFC 2406, 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.

7.2.6 Dependencies

Figure 23 shows the functional dependencies of IxCryptoAcc component.

Figure 23. 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.

7.3 Use Models

7.3.1 IxCryptoAcc Data Flow Overview

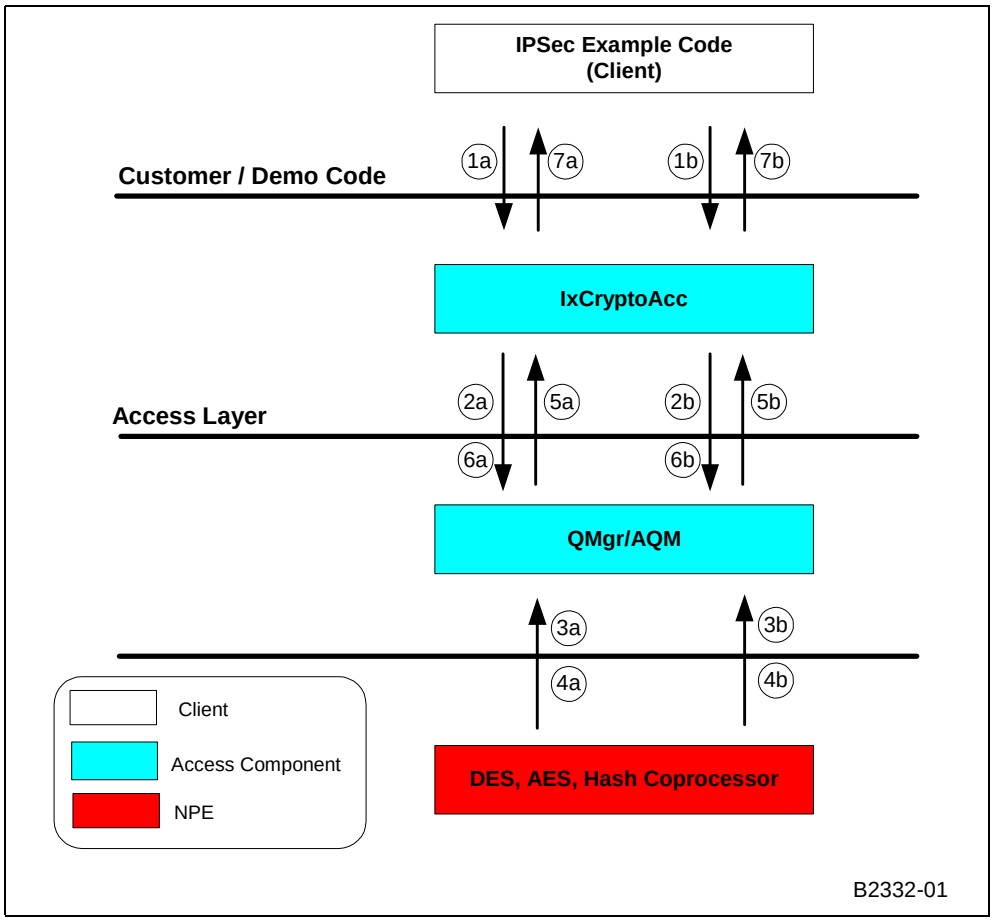
7.3.1.1 Description

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.

7.3.1.2 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 24. IxCrytoAcc Execution Model



7.3.1.3 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.

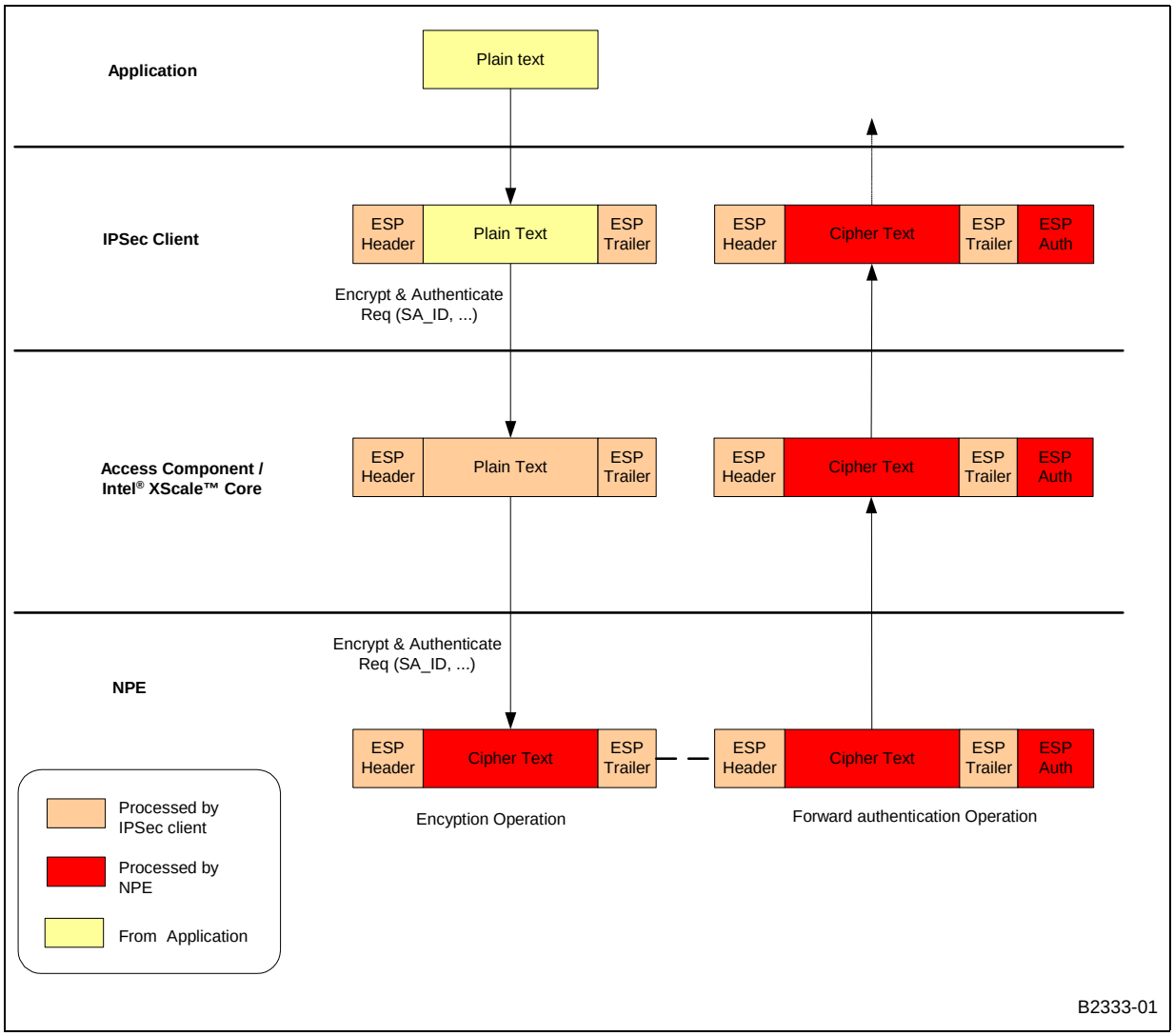
7b. IxCryptoAcc returns status to client via callback function.

7.3.2 Reference ESP Dataflow

Figure 25 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.

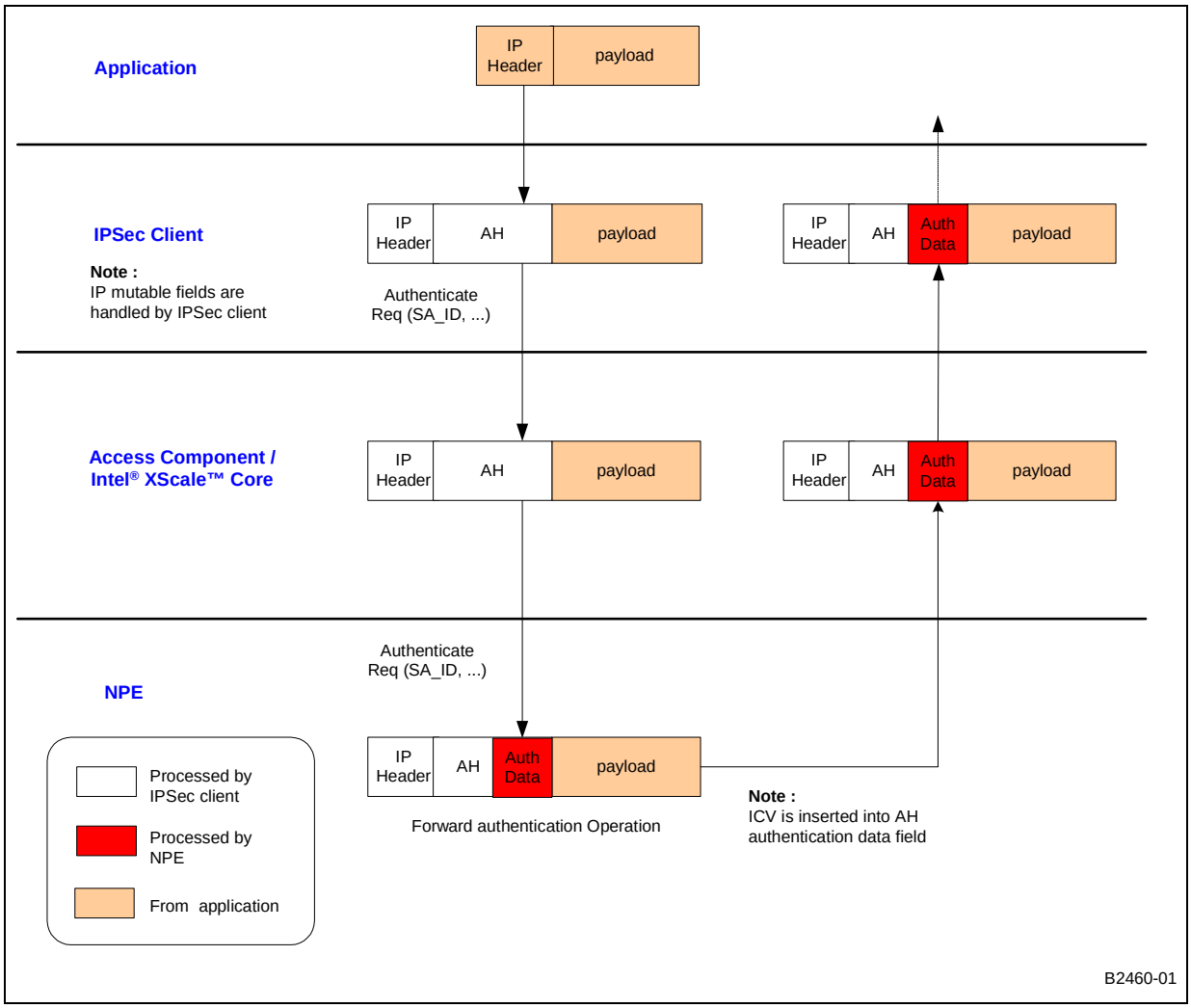
Figure 25. ESP Data Flow



7.3.3 Reference AH Dataflow

Figure 26 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 26. AH Data Flow



7.4 Memory Requirements

7.4.1 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 20 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 144-byte SA tunnels. Hence, 10,000 x 144 Bytes = 1,440 Kbytes of memory required for the CCD.

The breakdown of the size of each tunnel, in bytes, is shown in Table 20.

Table 20. 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	Subtotal (Byte)
Intel XScale® Core Access Layer	0	0	0	8	8	4	8	4	4	8	4	0	48
NPE	8	40	32	0	0	0	0	0	0	0	0	16	96
Total	8	40	32	8	8	4	8	4	4	8	4	16	**144**

7.5 Overview of NPE Support for Security

7.5.1 NPE Firmware Versioning

The security support is fully enabled in the IXP42X product line and is supported only in NPE C. There are variations in the firmware support.

"Standard Usage Example" on page 178 for more information on NPE version numbers.

7.5.2 Supported Encryption and Authentication Algorithms

The NPE supports three different ciphering algorithms

- Data Encryption Standard (DES)
- Triple DES
- Advanced Encryption Standard (AES)

Table 21 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 21. 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	128 192 256	NA	128 192 256	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 22 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 22. Supported Authentication Algorithms

Authentication Algorithm Supported	Data Block Size (Bits)	Key Size (Bits)
HMAC-SHA1	512	160-512
HMAC-MD5	512	128-512

7.5.3 Cipher Modes

There are three cipher modes supported by the NPE:

- Electronic code book (ECB)
- Cipher block chaining (CBC)
- Counter mode (CTR)

7.5.4 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.

7.5.5 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).

7.5.6 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.

7.5.7 Detailed Cryptographic NPE Data Flow

Communication between the Security Hardware Accelerator (NPE C) and the Intel XScale core processor is facilitated by two mechanisms:

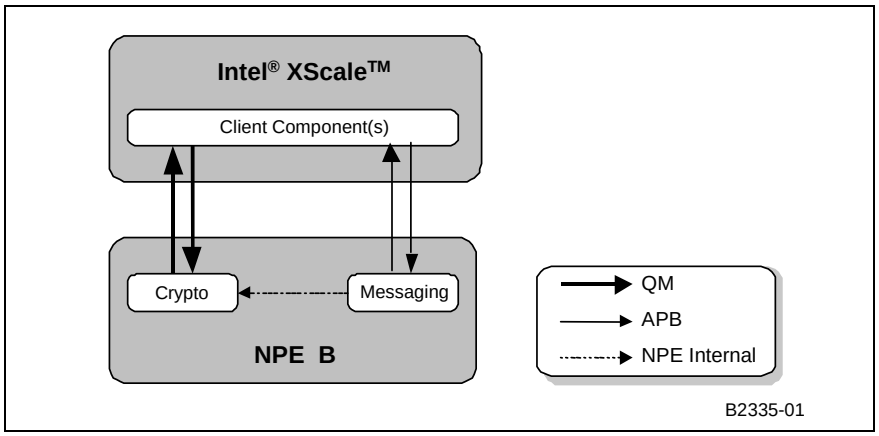
- Queue Manager — External to the NPE
- APB— The NPE accesses it via its internal debug and execution control block

Figure 27 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.

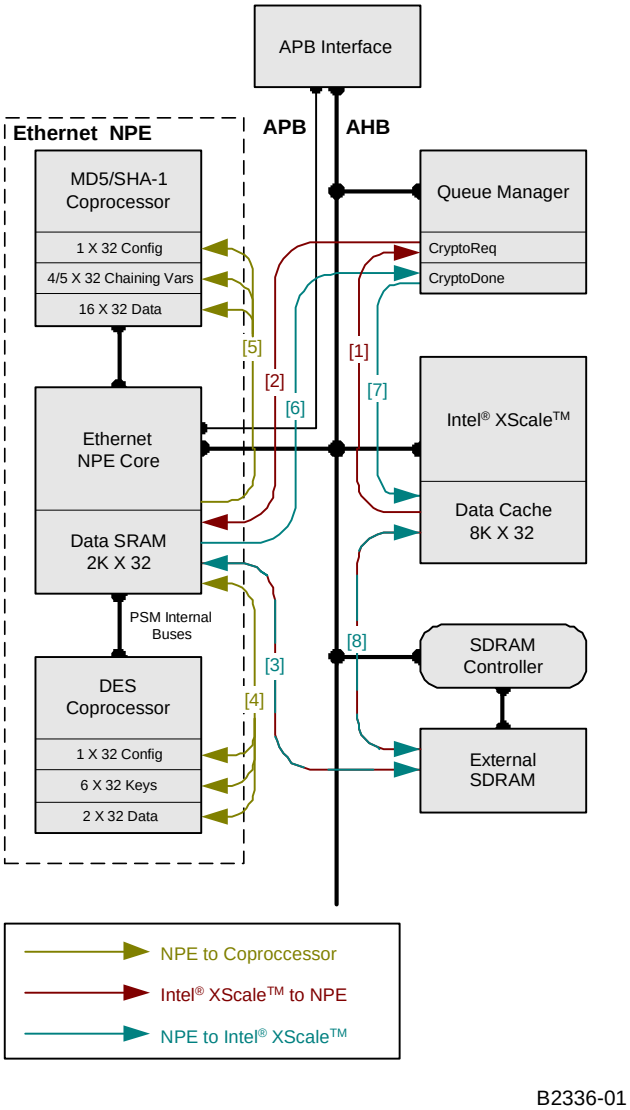
Figure 27. NPE IxCryptoAcc Component Interface



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 28 shows the detailed interaction between the Intel XScale core and the NPE.

Figure 28. 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]

7.6 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.

Access-Layer Components: DMA Access Driver (IxDmaAcc) API 8

This chapter describes the Intel® IXP400 Software v.1.3's "DMA Access Driver" access-layer component.

8.1 Overview

The IxDmaAcc provides DMA capability to offload large data transfers between peripherals in IXP42X product line processors' 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 buses (North AHB and South AHB buses). It also includes the APB bus, expansion bus, and PCI bus.

The ixDmaAcc component allows 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.

8.2 Features

The IxDmaAcc component provides these features:

- A DMA Access-layer API
- Clients' parameters validation
- Queues DMA requests (FIFO) to the Queue Manager

8.3 Assumptions

The DMA service is predicated on the following assumptions:

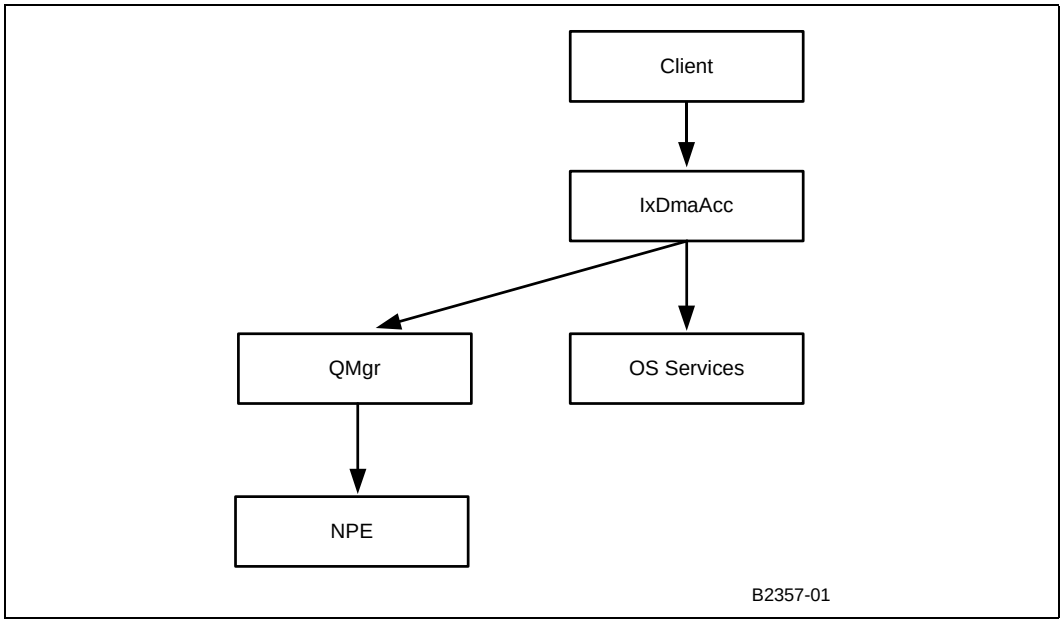
- IxDmaAcc has no knowledge about IXP42X product line processors' memory map.The client needs to verify the validity of the source address and destination address of the DMA transfer.
- IxDmaAcc has no knowledge on the devices that involve in the DMA transfer. The client is responsible for ensuring the devices are initialized and configured correctly before request for DMA transfer.

8.4 Dependencies

Figure 29 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

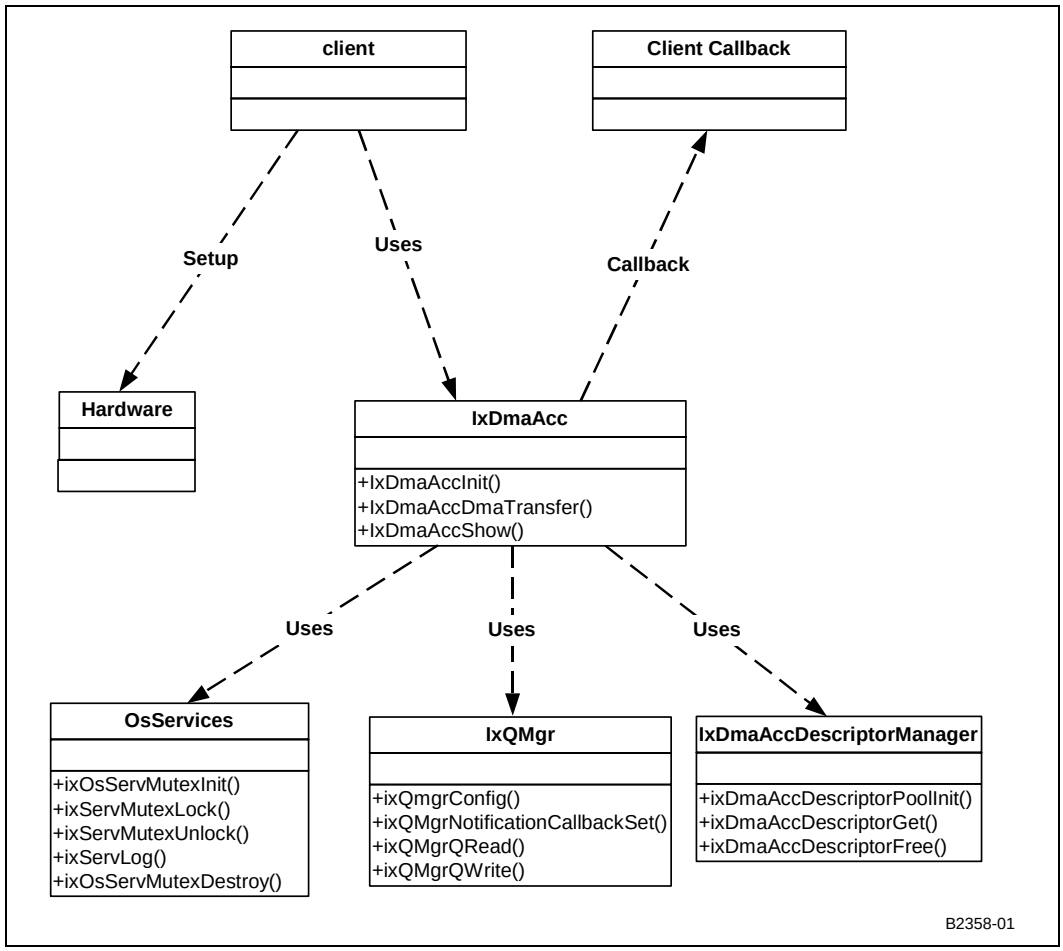
Figure 29. ixDmaAcc Dependencies



8.5 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 30. IxDmaAcc Component Overview



Note: IxDmaAcc components are in white.

Figure 30 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 IXP4xx 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).

8.5.1 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.

8.6 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.

8.6.1 Source Address

Source address is a valid IXP42X product line 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 memory map.

8.6.2 Destination Address

Destination address is a valid IXP42X product line 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 memory map.

8.6.3 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 byte swapped. The bytes are swapped within word boundary (for example, 0x 01 23 45 67 -> 0x 67 45 23 01 where the numbers indicate the source word and destination byte swapped word in the memory).
- **Copy and Bytes Reverse** — Moves the data from source to destination. The data written to the destination is byte reversed. The bytes are swapped across word boundary (for example, 0x 01 23 45 67 -> 0x 76 54 32 10 where the numbers indicate the source word and destination byte reversed word in the memory).

8.6.4 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).

8.6.5 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).

8.6.6 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

8.6.7 Supported Modes

This section summarizes the transfer modes supported by the IxDmaAcc. Some of the supported modes have restrictions. For details on restrictions, see "Restrictions of the DMA Transfer" on page 105.

Table 23. 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 24. 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 25. 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

8.7 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.

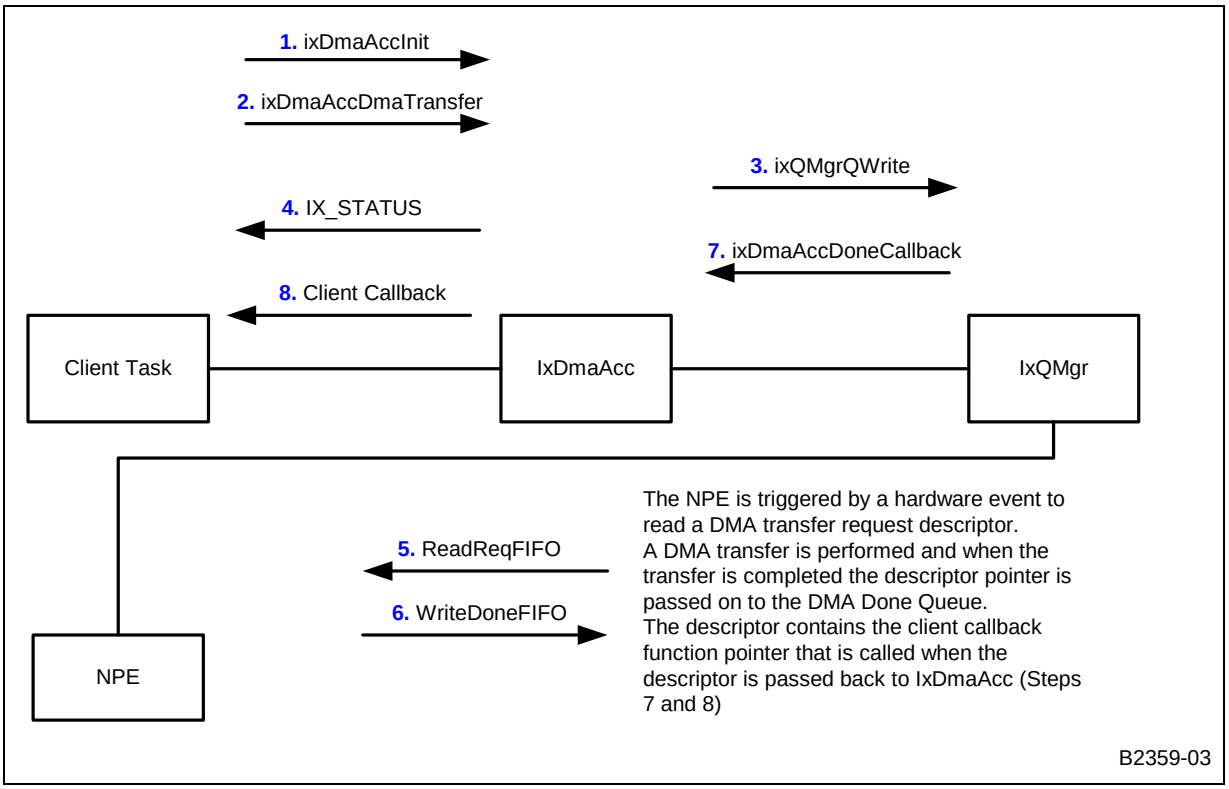
8.8 Control Flow

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 31 shows the overall flow of the DMA transfer operation between the client, the access layer, and the NPE.

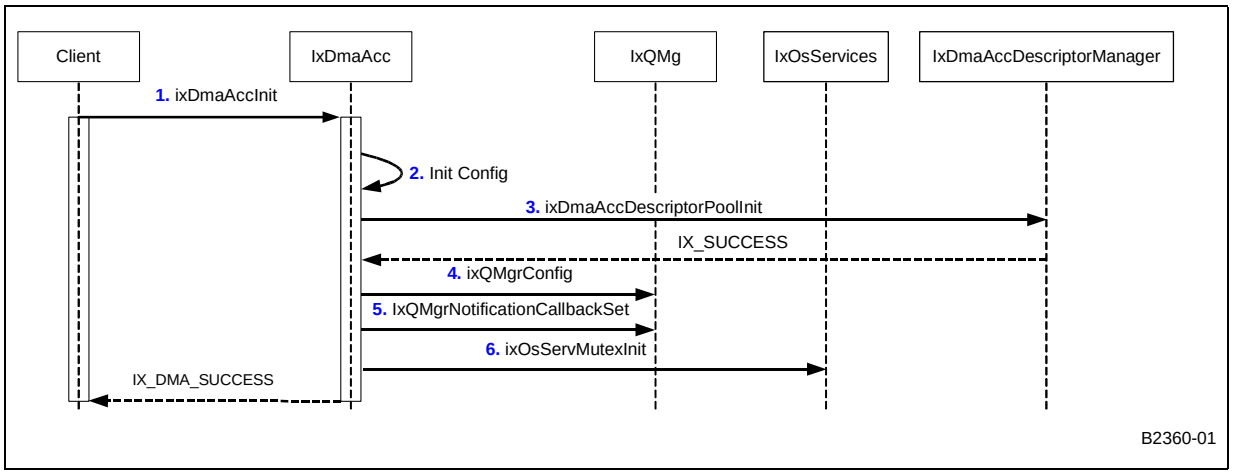
Figure 31. IxDmaAcc Control Flow



8.8.1 DMA Initialization

Figure 32 and the following steps describe the DMA access-layer initialization:

Figure 32. 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.

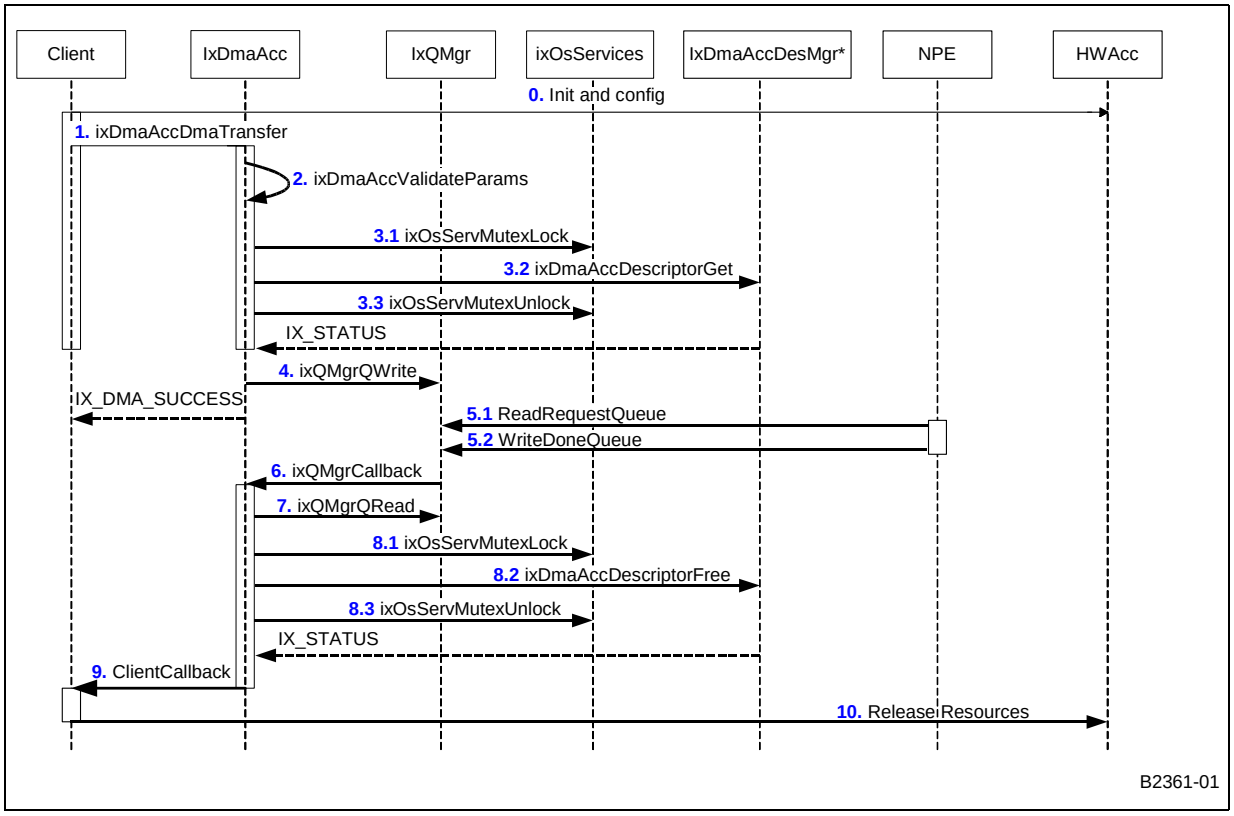
   The mutex ID will be used to access queue descriptor entry pool.

   ixDmaAccInit returns IX_DMA_SUCCESS upon completion of the DMA initialization.

8.8.2 DMA Configuration and Data Transfer

Figure 33 describes the configuration and DMA data transfer between a client and an NPE.

Figure 33. 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.

8.9 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’s 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.

8.10 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 error-reporting techniques, such as the OS services error-reporting mechanism.

8.11 Little Endian

This component does not work in Little Endian mode, nor will codelets that utilize this component.

Access-Layer Components: Ethernet Access (IxEthAcc) API 9

This chapter describes the Intel® IXP400 Software v.1.3's "Ethernet Access API" access-layer component.

9.1 What's New

New features specific to IxEthAcc — since IXP400 software release 1.1 — include:

- The MAC Address Learning and Filtering database component and the PHY Configuration component were decoupled from the IxEthAcc API. They now reside in IxEthDB and IxEthMii, respectively.
- It is no longer necessary to download the NPE firmware in between IxEthPortDisable() and IxEthPortEnable() calls.
- Disabling Ethernet port 2 (hosted on NPE-C) will not cause other NPE co-processors on NPE-C to halt. This may allow other NPE-C hosted access layer component (such as IxCryptoAcc) to continue operating.
- Four functions were added: IxEthAccPortMulticastAddressLeaveAll(), IxEthAccPortMulticastAddressJoinAll(), IxEthAccPortTxFrameAppendPaddingEnable(), and IxEthAccPortTxFrameAppendPaddingDisable().

9.2 IxEthAcc Overview

The IxEthAcc component (along with it's related components, IxEthDB and IxEthMii) provides data plane, control plane, and management plane information for the Ethernet MAC devices residing on IXP42X product line processors. The IXP42X product line processors contain one or two 10/100-Mbps Ethernet MAC devices depending on which processor variants are 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 which is memory mapped to the Intel XScale core.

The IxEthAcc component is strictly limited to supporting the internal Ethernet MACs on IXP42X product line processors.

The services provided by the Ethernet Access component include:

- Ethernet Frame Transmission
- Ethernet Frame Reception
- Ethernet MAC Statistics, Tracking and Reporting
- Ethernet Usage of the IxEthDB Filtering/Learning Database

The Ethernet NPEs also provide AAL-5-to-Ethernet fast-path packet modification. These capabilities are not accessible via the IxEthAcc component, but are part of the Fast-Path (IxFpathAcc) Access Layer 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, but rather the IxEthMii component. 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 IxEthMii.

The BSD-based buffering scheme is used as a mechanism for Ethernet frame transmission and reception. This scheme avoids excessive copying of data and maintain a high level of performance.

9.3 Ethernet Access Layers: Architectural Overview

IxEthAcc is not a standalone API. It relies on services provided by a number of other components. These firmware modules, APIs and messaging services support IxEthAcc's primary role of managing the scheduling, transmission and reception of Ethernet traffic.

9.3.1 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 the collection of MAC statistics. On the Ethernet transmit path, the Ethernet NPE firmware performs priority queuing of outgoing frames and MAC statistics collection, and destination port lookup based upon destination MAC address. 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 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 the software release 1.3 version of 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.

9.3.2 Queue Manager

Communication between an Ethernet NPE and the Intel XScale core is facilitated by two mechanisms.

The AHB Queue Manager is a hardware block that communicates buffer pointers between the NPE cores and the Intel XScale core. The IxQMgr API provides the queuing services to the access-layer and other upper level software executing on the Intel XScale core. 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 Ethernet coprocessor which serves as an interface between the Ethernet MAC and the NPE core block. 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 and if the frame is received without error, the frame's source MAC address is submitted to the learning service.

9.3.3 Learning/Filtering Database

IxEthAcc relies on the IxEthDB component for the MAC learning and filtering required in a routing or bridging application.

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.

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.

9.3.4 MAC/PHY Configuration

IxEthMii is used primarily to manipulate a minimum number of necessary configuration registers on Ethernet PHYs supported on the Intel® IXDP425 / IXCDP1100 Development Platform and the Coyote* Gateway Reference Design, without the support of a third-party operating system. Codelets and software used for Intel internal validation are the consumers of this API, although it is provided as part of the IXP400 software for public use.

While the MAC configuration is performed within IxEthAcc, the PHY configuration requires both IxEthAcc and IxEthMii. Since the MAC also controls the MDIO interface that is used for configuring the PHY, IxEthMii must initialize the MAC in order for the PHY to be configured. IxEthAcc initializes the MAC and virtual memory mapping and executes all register reads/writes on the PHY. IxEthMii provides the register definitions for supported PHYs. Thus, IxEthMii and IxEthAcc are dependant upon each other.

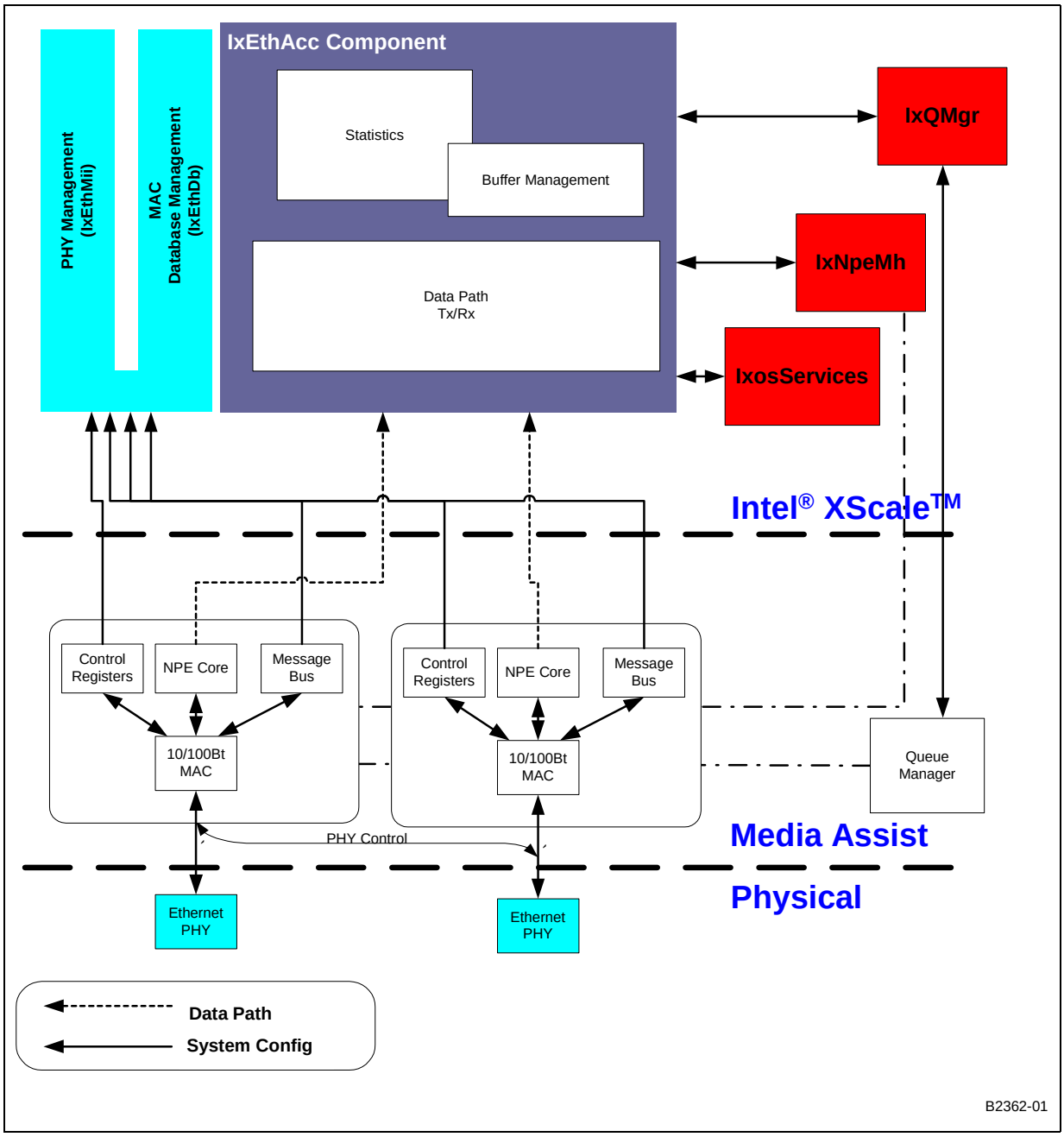
9.4 Ethernet Access Layers: Component Features

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

- **Data Path —** Responsible for the transmission and reception of IEEE 803.2 Ethernet frames. The Data Path is performed by IxEthAcc.
- **Control Path —** Responsible for the control of the MAC interface characteristics and some learning/filtering database functions. Control Plane functionality is included in both IxEthAcc and IxEthDB

- **Management Information —** Responsible for retrieving counter and statistical information associated with the interfaces. IxEthAcc provides this management support.

Figure 34. Ethernet Access Layers - Block Diagram



9.5 Data Plane

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

9.5.1 Port Initialization

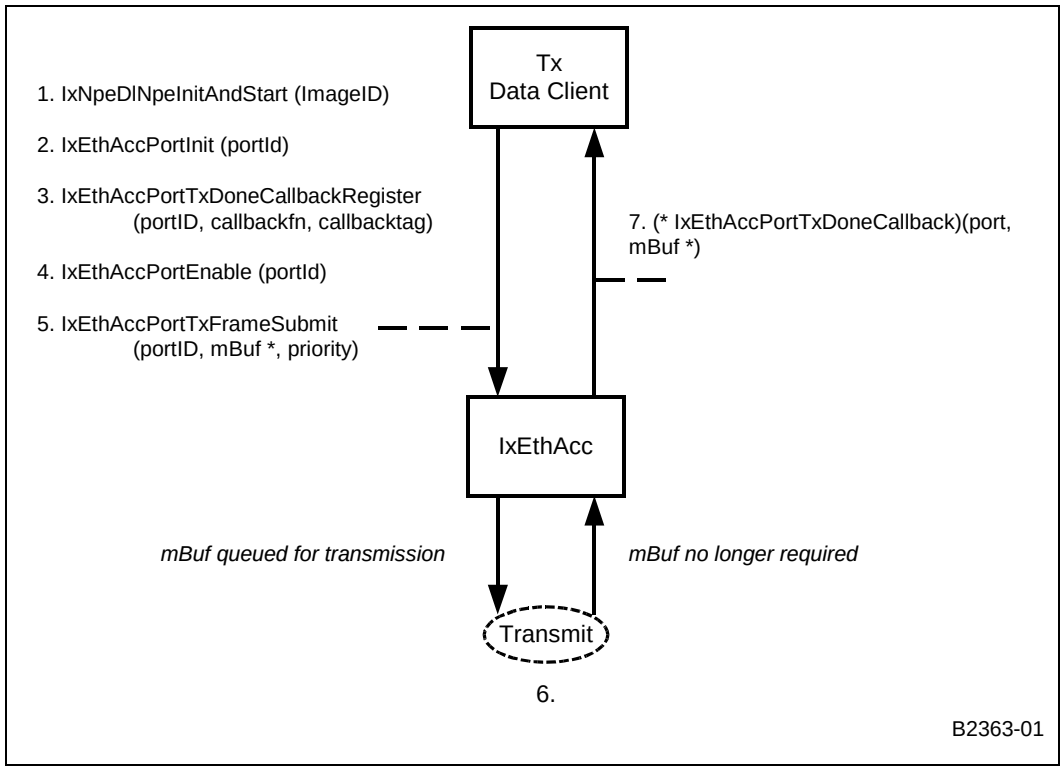
Prior to any operation being performed on a port, the appropriate microcode must be downloaded to the NPE using the IxNpeDl component.

The IxEthAccPortInit() function initializes all internal data structures related to the port and checks that the port is present before initialization. The Port state remains disabled even after IxEthAccPortInit() has been called. The port is enabled using the IxEthAccPortEnable() function.

9.5.2 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 MBufMBuf is no longer required by the component, it is returned from the Ethernet access component via a free buffer callback mechanism. The flow of Ethernet frame transmission is shown in "Ethernet Transmit Frame API Overview" on page 111.

Figure 35. Ethernet Transmit Frame API Overview



9.5.2.1 Transmission Flow

1. Proper NPE images must be downloaded to the NPEs and initialized.
2. The transmitting port must be initialized.
3. Register a callback function for the port. This function will be called when the transmission buffer is placed in the TxDone queue.

4. After configuring the port, the transmitting port must be enabled in order for traffic to flow.
5. Submit the frame, setting the appropriate priority. This places the MBuf on the transmit queue for that port.
6. IxEthAcc transmits the frame on the wire.
7. When transmission is complete, the MBuf is placed in the TxDone queue. The callback function is passed a pointer to that MBuf.

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.

9.5.2.2 Transmit Buffer Management and Priority

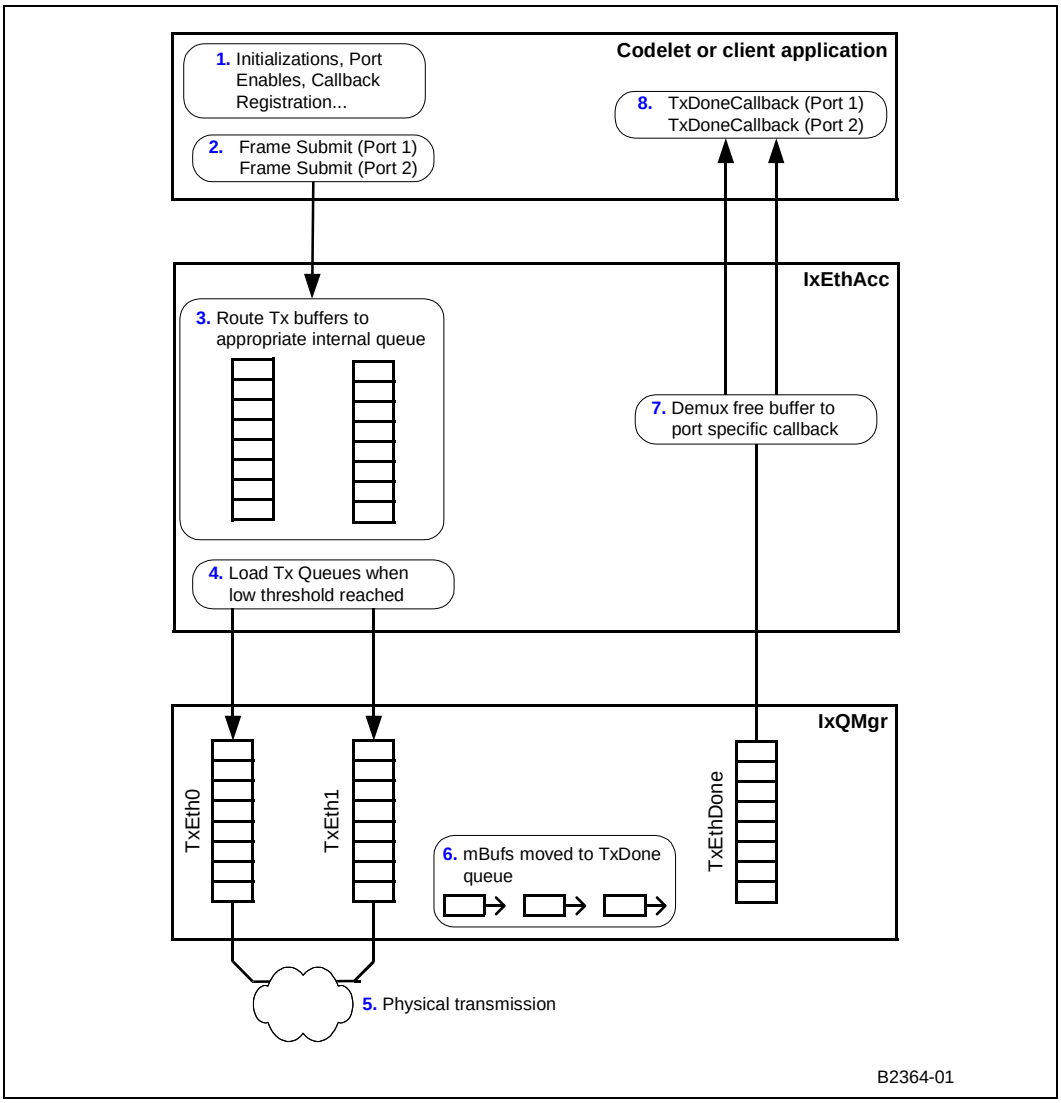
The key interface to the transmit data path NPEs is via a selection of IxQMgr queues. There are individual frame transmit queues assigned for each NPE Ethernet port. Another single queue is used to multiplex buffer complete/done messages from the NPEs.

The IxQMgr 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 queue entries due to fast path component requirements). For more information on the fast path dependency see the IX_ETH_ACC_FPATH_AWARE define in IxEthAccFpathDep.h. All internally queued buffers submitted for transmission but not queued to the hardware queues are stored in software queues. If priority FIFO queuing is being used, the frames will be saved in individual per priority FIFOs.

"Ethernet Transmit Frame Data Buffer Flow" on page 113 provides a visual explanation of queue management for Ethernet transmission.

Frames will be submitted to the port specific IxQMgr queue when a low/empty threshold is reached on the queue. Once frame transmission has completed, the buffer is placed on a buffer done IxQMgr 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 IxEthAccTxDoneCallbackRegister( ).

Figure 36. Ethernet Transmit Frame Data Buffer Flow



There are two scheduling disciplines selectable via the IxEthAccTxSchedulerDiscipline (). The frame submit behavior will be different for each case. Available scheduling disciplines are No Priority and Priority.

Tx FIFO No Priority

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

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 IxQMgr must be kept quite low to improve fairness among packets submitted. Once the low threshold on the IxQMgr 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.

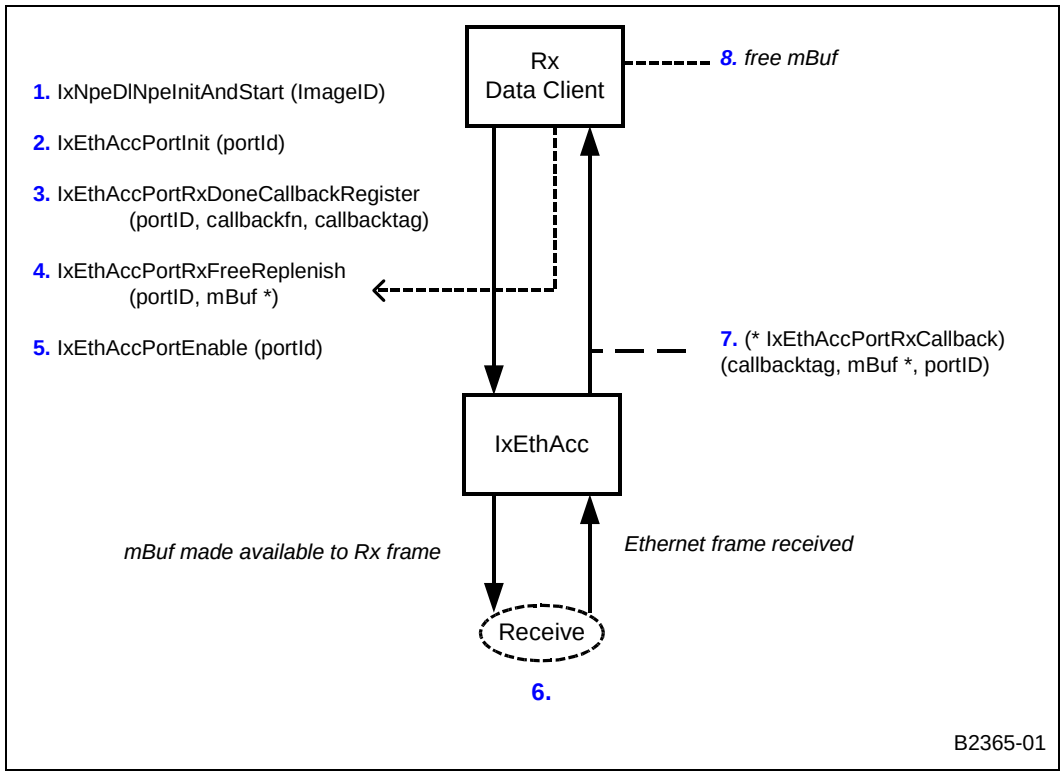
9.5.2.3 Additional Ethernet Transmission Information

Submission of chained MBuf clusters for transmission is supported, but excessive chaining may 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.

9.5.3 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 flow of Ethernet frame reception is shown in “Ethernet Receive Frame API Overview” on page 115.

Figure 37. Ethernet Receive Frame API Overview



9.5.3.1 Receive Flow

1. Proper NPE images must be downloaded to the NPEs and initialized.
2. The receiving port must be initialized.
3. Register a callback function for the port. This function will be called each time a frame is received.
4. Preload free receive buffers for use by IxEthAcc.
5. After configuring the receiving port and pre-loading buffers, the receiving port is enabled, allowing traffic to be received.
6. An ethernet frame is received on the wire and placed in the IxQMgr Rx queue.
7. The callback function is called for each frame, being passed a pointer to that MBuf. The callback function can now process and/or de-multiplex the incoming frame(s).
8. The upper-level user or OS processes must recover the receive buffers once processing of the frame is completed, and replenish the RxFree queue using IxEthAccPortRxFreeReplenish() as needed.

9.5.3.2 Receive Buffer Management

The key interface from the NPEs to the receive data path (IxEthAcc) is a selection of queues residing in the queue manager hardware component. These queues are shown in "Ethernet Receive Plane Data Buffer Flow" on page 117.

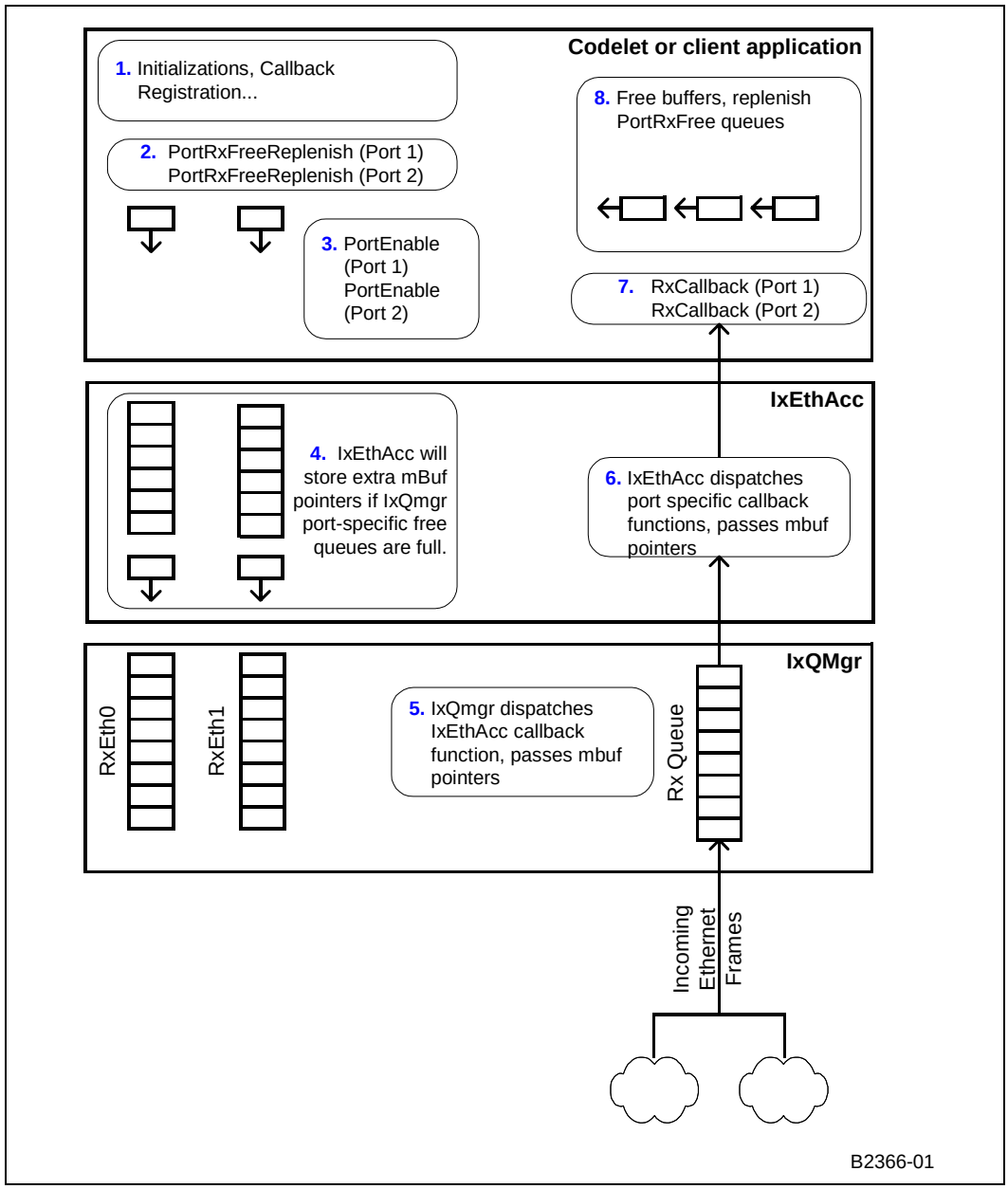
The receive data plane subcomponent must provide receive buffers to the NPEs. The buffers are supplied on a per port basis via the user level interface IxEthAccRxBufferReplenish(). There are two separate free buffer IxQMgr queues allocated to providing the NPEs with receive buffers (one per port). The replenish function loads the port specific free buffer IxQMgr queue with an IX_MBUF pointer. If the port specific free buffer IxQMgr 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.

The user also must ensure that there are sufficient buffers assigned to this component to maintain wire speed Ethernet receive performance. If the receive NPE does not have a receive buffer in advance of receiving an Ethernet frame, the frame will be dropped. Should a frame arrive while there are no free buffers is available, no callback indication will be provided and a rx_buffer_underrun counter will be incremented.

Received frames from both NPEs are multiplexed onto one queue manager queue. The IxEthAcc component will de-multiplex the received frames and call the associated user level callback function registered via IxEthAccRxCallbackRegister(). The frames placed in the IxQMgr 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. Note that the receive callback is issued in frame-receive order. No receive priority mechanisms are provided.

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). Buffers can also be freed by disabling the port, using the IxEthAccPortDisable() function.

Figure 38. Ethernet Receive Plane Data Buffer Flow



There is no specific port flush capability. To retrieve submitted buffers from the system, the port must be disabled, using the IxEthAccPortDisable() function. 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).

9.5.3.3 Additional Receive Path Information

An Rx polling interface is not 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.

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.

The maximum supported MTU for Ethernet frames is 1536 bytes. Any frames larger than 1536 bytes are rejected by the NPE firmware.

9.5.4 Data-Plane Endianness

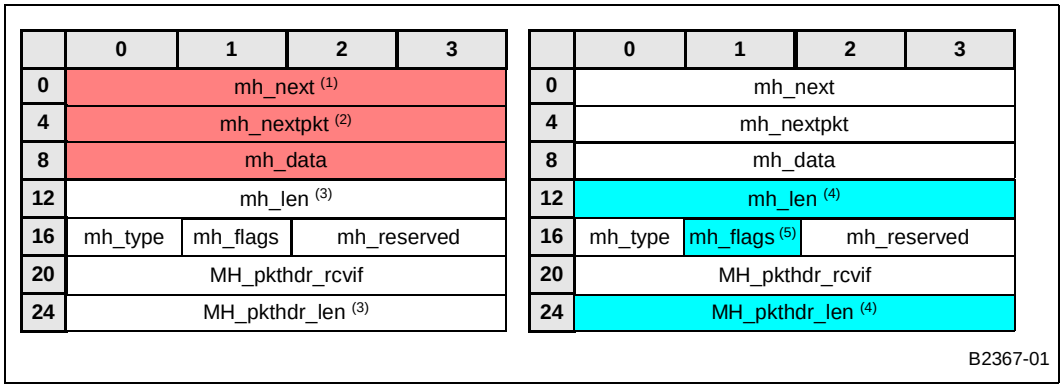
All data structures passed to the component are defined in host byte order. No changes to data structures are required in order to use the access component data path interfaces. The data pointed to by the MBuf is expected to be in network byte order (big endian). No byte swapping takes place on the data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

9.5.5 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 NPE 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.

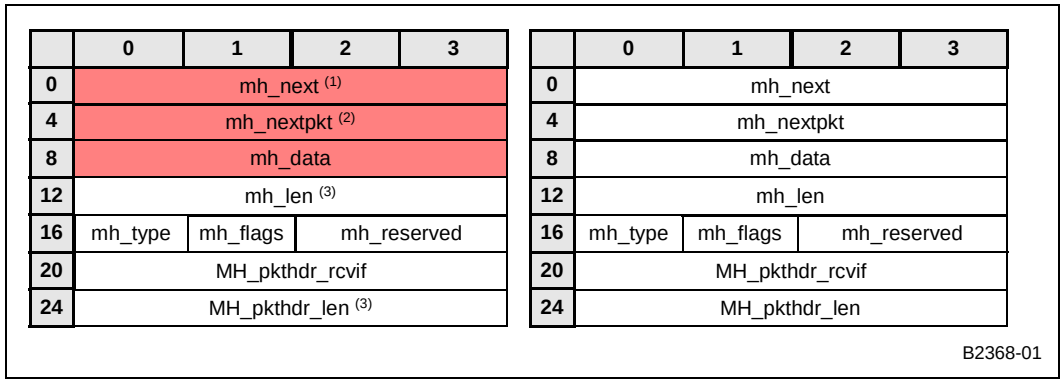
Figure 39. MBuf Fields Written by Intel XScale® Core (left) and NPE (right) on Ethernet Rx Path



Notes:

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 processors' 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 1,536 bytes (and preferably 2 Kbyte) 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 40. MBuf Fields Written by Intel XScale® Core (left) and NPE (right) on Ethernet Tx Path



Notes:

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 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).

9.6 Control Path

The main control path functions are performed by two external components, IxEthMii and IxEthDB.

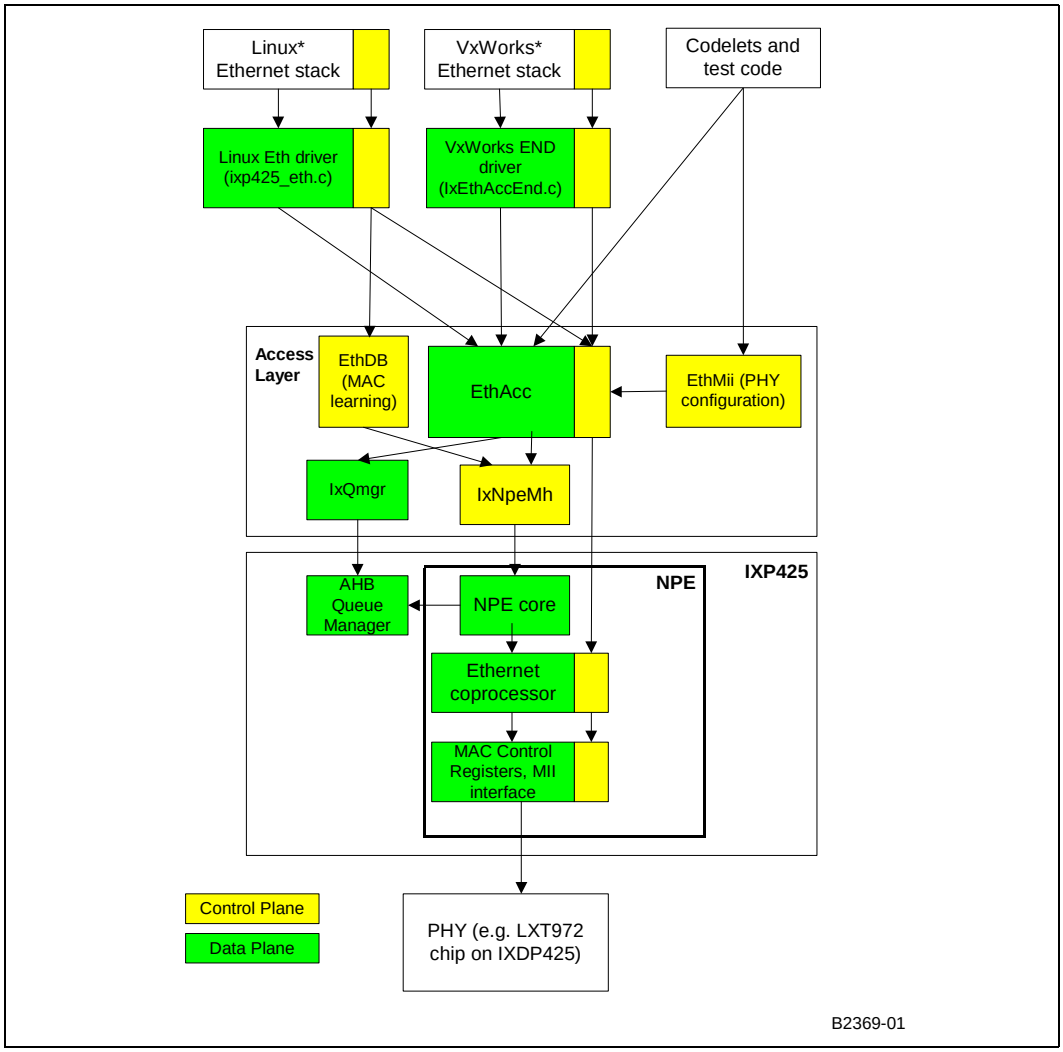
IxEthMii is used primarily to manipulate a minimum number of necessary configuration registers on Ethernet PHYs supported on the IXDP425 / IXCDP1100 platform and the Coyote gateway platform, without the support of a third-party operating system. IxEthMii exists as a separate function in order to make IxEthAcc independent of the specific PHY devices used in a system.

However, IxEthAcc does retain control of configuring the Ethernet MAC devices on the NPEs and drives the MII and MDIO interfaces, which are used IxEthMii to communicate physically with the PHYs.

IxEthDB is the learning and filtering database. IxEthDB handles the database structure, maintenance, searching, aging, and the provisioning of dynamic and static addresses. This database runs in the context of the NPEs, though an API is provided to the access-layer.

The relationship between IxEthAcc, IxEthDB, and IxEthMii is demonstrated in Figure 41.

Figure 41. IxEthAcc and Secondary Components



The control path component remaining for IxEthAcc is the provision of the MAC registers with their required functionality.

9.6.1 Ethernet MAC Control

The role and responsibility of this module is to enable clients to configure the Ethernet coprocessor MACs for both NPEs. This API permits the setting and retrieval of uni-cast and multi-cast addresses, duplex mode configuration, FCS appending, frame padding, promiscuous mode configuration, and reading or writing from the MII interface.

9.6.1.1 MAC Duplex Settings

Functions are provided for setting the MACs at full or half duplex. This setting should match the setting of the connected PHYs.

9.6.1.2 MII I/O

IxEthAcc provides three functions that interact with the MII interface. The MAC must be enabled with IxEthAccMacInit () first.

- IxEthAccMiiReadRtn () — Read a 16 bit value from a PHY
- IxEthAccMiiWriteRtn () — Write a 16 bit value from a PHY
- IxEthAccMiiStatsShow () — Displays the values of the first eight PHY registers

9.6.1.3 Frame Check Sequence

An API is provided to provision whether the MAC appends an IEEE-803.2 Frame Check Sequence (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 and should be set before a port is enabled.

9.6.1.4 Frame Padding

The IxEthAcc component by default will add up to 60-bytes to any Tx frames submitted that do not meet the Ethernet required minimum of 64-bytes. When padding is enabled, FCS appending will also be turned on.

Frame padding may not be desirable in all situations, such as when generating a "heartbeat" signal to other nodes on the network. To disable frame padding, the function IxEthAccPortTxFrameAppendPaddingDisable() is available.

This feature is available on a per-port basis and should be set before a port is enabled.

9.6.1.5 MAC Filtering

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. NPE-level filtering will not function in IxEthDB unless the MACs are configured in promiscuous mode.

Non-Promiscuous Mode

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

- Frame destination MAC address = Provisioned uni-cast MAC address
- Frame destination MAC address = Broadcast address
- Frame destination MAC address = Provisioned multi-cast MAC addresses

Address Filtering

The following functions are provided to manage the MAC address tables:

- IxEthAccPortMulticastAddressJoinAll() — all multicast frames are forwarded to the application.
- IxEthAccPortMulticastAddressLeaveAll() — Rollback the effects of IxEthAccPortMulticastAddressJoinAll().
- IxEthAccPortMulticastAddressLeave() — Unprovision a new filtering address.
- IxEthAccPortMulticastAddressJoin() — Provision a new filtering address.
- IxEthAccPortPromiscuousModeSet() — All frames are forwarded to the application regardless of the multicast address provisioned.
- IxEthAccPortPromiscuousModeClear() — Frames are forwarded to the application following the multicast address provisioned.

9.7 Management Information

The IxEthAcc 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 26 and Table 27. For more details on these statistics objects, see RFC 2665.

These APIs are provided to retrieve these statistics:

- IxEthAccMibIIStatsGet() — Returns the statistics maintained for a port
- IxEthAccMibIIStatsGetClear() — Returns and clears the statistics maintained for a port
- IxEthAccMibIIStatsClear() — Clears the statistics maintained for a port

Table 26. 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 27. 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

This page intentionally left blank.

Access-Layer Components: Ethernet Database (IxEthDB) API 10

This chapter describes the Intel® IXP400 Software v.1.3's "Ethernet Database API" access-layer component.

10.1 Overview

To minimize the unnecessary forwarding of frames, an IEEE 802.1d-compliant bridge maintains a filtering database. IxEthDB provides MAC address-learning and filtering database functionality that can be easily extended beyond the Ethernet NPE interfaces.

10.2 What's New

In IXP400 software release 1.1, the MAC Address Learning and Filtering database component was decoupled from the ixEthAcc component. Changes since software release 1.1 include:

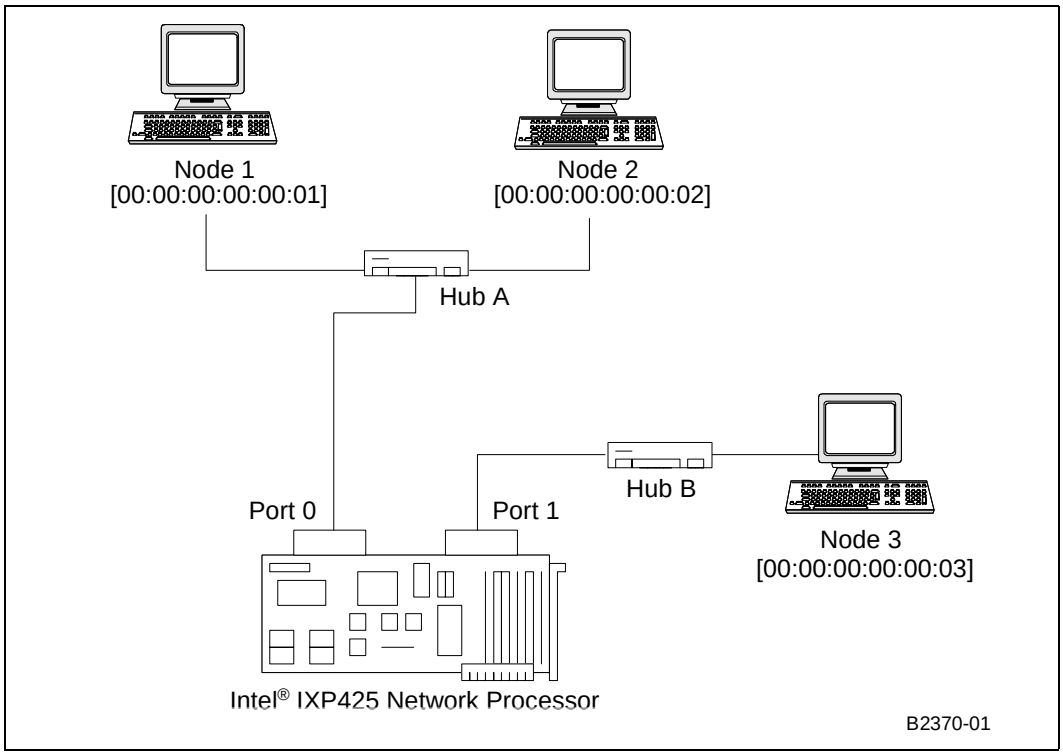
- IX_ETH_DB_NUMBER_OF_PORTS is now calculated from the ixEthDBPortDefinitions table, and should no longer be set manually
- Disabling a port now automatically removes all learned access entries for that port from the database.

10.3 Address Filtering Theory

The NPEs have learning/filtering capabilities which, when enabled, allow the system to operate in a half-bridge mode. This mode of operation consists of saving *source* MAC addresses — extracted from Rx frames — into *learning trees* and performing search operations on *destination* MAC addresses from Rx frames. If, when receiving a frame, the destination MAC address is found in the search tree of the port that received the frame, then the frame is dropped (filtered). The reasoning behind this decision is that frames received on a port whose destination is a node connected to the same network-side of the port should not be forwarded to other parts of the network.

This process is illustrated below in "Example Network Diagram for MAC Learning" on page 126. Consider the following network topology in conjunction with a bridging (simple frame forwarding) application running on the IXP42X product line processors:

Figure 42. Example Network Diagram for MAC Learning



Assuming we start with blank (empty) learning trees, a possible scenario of filtering is the following:

- Node 1 sends a frame to Node 3 (source MAC 00:00:00:00:00:01, destination 00:00:00:00:00:03)
  - The frame is forwarded by Hub A to Node 2 (ignores the frame, as the destination does not match its own address) and Port 0
  - Port 0 adds the source address (00:00:00:00:00:00:01) to its learning tree
  - Port 0 searches for the destination address (00:00:00:00:00:03) in its learning tree, it is not found therefore the frame is forwarded to the other ports – in this case Port 1
  - Port 1 forwards the frame to Hub B
  - Hub B forwards the frame to Node 3, intended recipient of the frame
- Node 2 sends a frame to Node 1 (source MAC 00:00:00:00:00:02, destination 00:00:00:00:00:01)
  - The frame is sent to Hub A which forwards it to Node 1 (intended recipient) and Port 0
  - Port 0 adds the source MAC address (00:00:00:00:00:02) to its learning tree
  - Port 0 searches for the destination address (00:00:00:00:00:01) in its learning tree, it is found therefore Port 0 knows that both Node 1 and Node 2 are connected on the same side of the network, and this network already has a frame forwarder (in this case Hub A) – the frame is filtered (dropped) to prevent unnecessary propagation

10.4 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.

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. Dynamic entries subject to aging may also be provisioned via the API. It is important to note that if a static MAC address is provisioned for port X but later a frame having this source MAC address is detected arriving from port Y, the record in the database will be updated from X to Y and the record will no longer be marked as static.

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 with these considerations:

- 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 IxEthDB API.
- Removing dynamic entries by aging, controlled by the IxEthDB API.
- 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.

Each NPE is capable of learning 511 MAC address. The Intel XScale core's learning database will handle all the addresses for both NPEs plus any number of addresses required for user-defined ports. It is not recommended, to add more than 511 addresses per NPE port.

The database will accommodate by default up to 2,000 records. This will suffice for the two NPEs and a small number of user-defined ports plus operating headroom, however if the figure is not large enough the user can tweak database pre-allocation structures by changing ixp425_xscale_sw/src/ethDB/include/IxEthDB_p.h.

A high-level overview of the system is provided in "Ethernet Receive Frame API Overview" on page 129.

Figure 43. Ethernet Receive Frame API Overview

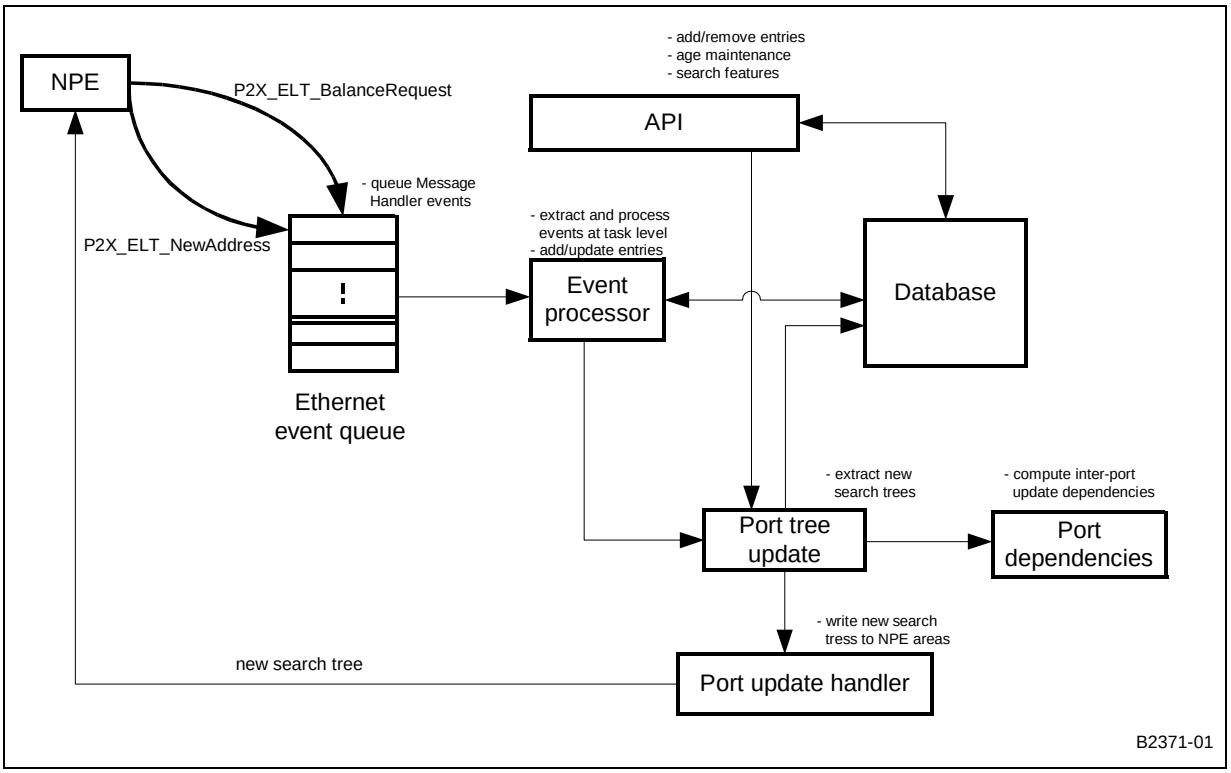


Figure 43 exemplifies the main event flows from NPEs and API, which may trigger changes to the database, which in turn determines tree updates that provides the NPEs with updated binary search trees. It should be noted that the design allows for ports that are not specifically Ethernet NPEs to be integrated 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

10.4.1 EthDB Initialization

IxEthDB performs an ixFeatureCtrlSwConfigurationCheck( ) to determine the value of IX_FEATURECTRL_ETH_LEARNING. IxEthDB uses this value to decide whether or not to activate the NPE-based EthDB learning and to spawn an Intel XScale core thread to monitor it.

Any component or codelet can modify the value prior to IxEthDB initialization using ixFeatureCtrlSwConfigurationWrite(IX_FEATURECTRL_ETH_LEARNING, [TRUE or FALSE]). Once IxEthDB has been initialized, the software configuration cannot be changed.

IX_FEATURECTRL_ETH_LEARNING is TRUE by default.

10.4.2 Promiscuous-Mode Requirement

MAC address-frame filtering, based on learning trees, is usable only when a port operates in *promiscuous* mode. Otherwise the frames will be filtered at MAC and not NPE level according to normal MAC filtering rules. Those filtering rules are that the frame is received only if one of the following is true:

- The destination address matches the port address
- The destination is the broadcast address, or c) if the destination is a multicast address subscribed to by the port.

Configuration of promiscuous mode and multi-cast configuration is described in the section for IxEthAcc, “MAC Filtering” on page 121.

10.4.3 Port Definitions

IxEthDB is not directly dependant on the two Ethernet ports available on the Intel® IXP425 Network Processor. The user can define up to 32 ports (including the two Ethernet NPE ports), which will be recognized by the component, although these definitions are static and cannot be changed at run-time. The only requirement is that port ID 0 and 1 are reserved for Ethernet NPE B and Ethernet NPE C and cannot be used for user ports (nor should they be removed).

Port definitions are located in the public include filexscale_sw/src/include/IxEthDBPortDefs.h. The main port definition table is an array having the following format:

```
static IxEthDBPortDefinition ixEthDBPortDefinitions[] =
{
/*      id       type           capabilities */
  {   /* 0 */    ETH_NPE,       ENTRY_AGING },/* Ethernet NPE B */
  {   /* 1 */    ETH_NPE,       ENTRY_AGING },/* Ethernet NPE C */
  {   /* 2 */    ETH_GENERIC,   NO_CAPABILITIES } /* WAN port */
};
```

The first two entries are reserved and the user can add additional ports starting with ID 2. The definitions listed above include the two Ethernet NPEs and one example user-defined WAN port. Port numbers (IDs) are automatically determined from the definition location – they are written as comments above only for clarity reasons.

All user ports must be defined as ETH_GENERIC with NO_CAPABILITIES. Unlike the Ethernet NPEs, user-defined ports lack certain capabilities. “ETH_GENERIC” describes a port with no automatic database update features, and “NO_CAPABILITIES” describes a port unable to age its corresponding entries. This characterization will instruct IxEthDB not to attempt to upload up-to-date learning trees in user ports, and instructs IxEthDB to age the entries itself instead of relying on external logic.

Do not change or remove the first two ports — the IxEthAcc component relies on this definition. Accordingly, IX_ETH_DB_NUMBER_OF_PORTS should have a value of at least two at any time. Other components may have also defined their own ports (see the header file for up-to-date information).

Note: Software releases previous to IXP400 software release 1.3 *do not* automatically update the IX_ETH_DB_NUMBER_OF_PORTS symbol based on the definitions array. In early software releases, the user manually defined IX_ETH_DB_NUMBER_OF_PORTS. Beginning in software release 1.3, IX_ETH_DB_NUMBER_OF_PORTS is calculated from the ixEthDBPortDefinitions table.

10.4.4 Selective Port Disabling

Ethernet NPEs will perform by default MAC-learning and filtering operations when operating in promiscuous mode. If this behavior is not desired, the IxFeatureCtrl component provides the ability to disable learning/filtering operations on the NPEs by disabling IxFeatureCtrlSwConfig(IX_FEATURECTRL_ETH_LEARNING). If this configuration is disabled, no NPE filtering will be present, nor will it be possible to enable it at any stage without recompiling the component. Disabling NPE filtering affects for NPE ports (Eth 0 and Eth 1) and user-defined (non-NPE ports) ports.

10.4.5 Static Entries

As stated earlier, the IxEthDB API provides the ability to add and remove static MAC/PORT entries. If a static MAC address is provisioned for port X but later a frame having this source MAC address is detected arriving from port Y the record in the database will be updated from X to Y, and the record will no longer be marked as static.

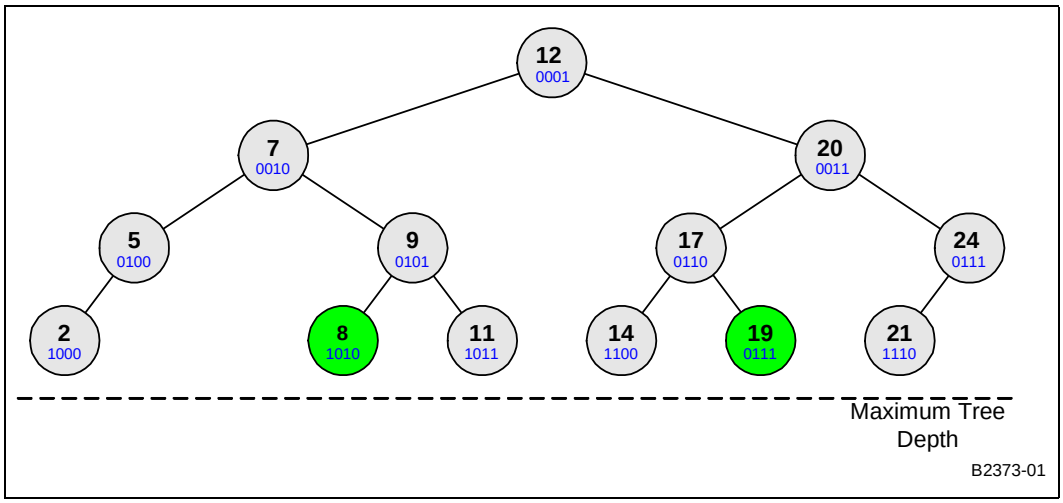
10.4.6 Database Maintenance

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/port pairs for a period of roughly fifteen minutes). If so, they are removed from the filtering database.

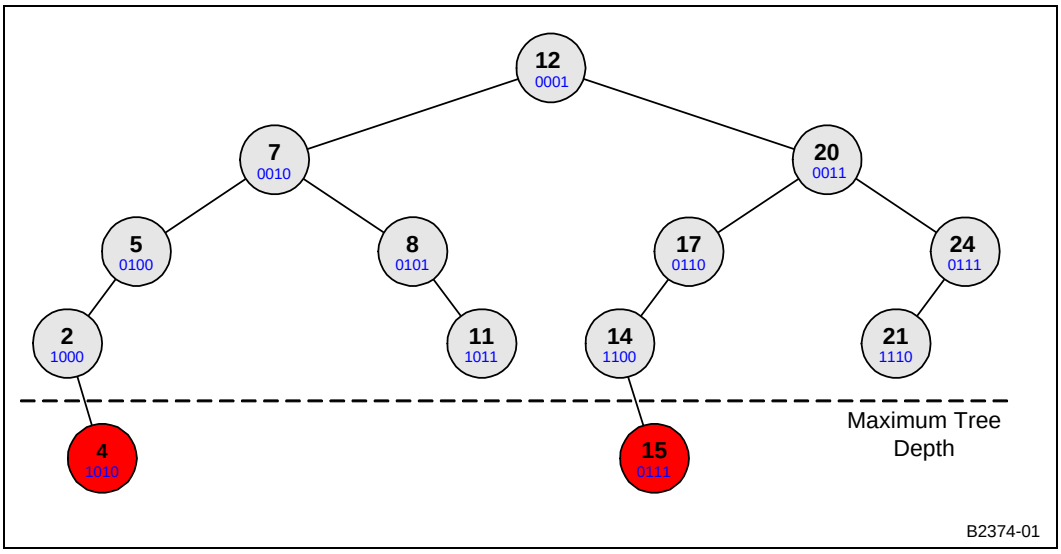
As new MAC addresses are inserted into the filtering database via the learning process, the binary tree structure used to maintain this database tends to become unbalanced.

Figure 44. 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 45. Examples of Node Insertion Requiring Re-Balancing



The ixEthDB component performs all database maintenance functions. To facilitate this, the ixEthDBDatabaseMaintenance() function must be called with a frequency of IX_ETH_DB_MAINTENANCE_TIME.

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

Note: It is the client's responsibility to ensure the ixEthDBDatabaseMaintenance() function is executed with the required frequency. The default value of IX_ETH_DB_MAINTENANCE_TIME is one minute.

10.4.7 Database Elements

MAC addresses are unique keys in the database. Therefore, no records can share the same address. Adding an existent address on a different port, either using the API or when detected by an NPE, will automatically update the existing record with the new port information. In real terms this corresponds to moving a network node on a different network segment, which will automatically update the database information as soon as the node sends packets intercepted by an IXP425 NPE. Since MAC address duplication is an error condition on a network, the database is designed to prohibit duplicate addresses on the same or different ports.

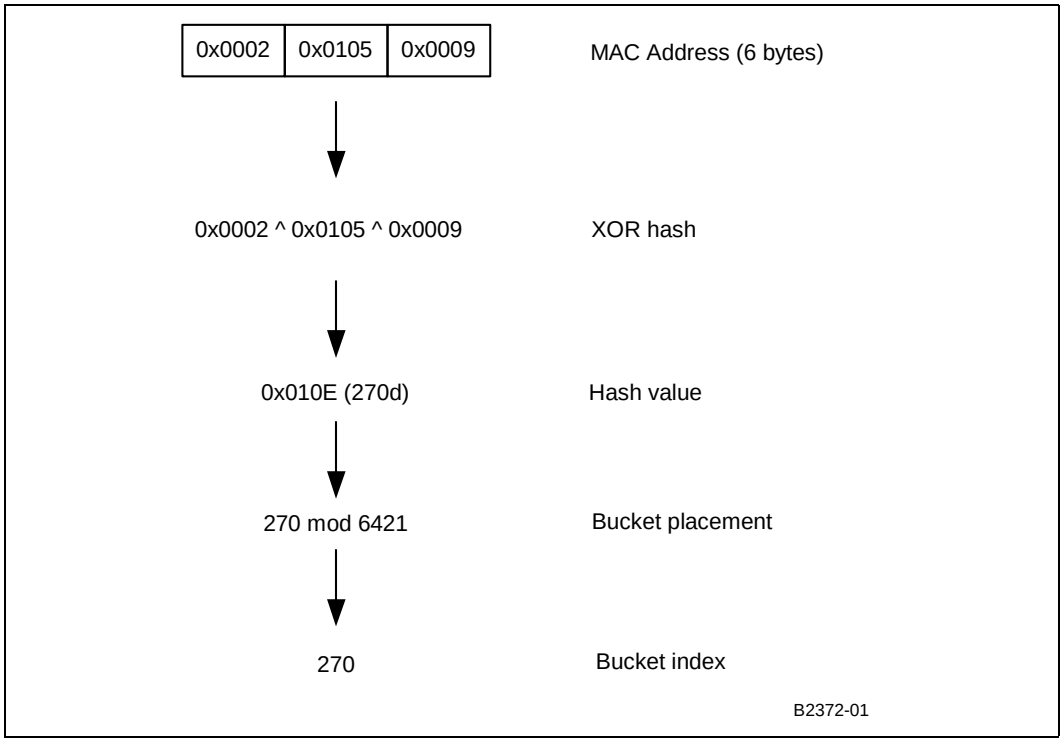
10.4.8 Algorithms Used by the Ethernet Learning Tree

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, thereby 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. "Hashing" on page 134 describes the process of hashing a MAC address entry, assuming 6421 buckets are used.

Figure 46. 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 eight to 10 times the expected number of entries, ideally prime number of buckets. For approximately 600 and 1,000 entries, the values of 6,421 and 12,853 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-utilization and speed. It executes in O(n) and it requires no extra memory storage apart from the original tree. The resulting trees are route balanced yet not perfectly balanced since creating the second variety is more expensive time-wise without resulting in improvement 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.

Access-Layer Components: Ethernet PHY (IxEthMii) API 11

This chapter describes the Intel® IXP400 Software v.1.3's "Ethernet PHY API" access-layer component.

11.1 Overview

IxEthMii is used primarily to manipulate a minimum number of necessary configuration registers on Ethernet PHYs supported on the Intel® IXDP425 / IXCDP1100 Development Platform and the Coyote* Gateway Reference Design, without the support of a third-party operating system. Codelets and software used for Intel internal validation are the consumers of this API, although it is provided as part of the Intel® IXP400 Software for public use.

11.2 What's New

The IxEthMii API is a new component that was introduced after software release 1.1. In software releases 1.0 and 1.1, the PHY register configuration functions were part of IxEthAcc. New features added to IxEthMii since software release 1.1 include:

- Support for additional PHYs
- Loopback support

11.3 Features

The IxEthMii components provide the following features:

- Scan the MDIO bus for up to 32 available PHYs
- Configure a PHY link speed, duplex, and auto-negotiate settings
- Enable or disable loopback on the PHY
- Reset the PHY
- Gather and/or display PHY status and link state

11.4 Supported PHYs

The supported PHYs are listed in the table below. Other Ethernet PHYs are also known to use the same register definitions but are unsupported by this software release (e.g. Intel® 82559 10/100 Mbps Fast Ethernet Controller). Register definitions are located ixp425_xscale_sw/src/ethMii/IxEthMii_p.h.

Table 28. PHYs Supported by IxEthMii

Intel® LXT971 Fast Ethernet Transceiver
Intel® LXT972 Fast Ethernet Transceiver
Intel® LXT973 Low-Power 10/100 Ethernet Transceiver
Micrel / Kendin* KS8995 5 Port 10/100 Switch with PHY

11.5 Dependencies

IxEthMii is used by the EthAcc codelet and is dependant upon the IxEthAcc access-layer component.

Access-Layer Components: Feature Control (IxFeatureCtrl) API 12

This chapter describes the Intel® IXP400 Software v.1.3's "Feature Control API" access-layer component.

IxFeatureCtrl is a component that detects the capabilities of the IXP42X product line processor and provides a configurable software interface that can be used to simulate different processors variants in the IXP42X product line.

12.1 What's New

The IxFeatureCtrl API is a new component that was introduced after software release 1.1.

12.2 Overview

IxFeatureCtrl provides three major functions. First, functions are provided that read the hardware capabilities of the processor. The IxFeatureCtrl API is also capable of disabling the peripherals or components on the processor. Finally, the API provides a modifiable software configuration structure that can be read or modified by other software components to determine the run-time capabilities of a system.

12.3 Hardware Feature Control

Detecting and controlling the hardware features of the processor is performed using the Product ID and the Feature Control Register. The product ID is returned from the function ixFeatureCtrlProductIdRead() which reads register 0 of the processors' co-processor 15. This register contains the maximum frequency that the Intel® XScale™ core is capable of running, although not necessarily the actual speed of operation (the platform could be clocked down to a lower frequency). The product ID also contains the stepping of the processor, which is also returned by ixFeatureCtrlProductIdRead().

Note: The ProductID register is read-only. This function will not lower the operating frequency of the processor.

Table 29. Product ID Values (Sheet 1 of 2)

Bits	Description
31:28	Reserved. Value: 0x6
27:24	Reserved. Value: 0x9
23:20	Reserved. Value: 0x0
19:16	Reserved. Value: 0x5

Table 29. Product ID Values (Sheet 2 of 2)

Bits	Description
15:12	Reserved. Value: 0x4
11:4	Maximum Achievable Xscale Core Frequency. 533 MHz: 0x1C 400 MHz: 0x1D 266 Mhz: 0x1F
3:0	Si Stepping ID. A-step: 0x0 B-step: 0x1

The Feature Control Register is a structure which contains information on which components are physically available on the processor. The detectable capabilities include the existence of key coprocessors or peripherals (PCI controller, AES co-processor, NPEs, etc.).

This register is also the mechanism which can disable components of the processor. You cannot disable components that do not exist.

Table 30. Feature Control Register Values (Sheet 1 of 2)

Bits	Description
31:18	(Reserved)
17:16	UTOPIA PHY Limits. 32 PHYs: 0x0 16 PHYs: 0x1 8 PHYs: 0x2 4 PHYs: 0x3
15	(Reserved)
14 †	PCI Controller
13 †	NPE C
12 †	NPE B
11 †	NPE A
10 †	Ethernet 1 Coprocessor
9 †	Ethernet 0 Coprocessor
8 †	UTOPIA Coprocessor
7 †	HSS Coprocessor
6 †	AAL Coprocessor
5 †	HDLC Coprocessor
4 †	DES Coprocessor
3 †	AES Coprocessor

† For bit 0 through 14, the following values apply:

- 0x0 — The hardware component exists and is not software disabled.
- 0x1 — The hardware component does not exist, or has been software disabled.

Table 30. Feature Control Register Values (Sheet 2 of 2)

Bits	Description
2 †	Hashing Coprocessor
1 †	USB Coprocessor
0 †	RComp Circuitry

† For bit 0 through 14, the following values apply:

- 0x0 — The hardware component exists and is not software disabled.
- 0x1 — The hardware component does not exist, or has been software disabled.

12.4 Software Configuration

A software configuration structure and supporting functions are provided that can be modified at runtime. The software configuration structure is an array that stores the enable/disable state of a particular run-time modifiable configuration. Other software components can be designed to read or write the software configuration array enable or disable certain software features prior to initialization.

In software release 1.3, there is only one entry in the software configuration array. IxEthDb performs an ixFeatureCtrlSwConfigurationCheck( ) to determine the value of IX_FEATURECTRL_ETH_LEARNING. IxEthDb uses this value to decide whether or not to activate the NPE-based EthDB learning and to spawn an Intel XScale core thread to monitor it. Any component or codelet can modify the value prior to IxEthDb initialization using ixFeatureCtrlSwConfigurationWrite(IX_FEATURECTRL_ETH_LEARNING, [TRUE or FALSE]). Once IxEthDB has been initialized, the software configuration cannot be changed.

12.5 Dependencies

- IxFeatureCtrlProductIdRead() is used to identify the stepping of the silicon in other parts of IXP400 software. Subsequently, this information is used to switch to either A0 or B0 branches within affected components. For example, the switch can used to differentiate between code with workaround in-place for A0 silicon and code without workaround for B0 silicon.
- IxNpeDl uses the function ixFeatureCtrlComponentCheck to prevent the erroneous download of NPE firmware to disabled or unavailable NPEs.

This page intentionally left blank.

Access-Layer Components: Fast-Path Access (IxFpathAcc) API 13

This chapter describes the Intel® IXP400 Software v.1.3's "fast path" access-layer component.

The IxFpathAcc component provides access to the fast path services provided by the NPEs. This service classifies incoming AAL-5 packets on the ATM NPE (NPE-A), modifies the packets according to set modification instructions, and transmits them via one of the Ethernet NPEs (NPE-B).

13.1 What's New

The following major change has been implemented since the Intel® IXP400 Software v1.1:

- The modifier engine has been enhanced to add IP and TCP/UDP checksum

13.2 Functional Description

Figure 47. Data and Control Flow for Fast Path

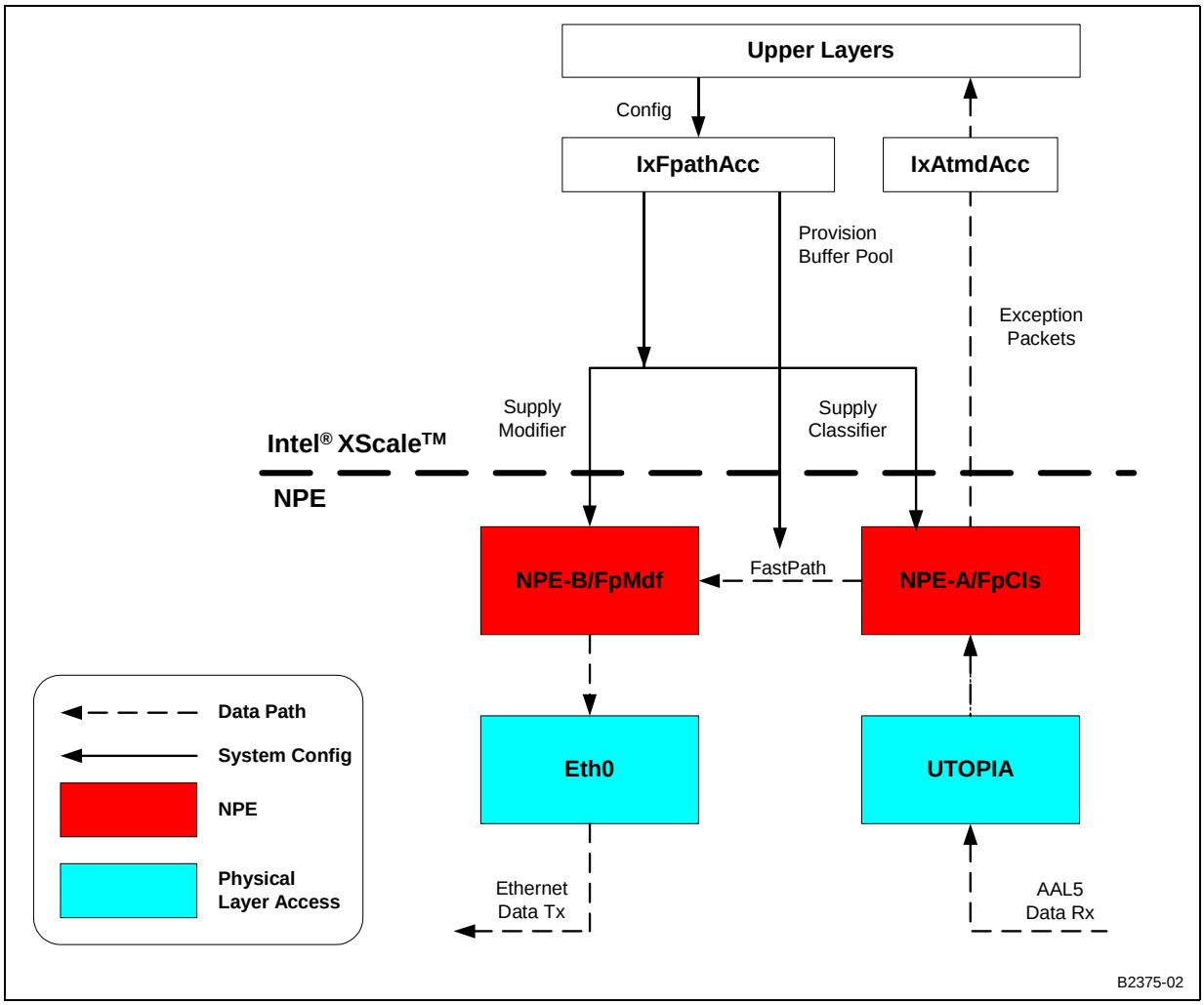


Figure 47 shows the data and control flow for the fast path service. The IxFpathAcc component provides access to the underlying fast path services provided by NPEs. Currently, only NPE-A (ATM) and NPE-B (Ethernet) together support fast path functionality.

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. Packets that do not meet the classification template criteria are sent to IxAtmdAcc for further processing.

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

13.3 Enabling/Modifying/Disabling Fast Path for a VC

A fast path controlling entity above 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.

13.4 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 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, however the modifier engine only supports IPv4 at this time.

The modifier also updates the IP or TCP/UDP checksum for outbound fast path Ethernet packets prior to transmission.

13.5 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.

IxFpathAcc also provides a number of statistics related to Fast Path configuration and operation.

13.6 Fast-Path Buffers

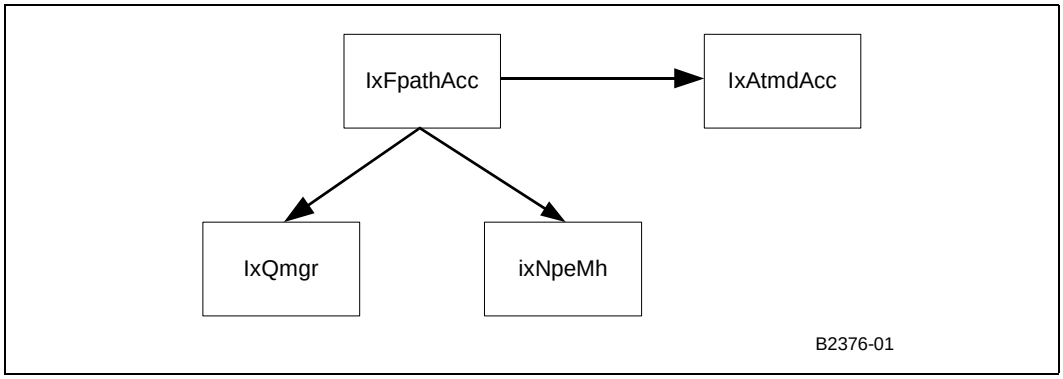
The fast-path service requires buffers that are used to store and convey packets on the fast path. Buffers supplied to IxAtmdAcc (slow path buffers) are never sent on the fast path.

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.

13.7 Fast-Path Dependency Diagram

Figure 48. Fast-Path Dependency Diagram



IxQmgr and IxNpeMh

IxQmgr is used by IxFpathAcc to supply fast-path buffers to the NPEs. 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. IxQmgr and IxNpeMh must be configured before IxFpathAcc can be initialized.

IxAtmdAcc

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.

The fast path cannot be enabled for a VC if the VC has not already been connected on IxAtmdAcc. A valid npeVcId is required at all times, so if a VC configured via IxAtmdAcc is reconfigured resulting in a different npeVcId, then the template pair must be refreshed.

IxEthDb

The IxEthDb component must be configured for Ethernet Port 0 in order for traffic to flow on the fast path.

IxEthAccFpathDep

Enabling the IxFpath component substantially changes the transmit queue management behavior of IxEthAcc. IxEthAcc must be made aware that IxFpath is enabled by modifying the #define IX_ETH_ACC_FPATH_AWARE value, located in IxEthAccFpathDep.h. Doing this has the effect of instructing IxEthAcc to reduce it's transmit queue usage for Eth 0 in half. IxFpath will use the remaining queues.

13.8 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 software release 1.3 error-reporting techniques, such as the OS-services and error-reporting mechanism.

13.9 NPE Fpath Classifier/Modifier Configuration

A proprietary set of instructions is used by the Fpath component to facilitate the construction of modifier and classifier templates for the fast-path capability.

Note: The Fast Path Access Codelet (IxFpathAccCodelet) provides additional utilities to create and view sample templates, and should be reviewed for further clarification in the use of the IxFpath component.

13.9.1 Fast-Path Classifier Template Definition

NPE-A interprets a sequence of bytes, including the opcodes listed below, as a classifier template.

Classifier Header

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 a 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.

Classifier Body

The classifier header is followed by a variable sequence of bytes which describe the instructions the classifier engine should take. "Classifier Instruction Format in the Classifier Template" on page 146 describes how these instructions are generally formatted in the byte structure of the template. "Opcode Operations in TempCls" on page 146 describes all of the available opcodes and their respective parameters and attributes.

Table 31. Classifier Instruction Format in the Classifier Template

Instruction	OpName	Offset	Operand	Parameters
Instruction shorthand	*Op*	*D0*	*Nd*	*Pt*
Length	1 byte	1 byte	1 byte	*Np* Bytes Depending on *Op*. (Optional)
Description	Operation. Unsigned char.	Position (or relative position, relative to the current location) of the first byte to check. Unsigned char. 16 is the first byte of the current cell. Values 0-15 indicate bytes in the previous cell and are therefore only valid after a Save16Bytes operation.	Number of bytes of data from the first byte needed to perform the checking. Unsigned char.	Some opcodes require opcode specific parameters. The number of which depends on the opcode and the OperandCount.

Table 32. Opcode Operations in TempCls (Sheet 1 of 2)

OpName *(Op)*	Opcode *(Op)* Value	Possible Operand *(Nd)* Values	Parameter Length-Values *(Np)*	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, 4, 8, 10	Nd	*Nd* byte data must be == *Nd* byte of *Pt*.
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, 16, 48	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[3]	6	N/A	2	One byte data must be either one of the two bytes in *Pt*.
MacWanLearning	7	N/A	0	Inspect and report WAN MAC source addresses
Conditional Branch[3]	8	Special*Nd*	1	If one byte matches, skip next “Special*Nd*” 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.

Table 32. Opcode Operations in TempCls (Sheet 2 of 2)

OpName *(Op)*	Opcode *(Op)* Value	Possible Operand *(Nd)* Values	Parameter Length-Values *(Np)*	Definition of the Operation and Parameter[1]
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 pre-pended 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 NOPpad3 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 TempCls pointer by one, i.e. past this NOPpad3 instruction.
reserved	13	N/A	N/A	THIS IS AN ILLEGAL OPCODE. It is used internally by the NPE to efficiently implement the NOP instructions above.

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 IxFpath classifier. The operation is always performed if a packet passes the classifier test, before sending the packet to FpathMdf.

13.9.2 Fast-Path Modifier Template Definition

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

The Per-PVC template is as shown in "Per-VC TempMdf" on page 148.

The template shown in Table 34 is also supplied for applying to all PVCs.

Table 33. Per-VC TempMdf

Bytes	Definition
0:1	Protocols enabled • Bit 0 = Bridged • Bit 1 = IP • Bit 2 = NAT • Bit 3 = NAPT • Others = (Unused)
2:3	Delta mData/mLen: To be added to mData unconditionally before mbuf is read. This value should be the difference between the base of AAL-5 PDU and one of the following: • Base of Ethernet packet for bridge • Base of IP packet for IP router
4:7	Byte 0-3 of destination MAC address of destination 0.
8:11	Byte 0-3 of destination MAC address of destination 1.
12:15	Byte 0-3 of destination MAC address of destination 2.
16:19	Byte 0-3 of destination MAC address of destination 3.
20:21	Byte 4-5 of destination MAC address of destination 0.
22:23	Byte 4-5 of destination MAC address of destination 1.
24:25	Byte 4-5 of destination MAC address of destination 2.
26:27	Byte 4-5 of destination MAC address of destination 3.
28:31	Local destination IP address of destination 0. (For NAT or NAPT. For NAT no table lookup is needed.)
32:35	Local destination IP address of destination 1. (For NAPT)
36:39	Local destination IP address of destination 2. (For NAPT)
40:43	Local destination IP address of destination 3. (For NAPT)
44:45	Local destination port number of destination 0. (For NAPT)
46:47	Local destination port number of destination 1. (For NAPT)
48:49	Local destination port number of destination 2. (For NAPT)
50:51	Local destination port number of destination 3. (For NAPT)
52:53	IP checksum difference for destination 0. This value is calculated by IxFpath.
54:55	IP checksum difference for destination 1. This value is calculated by IxFpath.
56:57	IP checksum difference for destination 2. This value is calculated by IxFpath.
58:59	IP checksum difference for destination 3. This value is calculated by IxFpath.
60:61	TCP/UDP port checksum difference for destination 0. This value is calculated by IxFpath.
62:63	TCP/UDP port checksum difference for destination 1. This value is calculated by IxFpath.
64:65	TCP/UDP port checksum difference for destination 2. This value is calculated by IxFpath.
66:67	TCP/UDP port checksum difference for destination 3. This value is calculated by IxFpath.

The Source MAC address and Ethernet Type are the same for all VCs and are stored in universally accessible global variable. These values are pre-pended to the Ethernet header during IxFpath processing, prior to transmission.

Table 34. Source/Type Template

Byte	Meaning
0:5	Local-source MAC address.
6:7	Ethernet type to be put into MAC header.

13.9.3 Sample Templates

The Fast Path Access Codelet (IxFpathAccCodelet) provides additional utilities to create and view sample templates, and should be used for further clarification in the use of the IxFpath component.

13.10 Little Endian

This component does not work in Little Endian mode, nor will codelets that utilize this component.

This page intentionally left blank.

Access-Layer Components: HSS-Access (IxHssAcc) API 14

This chapter describes the Intel® IXP400 Software v.1.3's "HSS-Access API" access-layer component.

14.1 Overview

The IxHssAcc component provides client applications with driver-level access to the High-Speed Serial (HSS) and High-Level Data Link Control (HDLC) coprocessors, which enable the IXP42X product line to communicate in a serial-bit fashion with "the outside world," using TDM data.

The bit-stream protocols supported are T1, E1, and MVIP with line speeds from 512 KHz and up to 8.192 MHz. Coupled with HDLC coprocessors, the HSS coprocessor can provide packetized service and/or channelized service.

The HSS access-layer API is contained in the file `IxHssAcc.h`. The file's individual elements are listed in "Functional Interface Description" on page 154. There are three structures used for HSS and HDLC configuration:

- IxHssAccConfigParams — Contains HSS-coprocessors configuration parameters
- IxHssAccPortConfig — Contains HSS-coprocessors configuration parameters
- IxHssAccPktHdlcFraming — Contains HDLC-coprocessors configuration parameters

This chapter provides the details of how to use the HSS-access component to:

- Initialize and configure the HSS and HDLC coprocessors
- Allocate data buffers for transmitting and receiving
- Connect and enable packetized service and/or channeled service
- Handle the transmitting and receiving process
- Disconnect and disable the services

14.2 Features

The HSS access component is used by a client application to configure both the HSS and HDLC coprocessors and to obtain services from the coprocessors. It provides:

- Access to the two HSS ports for the IXP42X product line
- Support for TDM signals up to a rate of 8.192 Mbps (four E1/T1) on a HSS port
- Support for one client of the NPE channelized service per HSS port
- Support for four clients of the NPE packetized service (or four HDLC channels) per HSS port or one client per E1/T1 trunk

- Channelized access to N x 64 Kbps (or 56 Kbps) channels within a TDM stream per HSS port, where 1<=N<=32
- Support for up to 4-Kbyte HDLC packets for transmission [1] in the form of chained mbufs
- Support for receiving HDLC packets with a recommended maximum size of 16 Kbytes
- Configuration of the HSS and HDLC coprocessor through the NPE

14.3 HSS and HDLC Coprocessors Operation

The HSS coprocessor enables IXP42X product line processors to communicate externally, in a serial-bit fashion, using TDM data. The bit-stream protocols supported are T1, E1, and MVIP. The HSS coprocessor also can interface with xDSL framers.

The HSS coprocessor communicates with an external device using three signals per direction: a frame pulse, clock, and data bit. The data stream consists of frames — the number of frames per second depending on the protocol. Each frame is composed of time slots — each time slot consisting of 8 bits (1 byte) and having a number that indicates the time slot's location within the frame.

The maximum frame size is 1,024 bit and the maximum frame pulse offset is 1,023 bit. The line clock speed can be set to one of the following values:

- 512 KHz
- 1.536 MHz
- 1.544 MHz
- 1.568 MHz
- 2.048 MHz
- 4.096 MHz
- 8.192 MHz

The frame size and frame offsets are all programmable according to differing protocols. Other programmable options include signal polarities, signal levels, clock edge, endianness, and choice of input/output frame signal. For more details, see "Initialization and Configuration" on page 156.

The time slots within a stream can be configured as HDLC, voice64K, or Voice56K or left unassigned. For HDLC time slots, data will be passed to the HDLC coprocessor for processing as packetized data. The HDLC coprocessor provides the bit-oriented HDLC processing for the HSS coprocessor. Both 16- and 32-bit CRC polynomials are supported.

The FIFO in the HSS coprocessor and the "channel switching" in the HDLC coprocessor can support up to four HDLC services per HSS port.

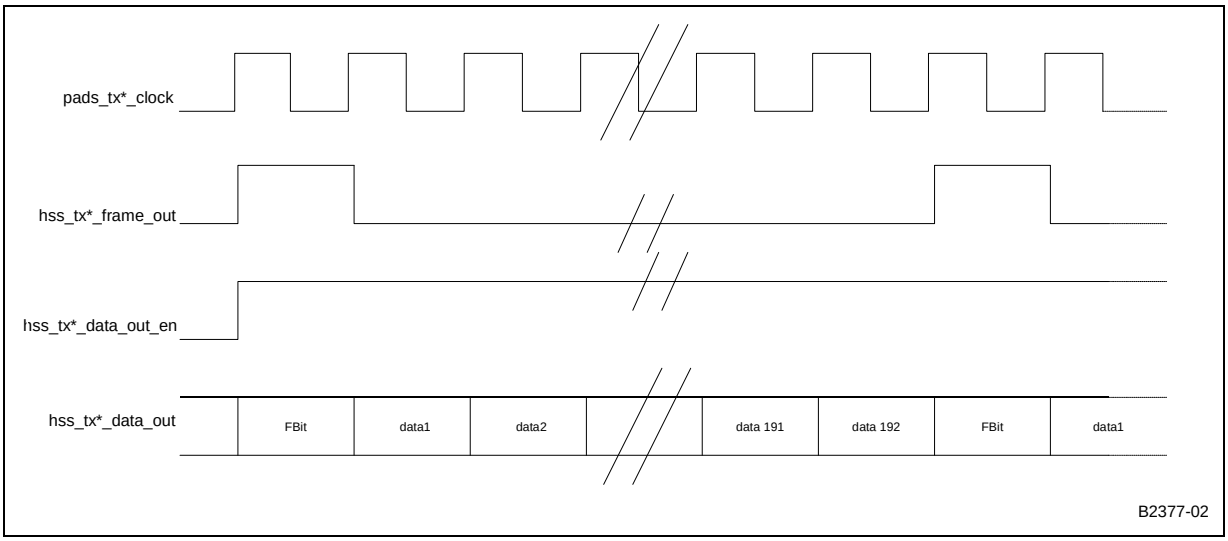
The following HDLC parameters are programmable:

- What to transmit when a HDLC port is idle.
- The HDLC data endianness
- The CRC type to be used for this HDLC port

Note: For more details, see "Packetized Connect and Enable" on page 170.

Figure 49 illustrates a typical T1 frame with active-high frame sync (level) and a posedge clock for generating data. If the frame pulse was generated on the negedge in the figure, it would be located one-half clock space to the right. The same location applies if the data was generated on the negedge of the clock.

Figure 49. T1 Tx Signal Format

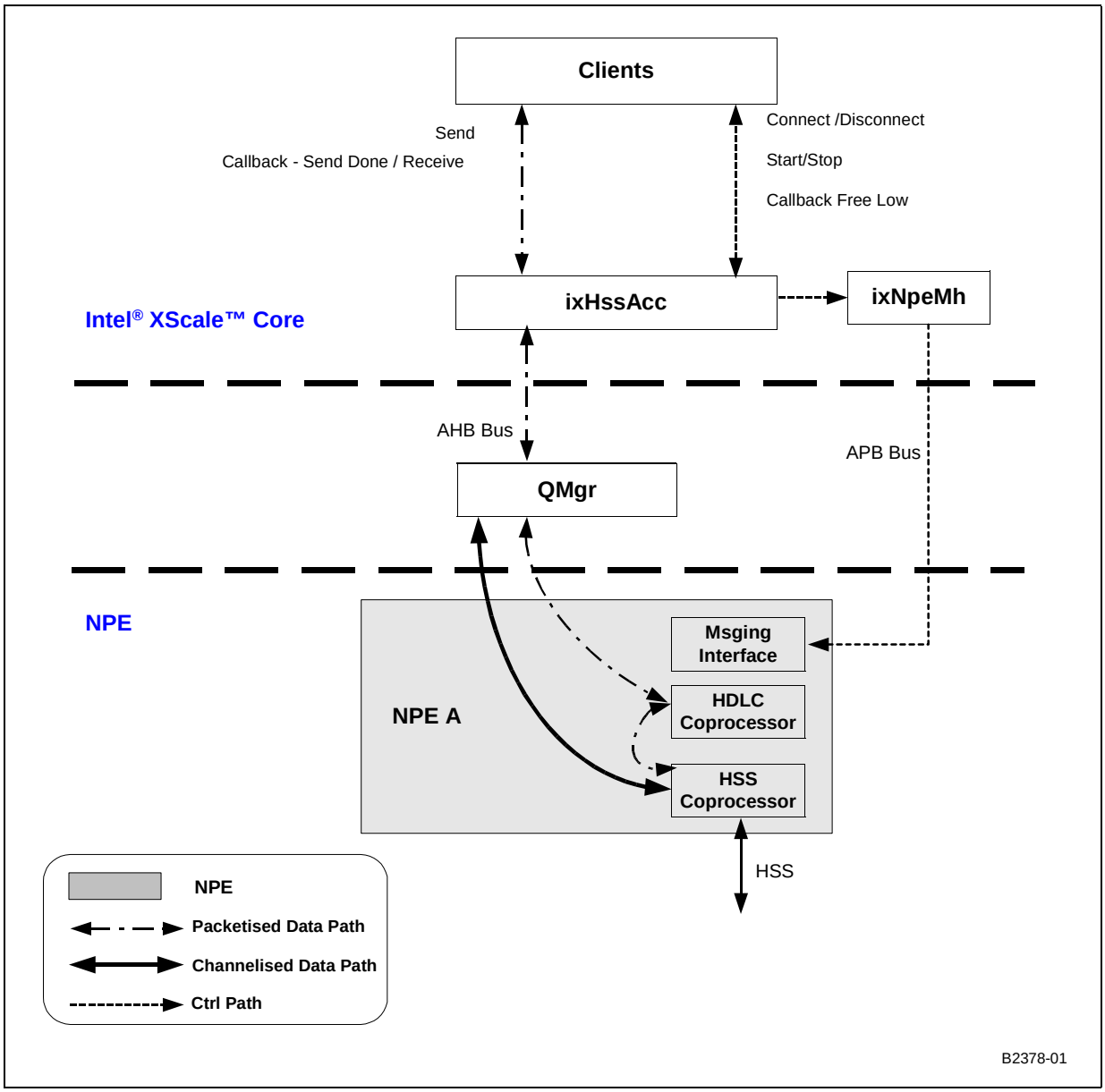


14.4 Control and Data Flow

An overview of the data and control flow for the HSS/HDLC is shown in Figure 50.

The client configures HSS through IxHssAcc to set the time slots in a HSS frame to provide either channelized or packetized service. The HSS component uses callback functions and data buffers provided by the client to exchange data for transmitting or receiving with the help of the QMgr. Packetized data — with the HDLC option turned on — will be passed to HDLC coprocessor.

Figure 50. HSS/HDLC Access Overview



14.5 Functional Interface Description

The HSS component provides the following interfaces:

- ixHssAccPortInit
- ixHssAccLastErrorRetrievalInitiate
- ixHssAccInit

- ixHssAccPktPortConnect
- ixHssAccPktPortEnable
- ixHssAccPktPortDisable
- ixHssAccPktPortDisconnect
- ixHssAccPktPortIsDisconnectComplete
- ixHssAccPktPortRxFreeReplenish
- ixHssAccPktPortTx
- ixHssAccChanConnect
- ixHssAccChanPortEnable
- ixHssAccChanPortDisable
- ixHssAccChanDisconnect
- ixHssAccChanStatusQuery
- ixHssAccShow
- ixHssAccStatsInit

The preceding interfaces are used to:

- Configure HSS ports
- Connect/disconnect to or from the packetized service on an HDLC-port basis
- Connect/disconnect to or from the channelized service on an HSS-port basis
- Enable and disable the receive service for both packetized and channelized service
- Send and receive buffers
- Manipulate the outstanding free buffers queue for a particular HDLC port
- Replenish the free queue
- Register call-backs when the free queue crosses a particular threshold
- Set the queue threshold and call-back source

These functions are described in the following sections.

14.6 Key Assumptions

The HSS service is predicated on the following assumptions:

- Two HSS ports will be supported.
- Packetized (HDLC) service is coupled with the HSS port.

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

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

14.7 Programming Procedures

Before IxHssAcc can be used, the client has to:

1. Download NPE_A code to NPE_A.
2. Start the NPE_A.
3. Initialize the IxNpeMh.
4. Initialize QMgr.

After initialization, the client will call *ixHssAccPortInit*() to configure HSS ports. This will configure each time slot in a frame to provide either packetized or channelized service as well as other characteristics of the HSS port.

Next, the clients prepare data buffers to exchange data with the HSS component, for transmitting or receiving. Depending on whether it is channelized or packetized service, the data is exchanged differently, as described in "Initialization and Configuration" on page 156.

The client then calls the *ixHssAccPktPortConnect*() or *ixHssAccChanConnect*() to connect the client to the IxHssAcc service. Additionally, the service provides callback functions for the service to inform the client when data is received and ready to delivered to the client.

The HSS component will be in operation after it is enabled.

Callback functions or a pool mechanism is used in the transmitting and receiving process. The client will process the received data or provide new data for transmission. This is done by providing new buffer pointers or by adjusting the existing pointers.

Finally, when the HSS component is no longer needed, ixHssAccPktPortDisable() and ixHssAccPktPortDisconnect() — or ixHssAccChanDisconnect() and ixHssAccChanPortDisable() — are called.

14.8 Initialization and Configuration

Before IxHssAcc is enabled, the client has to initialize the NPE and configure the HSS port for packetized/channelized access.

The HSS ports must be configured to match the configuration of any connected PHYs. This includes configuring the time slots within a frame in one of the following ways:

- Configuring as HDLC — For packetized service
- Configuring as Voice64K/Voice56K — For channelized service
- Configuring as unassigned — For unused time slot
- Choosing the line speed, frame size, signal polarities, signal levels, clock edge, endianness, choice of input/output frame signal, and other parameters

For initialization, the following functions must be called:

- ixNpeDlNpeInitAndStart

- ixNpeMhInitialize
- ixQMgrInit
- ixQMgrDispatcherLoopGet
- ixOsServIntBind

Note: The function calls — for downloading an NPE image and starting its execution — have been changed. The new function is ixNpeDlInitAndStart. The previous function calls used were the following:

— ixDemoUtilsNpeImageDownload

— ixNpeDlNpeExecutionStart

For the appropriate parameters to use for downloading the HSS images in the NPE image library, refer to the ixNpeDl section of this guide.

For HSS configuration, the client application calls function *ixHssAccPortInit*(). No channelized or packetized connections should exist in the HssAccess layer while this interface is being called.

This function takes the following arguments:

- IxHssAccHssPort hssPortId (in) — The HSS port ID.

  There are two identical ports:

  — IX_HSSACC_HSS_PORT_0 HSS

  — IX_HSSACC_HSS_PORT_1
- IxHssAccConfigParams *configParams (in) — A pointer to the HSS configuration structure
- IxHssAccTdmSlotUsage *tdmMap (in) — A pointer to an array defining the HSS time-slot assignment types
- IxHssAccLastErrorCallback lastHssErrorCallback (in) — Client callback to report the last error

The parameter IxHssAccConfigParams has two structures of type IxHssAccPortConfig — one for HSS Tx and one for HSS Rx. These structures are used to choose:

- Frame-synchronize the pulse type (Tx/Rx)
- Determine how the frame sync pulse is to be used (Tx/Rx)
- Frame-synchronize the clock edge type (Tx/Rx)
- Determine the data clock edge type (Tx/Rx)
- Determine the clock direction (Tx/Rx)
- Determine whether or not to use the frame sync pulse (Tx/Rx)
- Determine the data rate in relation to the clock (Tx/Rx)
- Determine the data polarity type (Tx/Rx)
- Determine the data endianness (Tx/Rx)
- Determine the Tx pin open drain mode (Tx)
- Determine the start of frame types (Tx/Rx)
- Determine the whether or not to drive the data pins (Tx)

- Determine the how to drive the data pins for voice56k type (Tx)
- Determine the how to drive the data pins for unassigned type (Tx)
- Determine the how to drive the Fbit (Tx)
- Set 56k data endianness, when using the 56k type (Tx)
- Set 56k data transmission type, when using the 56k type (Tx)
- Set the frame-pulse offset in bits w.r.t, for the first time slot (0-1,023) (Tx/Rx)
- Determine the frame size in bits (1-1,024)

IxHssAccConfigParams also has the following parameters:

- The number of channelized time slots (0 - 32)
- The number of packetized clients (0 - 4)
- The byte to be transmitted on channelized service, when there is no client data to Tx
- The HSS loop-back state
- The data to be transmitted on packetized service, when there is no client data to Tx
- The HSS clock speed

IxHssAccTdmSlotUsage is an array that take the following values to assign service types to each time slot in a HSS frame:

IX_HSSACC_TDMMAP_UNASSIGNED	Unassigned
IX_HSSACC_TDMMAP_HDLC HDLC	Packetized
IX_HSSACC_TDMMAP_VOICE56K Voice56K	Channelized
IX_HSSACC_TDMMAP_VOICE64K Voice64K	Channelized

IxHssAccTdmSlotUsage has a size equal to the number of time slots in a frame.

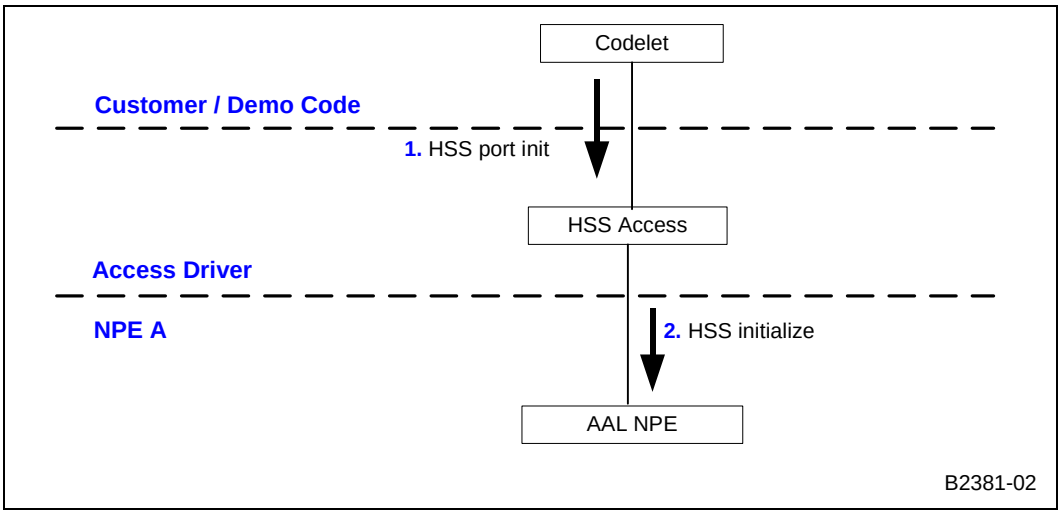
IxHssAccLastErrorCallback() is for error handling. The client will initiate the last error retrieval. The HssAccess component then sends a message to the NPE through the NPE Message Handler. When a response to the error retrieval is received, the NPE Message Handler will callback the HssAccess component, which will execute *IxHssAccLastErrorCallback*() in the same IxNpeMh context. The client will be passed the last error and the related service port.

In summary, initialization and configuration require calling the following functions:

- ixNpeDlNpeInitAndStart
- ixNpeMhInitialize
- ixQMgrInit
- ixQMgrDispatcherLoopGet
- ixOsServIntBind
- ixHssAccPortInit

After these functions are called, the HSS coprocessor will be running, although no access is given to the client until a connection occurs followed by an enable. Figure 51 shows the initialization process.

Figure 51. HssAccess Initialization



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 channelized time slots
   - A per-time-slot mapping of traffic type: HDLC, Voice, 56kVoice, or Unassigned
2. The IxHssAcc layer configures the NPE appropriately.
   When complete, the HSS coprocessor will be running, although no access is given to the client until a connect occurs followed by an enable.

14.9 Buffer Allocation Data-Flow Overview

Before service from HSS component is connected and enabled, buffers need to be allocated and provided to HSS component. The HSS component provides two services, packetized and channelized, and the HSS-component clients exchange data with the HSS component differently for transmitting and receiving, depending on what service is chosen.

14.9.1 Data Flow in Packetized Service

Data in the time slots configured as HDLC type will form packets for packetized service. There are four packetized services per HSS port. The packetized service uses mbufs to store data, or chains mbufs together into chained mbufs for large packets.

The client is responsible for allocating these buffers and passing the buffers to IxHssAcc.

An mbuf pool should be created for packetized service by calling function *IX_MBUF_POOL_INIT()* of the IxOsBuffMgt API with the mbuf size and number of mbuf needed. For example:

```
IxHssAccCodeletMbufPool **poolIdPtr;
unsigned numPoolMbufs;
unsigned poolMbufSize;
IX_STATUS status;
status = IX_MBUF_POOL_INIT(poolIdPtr, numPoolMbufs,
poolMbufSize, "HssAcc Codelet Pool");
```

A mbuf can be obtained from the pool by calling *IX_MBUF_POOL_GET*(). This mbuf pool is shared by the Tx and Rx processes.

For Rx, before the packetized service is enabled, the Rx mbuf queue in HSS component has to be replenished. This can be done by calling *ixHssAccPktPortRxFreeReplenish*().

When packetized service starts, it is the client's responsibility to ensure there is always an adequate supply of mbufs for the receive direction. This can be achieved in two ways. A call-back function can be registered with IxHssAcc to be called back when the free mbufs queue is running low. This call back function is registered with the HSS component packetized service when *ixHssAccPktPortConnect*() is called as described in the connection section below. Alternatively, the client can use its own timer for regular supply of mBufs to the queue.

The client also provides a receive call-back function to accept packets received through the HSS. After the data in the mbuf is processed, *IX_MBUF_POOL_PUT_CHAIN*() can be called to put the Rx mBuf back into the mbuf pool.

For Tx, mbufs are allocated from the mBuf pool by calling *IX_MBUF_POOL_GET*(). Data for transmitting can be put into the mbuf by using *IX_MBUF_MDATA*(). If the client data is too large to fit into one mbuf, multiple mbufs can be obtained from the pool and made into a chained mbuf by using *IX_MBUF_PKT_LEN*() and *IX_MBUF_NEXT_BUFFER_IN_PKT_PTR*(). The whole chained mbuf can be passed to HSS component for transmission by calling *ixHssAccPktPortTx*() as is described in more detail in packetized service Tx/Rx section.

A Tx callback function is also registered when *ixHssAccPktPortConnect*() is called before the service is enabled. When a chained mbuf is done with transmitting, the callback function is called and the mbuf can be returned to the mbuf pool. The data flow is showed in Figure 52 through Figure 62.

Figure 52. HSS Packetized Receive Buffering

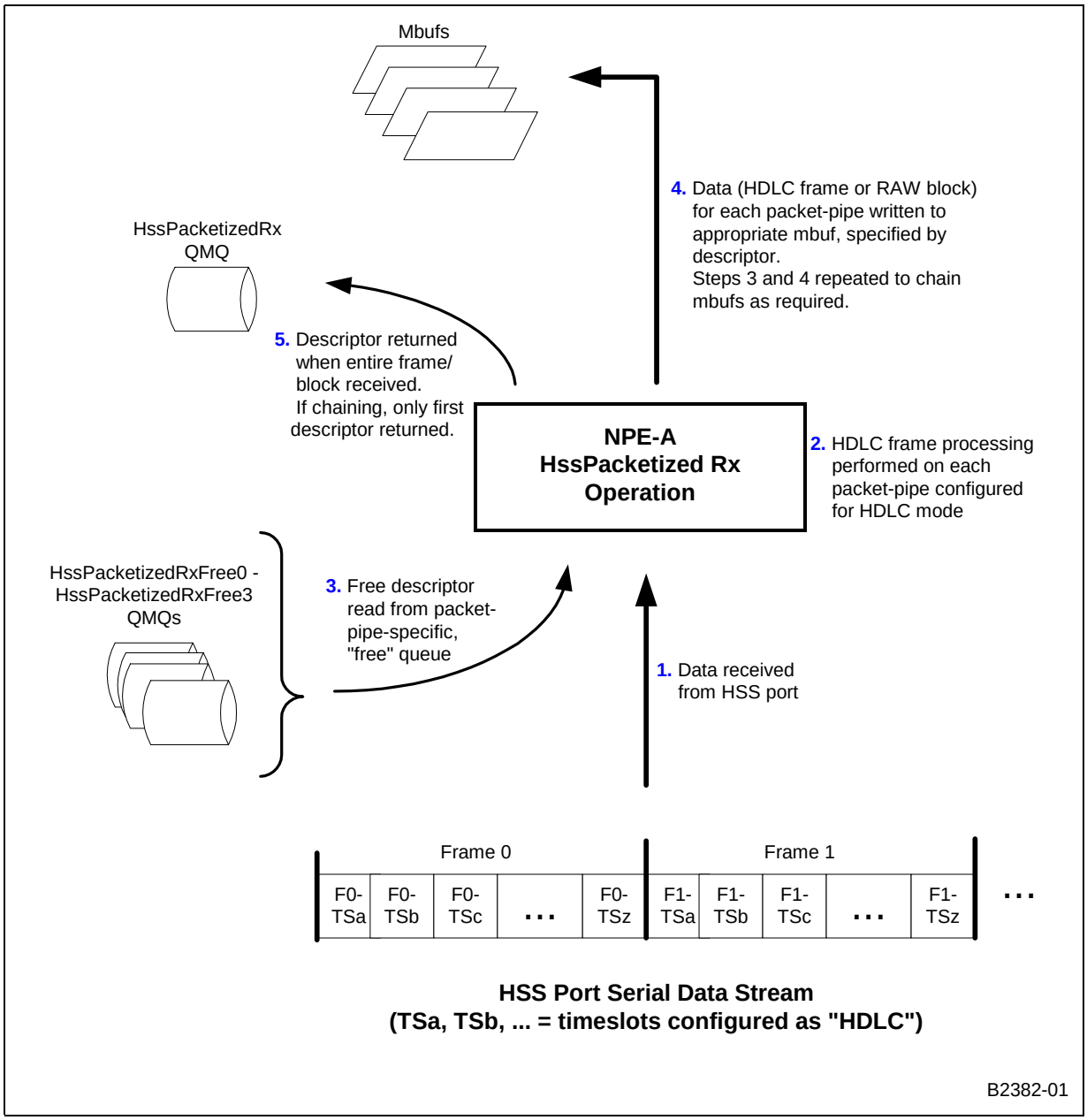
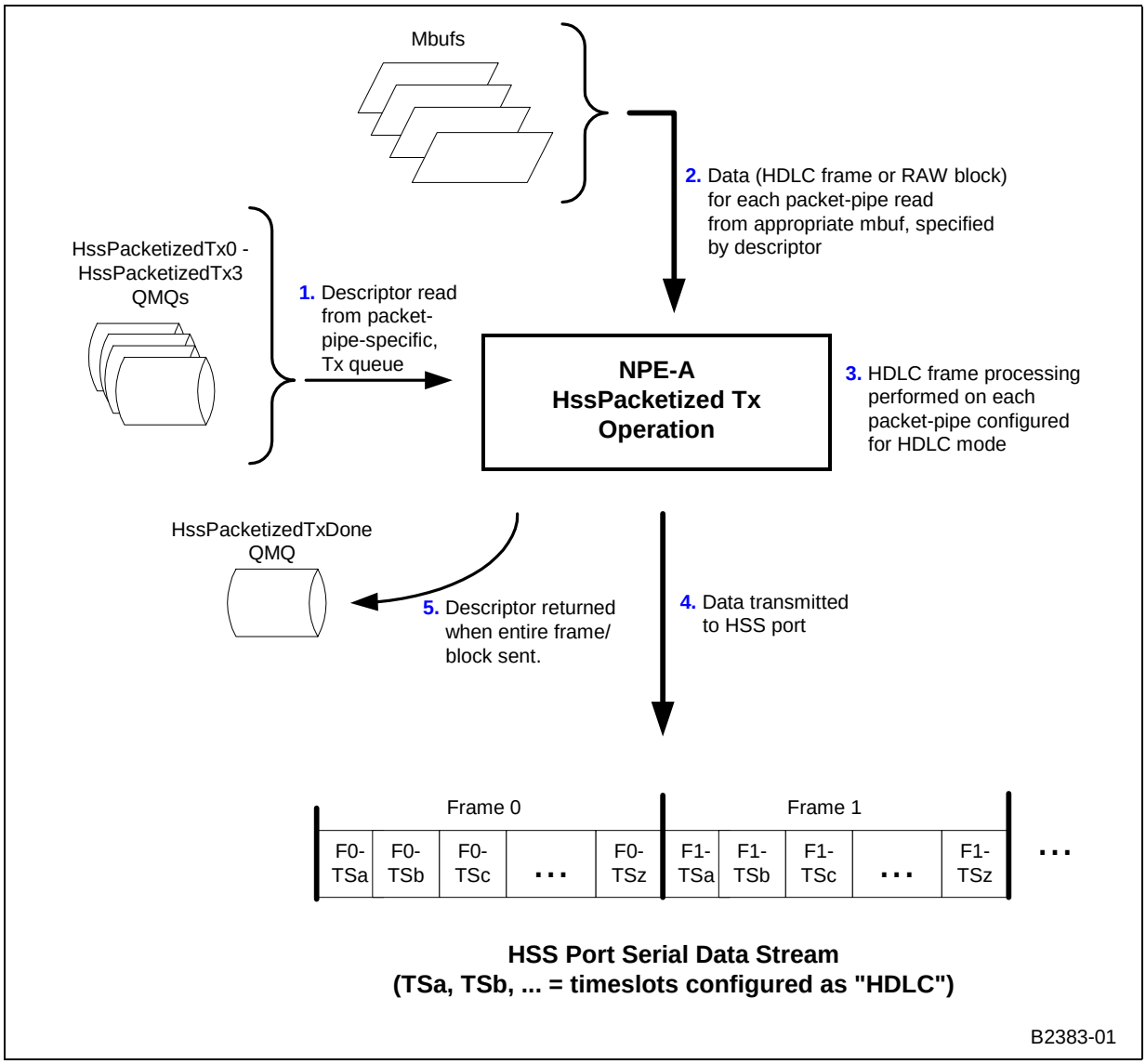


Figure 53. HSS Packetized Transmit Buffering



14.9.2 Data Flow in Channelized Service

Data in the time slots configured as Voice64K/Voice56K type will form channelized service. There are up to 32 such channels per HSS port. The channelized service uses memory that is shared between the Intel XScale core and the NPEs. The client is responsible for allocating the memory for HSS component to transmit and receive data through the HSS port.

For receive, *cacheDmaMalloc*() of the IxCacheMMU component can be used to create a pointer to a pool of contiguous memory from the shared memory of the Intel XScale core and the NPEs. The pointer to this Rx data pool needs to be a physical address because NPE will directly write data into this memory area. The memory pool is divided into *N* circular buffers, one buffer per channel. *N* is the total number of channels in service.

All the buffers have the same length. When the channelized service is initialized by *ixHssAccChanConnect*(), the pointer to the pool, the length of the circular buffers, and a parameter *bytesPerTStrigger* are passed to IxHssAcc, as well as a Rx call-back-function pointer.

Figure 54 shows how the circular buffers are filled with data received though the HSS. When each of the *N* channels receive *bytesPerTStrigger* bytes, the Rx callback function will be called, and an offset value *rxOffset* is returned to indicate where data is written into the circular buffer. Note that *rxOffset* is shared for all the circular buffers in the pool. *rxOffset* is adjusted internally in the HSS component so that it will be wrapped back to the beginning of the circular buffer when it reaches the end of the circular buffer.

The client has to make sure the Rx data is processed or moved elsewhere before being overwritten by the HSS component. Hence the length of the circular buffers has to be chosen properly. The buffer need to be large enough for data to be read by the client before the NPE rewrites over that memory. System latency is useful in making this calculation. If the circular buffer length is 3*44, for instance, this will give the client 3 * 5.5 ms of time for the 64-kbps data rate before received data is over-written.

For transmission, *cacheDmaMalloc*() is used to allocated two pools: a data buffer pool and a pointer list pool.

The data buffer pool has *N* buffers — one for each channel. Each buffer is divided into *K* sections and each section has *L* bytes.

The pointer list pool has *K* pointer lists. Each list has *N* pointers, each pointing to a section in a data buffer.

Before channelized service is enabled, the pointers have to be initialized to point to the first section of each data buffer in the data buffer pool, and data for transmission is prepared and moved to the data buffer. The pointers to the data buffer pool and pointer list pool are passed to IxHssAcc when *ixHssAccChanConnect*() is called. When Rx callback function is called, an offset value *txOffset* is returned.

txOffset indicates which pointer list in the pointer list pool is pointing to the sections of the data buffers currently being transmitted. Thus the client can use *txOffset* to determine where new data needs to be put into the data buffer pool for transmission. For example, data can be prepared and moved into sections pointed by the (*txOffset*-2)*th* pointer list. The length of the buffer, *K* * *L*, needs to be large enough so that the client has enough time to prepare data for transmission.

Figure 54. HSS Channelized Receive Operation

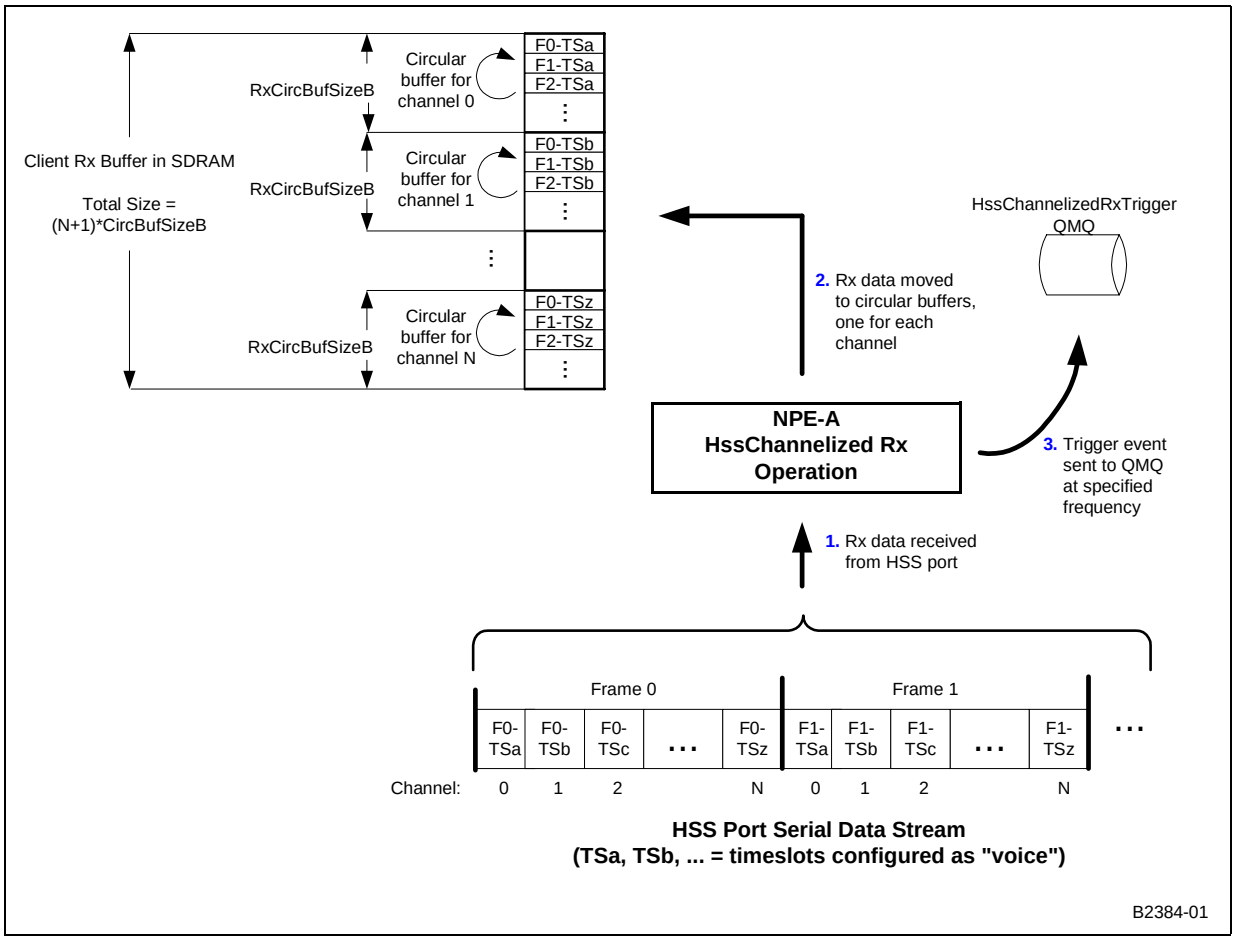
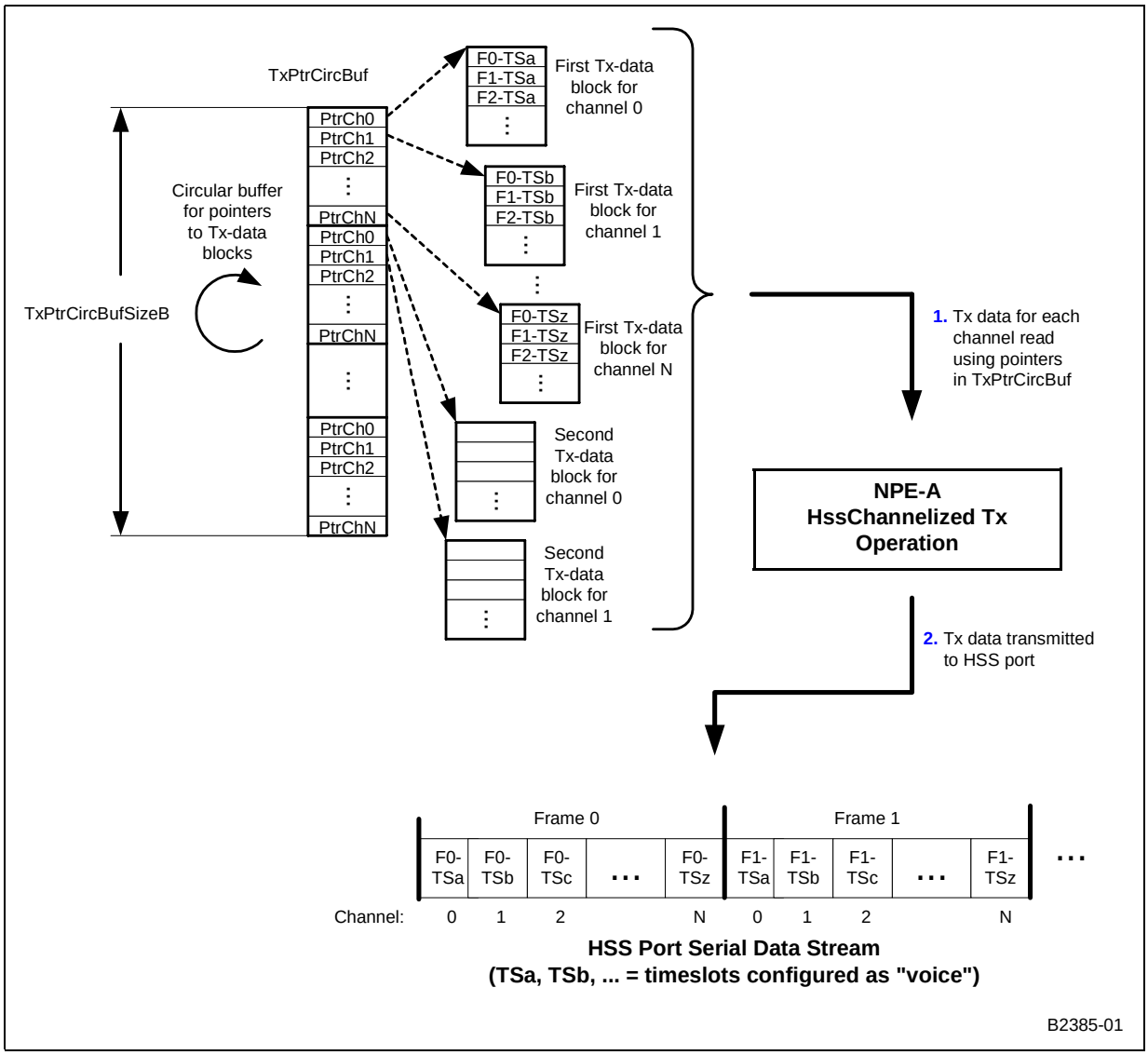


Figure 55. HSS Channelized Transmit Operation



14.10 HSS Channelized Operation

14.10.1 Channelized Connect and Enable

After the HSS component is configured, *ixHssAccChanConnect*() has to be called to connect the client application with the channelized service. This function is called once per HSS port, as there can only be one client per HSS port.

The client uses this function to:

- Register a Rx call-back function

- Set up how often this callback function will be called
- Pass the pointer to the Rx data circular buffer pool
- Set the size of the Rx circular buffers
- Set the pointer to the Tx pointer lists pool
- Set the size of the tx data buffers.

The parameters needed by *ixHssAccChanConnect*() include:

- IxHssAccHssPort hssPortId (in) — The HSS port ID.

  There are two identical ports (0-1).
- unsigned bytesPerTSTrigger (in) — The NPE will trigger the access component to call the Rx call back function *rxCallback*() after *bytesPerTSTrigger* bytes have been received for all trunk time slots.

  bytesPerTSTrigger is a multiple of eight. For example: 8 for 1-ms trigger, 16 for 2-ms trigger.
- UINT8 *rxCircular (in) — A pointer to the Rx data pool allocated by the client as described in previous section. It points to a set of circular buffers to be filled by the received data.

  This address will be written to by the NPE and must be a physical address.
- unsigned numRxBytesPerTS (in) — The length of each Rx circular buffer in the Rx data pool.
- UINT32 *txPtrList (in) — The pointer to the pointer lists pool described in previous section. It must be a pointer to a physical addresses.
- unsigned numTxPtrLists (in) — The number of pointer lists in txPtrList. This number 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 channelized data after *bytesPerTSTrigger* bytes have been received for all trunk time slots.

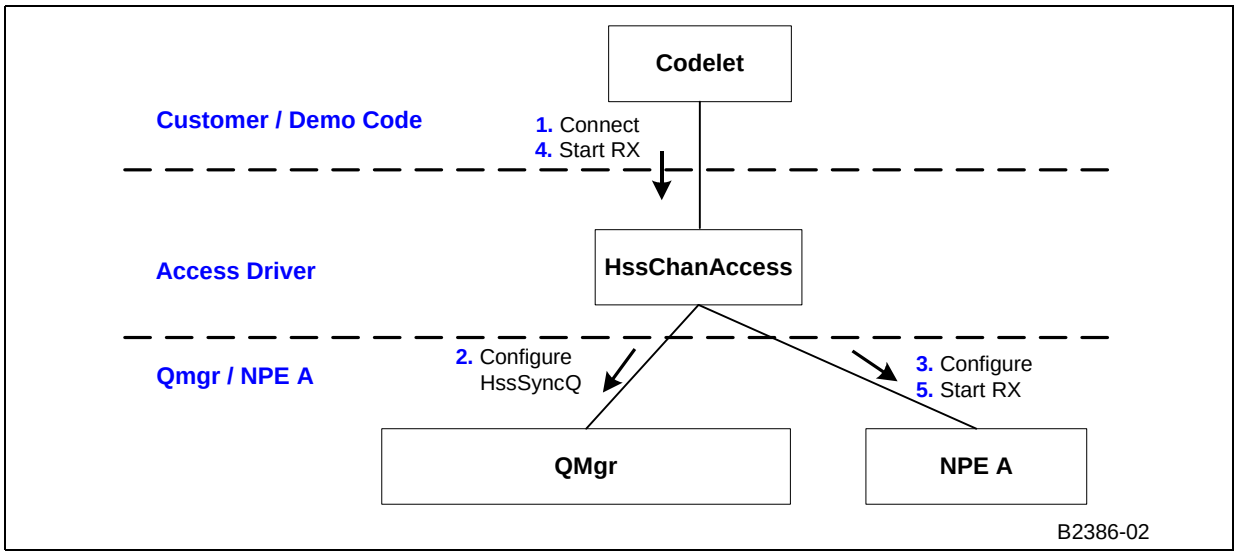
  If this is not NULL, an ISR will call this function.

  If this pointer is NULL, it implies that the client will use a polling mechanism to detect when the tx and rx of channelized data is to occur.

After the client application is connected with the channelized service, the HSS component then can be enabled by calling *ixHssAccChanPortEnable*() with the port ID provided to enable the channelized service from that particular HSS port.

The following figures show what is done in IxHssAcc when the channelized service connection function is called.

Figure 56. hsschannelizedAccess — Connect



1. The codelet issues a connect request to HssChanAccess.
2. If rxCallback, passed through the connect call, is not NULL, 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 IxHssAcc.
5. IxHssAcc enables the NPE Rx flow.

14.10.2 Channelized Tx/Rx

After being initialized, configured, connected, and enabled, the HSS component is up and running. There are two schemes to handle channelized service Tx/Rx process: callback and polled.

14.10.2.1 CallBack

If the pointer to the *rxCallback*() is not *NULL* when *ixHssAccChanConnect*() is called, an ISR will call *rxCallback*() to handle Tx/Rx data. It is called when each of *N* channels receives *bytesPerTStrigger* bytes.

Usually, a Rx thread is created to handle the HSS channelized service. The thread will be waiting for semaphore. When *rxCallback*() is called by IxHssAcc, *rxCallback*() will put the information from IxHssAcc into a structure, and send a semaphore to the thread. Then *rxCallback*() returns so that IxHssAcc can continue its own tasks.

The Rx thread — after receiving the semaphore — will wake up, take the parameters passed by *rxCallback*(), and perform Rx data processing, Tx data preparing, and error handling.

For Rx data processing, *rxCallback*() provides the offset value *rxOffset* to indicate where data is just written into each circular buffer. *rxOffset* is shared for all the circular buffers in the pool. The client has to make sure the Rx data are processed or moved to somewhere else before overwritten by the HSS component since the buffers are circular.

For TX data preparing, *rxCallback*() provide the offset value *txOffset* to indicate which pointer list in the pointer lists pool is pointing to the data buffers currently being or will be transmitted. As a result, the client can use *txOffset* to determine where new data needs to be put into the data buffer pool for transmission. For example, data can be prepared and moved into buffers pointed by the (*txOffset*-2)*th* pointer list.

rxCallback() also provides the number of errors NPE receives. The client can call function *ixHssAccLastErrorRetrievalInitiate*() to initiate the retrieval of the last HSS error.

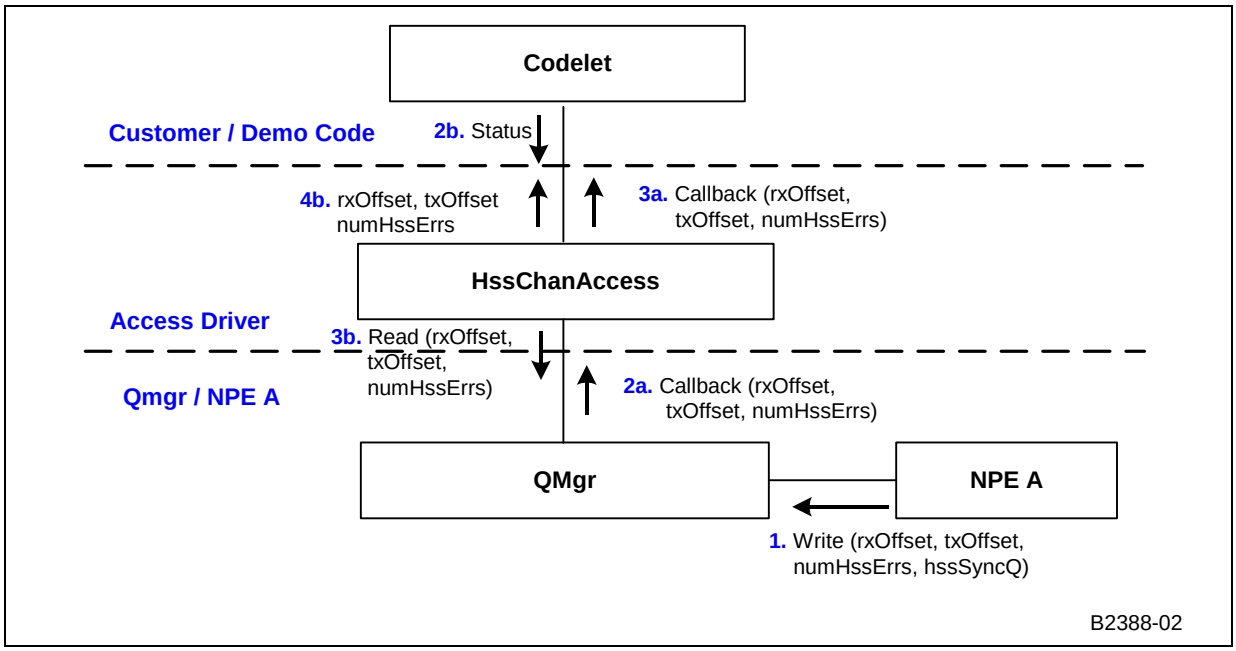
14.10.2.2 Polled

If the pointer to the *rxCallback*() is NULL when *ixHssAccChanConnect*() is called, it implies that the client will use a polling mechanism to detect when the Tx and Rx of channelized data is to occur. The client will use *ixHssAccChanStatusQuery*() to query whether channelized data has been received. If data has been received, IxHssAcc will return the details in the output parameters of *ixHssAccChanStatusQuery.*

ixHssAccChanStatusQuery() returns a flag *dataRecvd* that indicates whether the access component has any data for the client. If *FALSE*, the other output parameters will not have been written to. If it is *TRUE*, then *rxOffset*, *txOffset*, and *numHssErrs* — returned by *ixHssAccChanStatusQuery*() — are valid and can be used in the same way as in the callback function case above.

Figure 57 shows the Tx/Rx process.

Figure 57. hssChannelizedAccess — 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.

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

   b. HssChanAccess will call-back the codelet.

   — OR —

   a. The codelet will poll HssChanAccess.

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

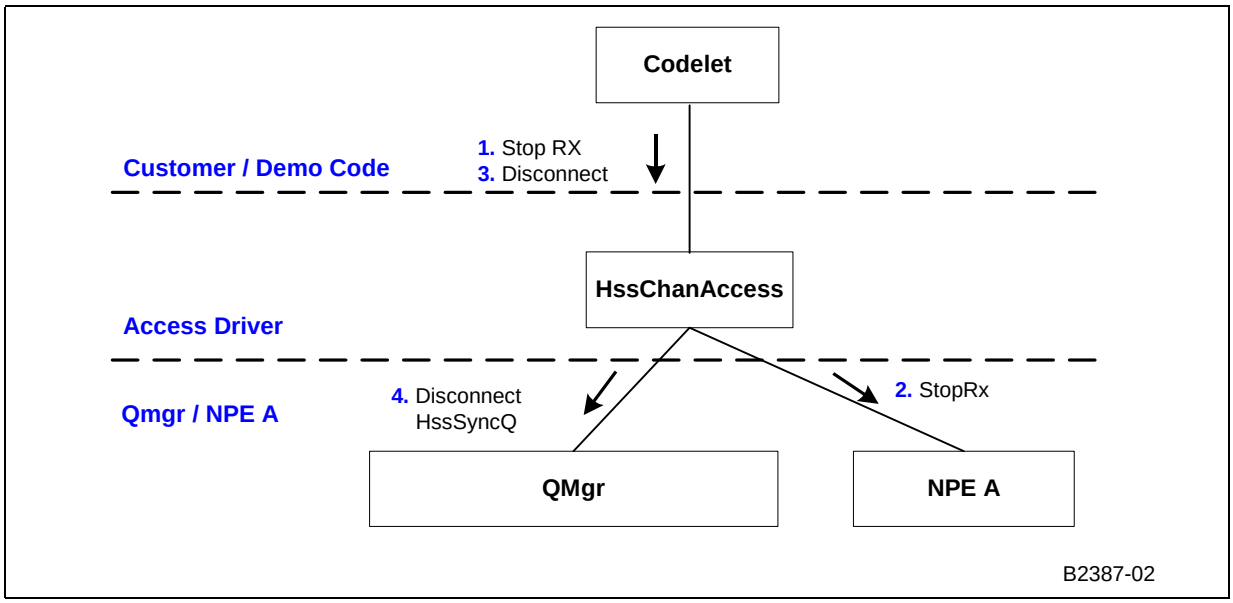
   c. If HssChanAccess read an entry from the hssSyncQ, it returns the details to the codelet.

14.10.3 Channelized Disconnect

When channelized service is not needed any more, ixHssAccChanPortDisable() is called to stop the channelized service, then ixHssAccChanDisconnect() is called to disconnect the service.

Figure 58 shows the disconnect process.

Figure 58. hsschannelizedAccess — 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 channelized access.
4. If a function pointer was registered to the QMgr for the trigger to transfer tx/rx blocks, it is de-registered.

14.11 HSS Packetized Operation

14.11.1 Packetized Connect and Enable

After the HSS component is configured, *ixHssAccPktPortConnect*() has to be called to connect the client application with the packetized services. There are four packetized services per HSS port, so this function has to be called once per packetized service.

The client uses this function to:

- Pass data structures to configure the HDLC coprocessor
- Register a Rx call back function for Rx data processing
- Register a callback function to request more Rx buffers
- Register a callback function to indicate Tx done
- Pass a flag to turn HDLC processing on or off

The HDLC configuration structure sets up:

- What to transmit when an HDLC port is idle
- HDLC data endianness
- CRC type to be used for this HDLC port.

The parameters for *ixHssAccPktPortConnect*() include:

- IxHssAccHssPort hssPortId (in) — The HSS port ID.
  There are two identical ports (0-1).
- IxHssAccHdlcPort hdlcPortId (in) — The number of the HDLC port and it corresponds to the physical E1/T1 trunk i.e. 0, 1, 2, 3
- BOOL hdlcFraming (in) — This value determines whether the service will use HDLC data or the debug raw data type, i.e. no HDLC processing
- unsigned blockSizeInWords (in) — The max 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 coprocessor for Tx:
  - 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
- IxHssAccHdlcFraming hdlcRxFraming (in) — The information required by the NPE to configure the HDLC coprocessor for Rx as hdlcTxFraming
- 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

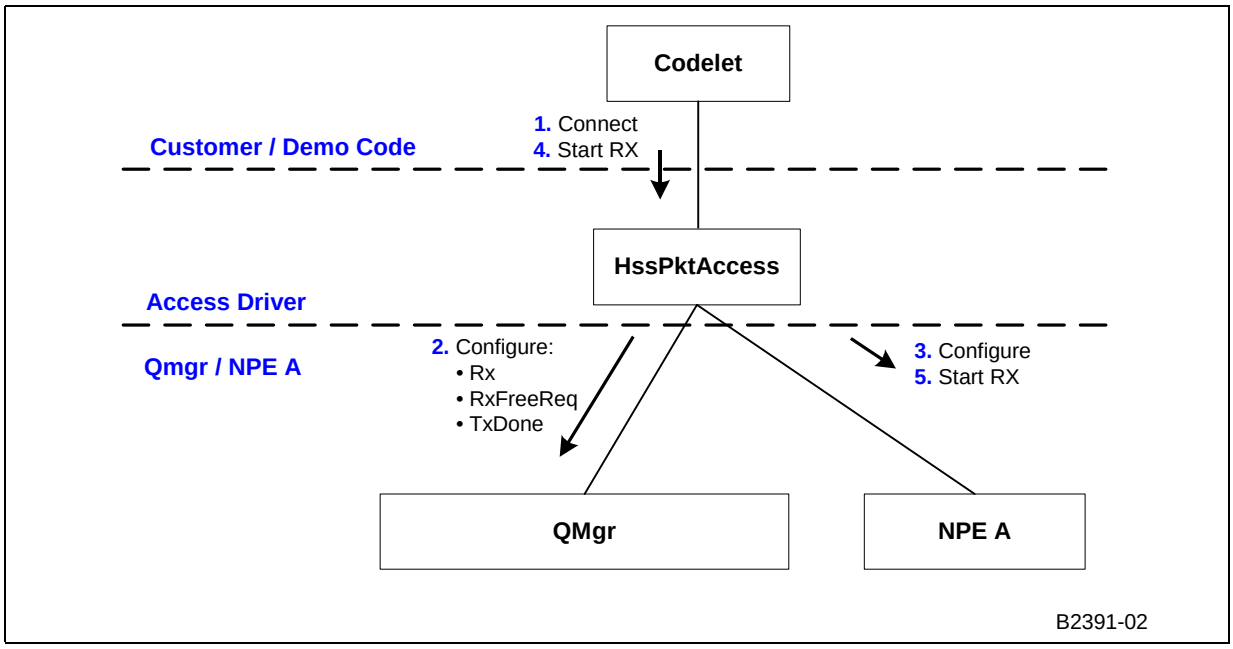
- IxHssAccPktRxFreeLowCallback rxFreeLowCallback (in) — Pointer to the clients Rx-free-buffer request function.

  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

Now the HSS component can be enabled by calling *ixHssAccPktPortEnable*() with the port ID provided to enable the packetized service from that particular HSS port after connect function is called.

Figure 59 show what is done in IxHssAcc when the packetized service connection function is called.

Figure 59. hsspacketizedAccess — Connect



1. The codelet issues a connect request to HssPktAccess.
2. HssPktAccess configures the QMgr queues and register callbacks to be 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.

14.11.2 Packetized Tx

After being initialized, configured, connected, and enabled, HSS component is up and running.

When the client has nothing to transmit, the HSS will transmit the idle pattern provided in the function *ixHssAccPktPortConnect*().

When the client has data for transmission, the client will call *IX_MBUF_POOL_GET*() to get a mbuf, put the date into the mBuf using *IX_MBUF_MDATA*(). If the client data is too large to fit into one mbuf. Multiple mBufs can be obtained from the pool, and put into a chained mBuf by using *IX_MBUF_PKT_LEN*() and *IX_MBUF_NEXT_BUFFER_IN_PKT_PTR*(). The whole chained mbuf is passed to IxHssAcc for transmission by calling *ixHssAccPktPortTx*().

When the transmission is done, the Tx done call back function, registered with *ixHssAccPktPortConnect*(), is called, and the mbuf can be returned to mbuf pool using *IX_MBUF_POOL_PUT_CHAIN*().

The following is example Tx code showing how to send an mbuf:

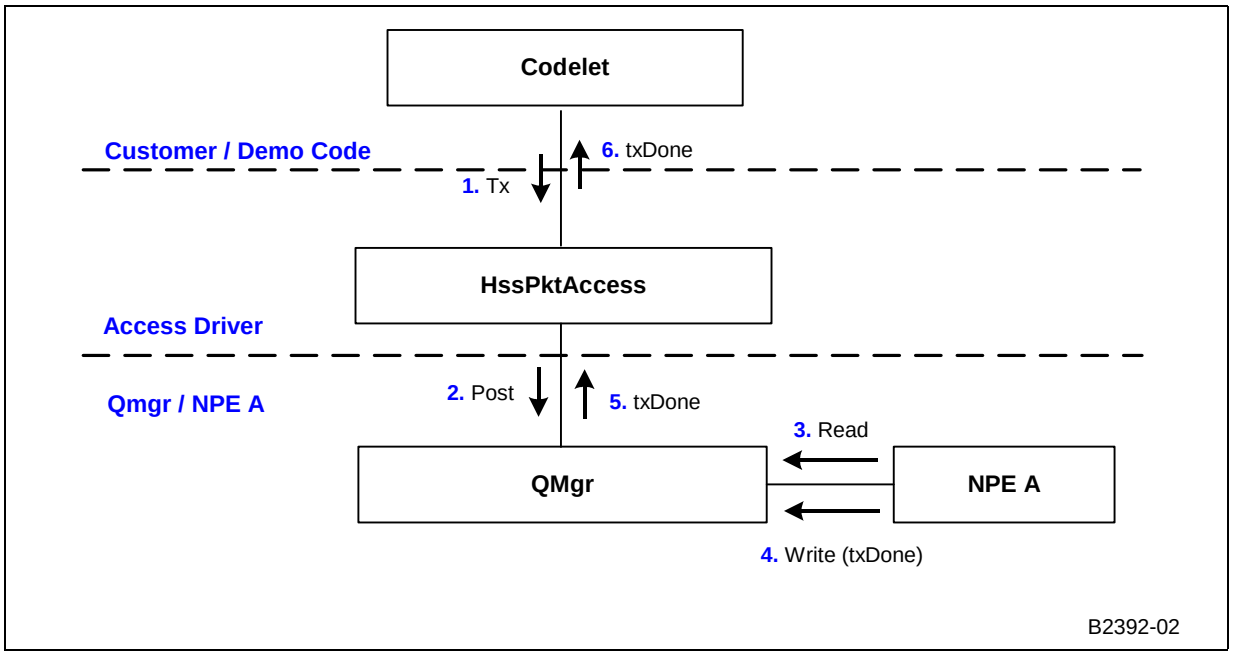
```
IX_MBUF *txBuffer;
IX_MBUF *txBufferChain = NULL;
// get a mbuf
IX_MBUF_POOL_GET(poolId, &txBuffer);
// set the data length in the buffer
IX_MBUF_MLEN(txBuffer) = NumberOfBytesToSend;
/* set the values to transmit */
for (byteIndex = 0; byteIndex < IX_MBUF_MLEN(txBuffer);
byteIndex++)
((UINT8 *)IX_MBUF_MDATA(txBuffer))[byteIndex]
=userData[byteIndex];
//send the buffer out
ixHssAccPktPortTx (hssPortId, hdlcPortId, txBuffer);
```

The following is example Tx code showing how to chain mbufs together:

```
IX_MBUF *txBufferChain = NULL;
   IX_MBUF *lastBuffer = NULL;
   if (txBufferChain == NULL)
   {  // the first buffer
      txBufferChain = txBuffer;
      /* set packet header for buffer */
      IX_MBUF_PKT_LEN(txBufferChain) = 0;
   }
   else
   {  // following buffers
      IX_MBUF_NEXT_BUFFER_IN_PKT_PTR(lastBuffer) = txBuffer;
   }
   // update the chain length
   IX_MBUF_PKT_LEN(txBufferChain) += IX_MBUF_MLEN(txBuffer);
   IX_MBUF_NEXT_BUFFER_IN_PKT_PTR(txBuffer) = NULL;
   lastBuffer = txBuffer;
   // send the bubber out
   ixHssAccPktPortTx (hssPortId, hdlcPortId, txBufferChain);
```

The process is shown in Figure 60.

Figure 60. hsspacketizedAccess — Tx



1. The codelet presents an 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 callback is executed. The descriptor is freed internally.
6. HssPktAccess calls back the codelet, passing it back its mbuf pointer.

14.11.3 Packetized Rx

Before packetized service is enabled, the Rx mbuf queue — in the HSS component — has to be replenished. This can be done by calling *IX_MBUF_POOL_GET*() to get a mbuf and calling *ixHssAccPktPortRxFreeReplenish*() to put the mBuf into the queue. This is repeated until the queue is full.

Here is an example:

```
// get a buffer
IX_MBUF_POOL_GET(poolId, &rxBuffer);
//IxHssAcc component needs to know the capacity of the mbuf
IX_MBUF_MLEN(rxBuffer) = IX_HSSACC_CODELET_PKT_BUFSIZE;
// give the Rx buffer to the HssAcc component
status = ixHssAccPktPortRxFreeReplenish (hssPortId,
hdlcPortId, rxBuffer);
```

Usually, an Rx thread is created to handle the HSS packetized service, namely, to handle all the callback functions registered with *ixHssAccPktPortConnect*(). The thread will be waiting for a semaphore. When any one of call back functions is executed by the HSS component, it will put the information from IxHssAcc into a structure, and send a semaphore to the thread. Then the call back function returns so that IxHssAcc can continues its own tasks. The Rx thread, after receiving the semaphore, will wake up, take the parameters from the structure passed by the call back function, and perform Rx data processing, Tx data preparing, and error handling.

When data is received, *rxCallback*() is called. It passes the received data in the form of a mbuf to the client. The mbuf passed back to the client could contain a chain of mbufs, depending on the packet size received. *IX_MBUF_NEXT_BUFFER_IN_PKT_PTR*() can be used to get access to each of the mbufs in the chained mbuf, and *IX_MBUF_MDATA*() can be used to get access to each data value. The mbuf is returned to mbuf pool by using *IX_MBUF_POOL_PUT_CHAIN*().

Here is an example:

```
IX_MBUF *buffer,
IX_MBUF *rxBuffer;
// go through each mbuf in the chained mbuf
for (rxBuffer = buffer;
   (rxBuffer != NULL) && (pktStatus == IX_HSSACC_PKT_OK);
   rxBuffer = IX_MBUF_NEXT_BUFFER_IN_PKT_PTR(rxBuffer))
for (wordIndex =0;wordIndex<(IX_MBUF_MLEN(rxBuffer) / 4);
   wordIndex++)
{  // get the values in the mbuf
value = ((UINT32 *)IX_MBUF_MDATA(rxBuffer))[wordIndex];
}
// free the chained mbuf
IX_MBUF_POOL_PUT_CHAIN(buffer);
```

rxCallback() also passes the packet status and the number of errors that NPE receives. The packet status is used to determine if the packet received is good or bad, and the client can call function *ixHssAccLastErrorRetrievalInitiate*() to initiate the retrieval of the last HSS error.

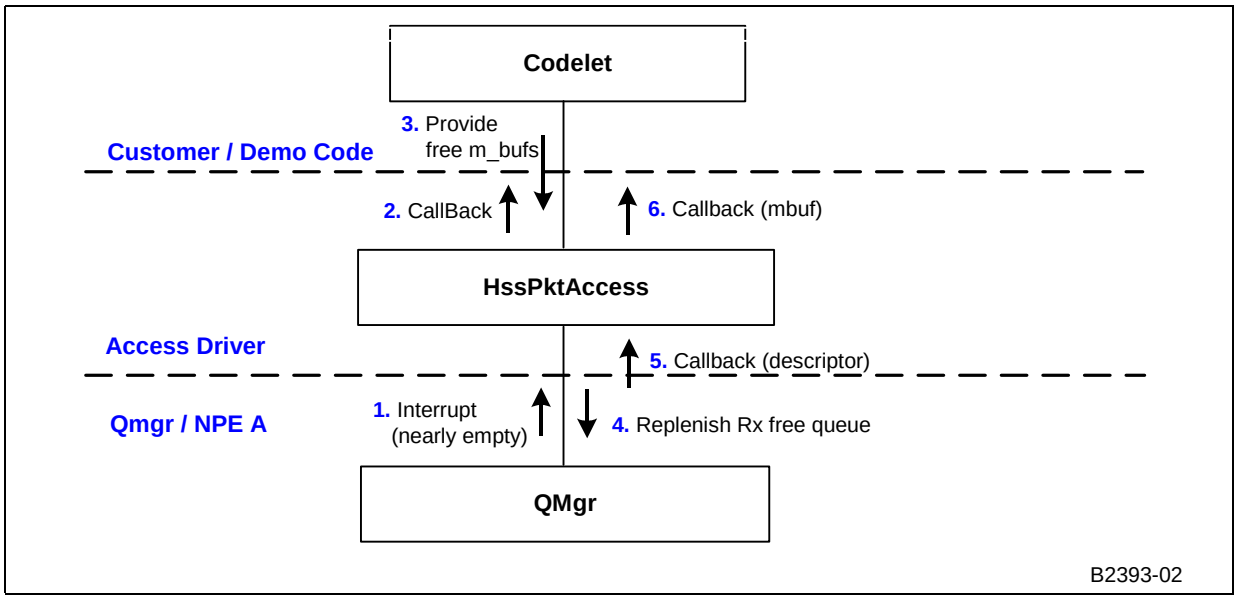
When the Rx mbuf queue is running low, *rxFreeLowCallback*() is called. Then, the client can call *X_MBUF_POOL_GET*() and *ixHssAccPktPortRxFreeReplenish*() to fill up the Rx queue again.

Alternatively, the client can use its own timer for regular supply of mbufs to the queue. This is the case if the pointer for *rxFreeLowCallback*() passed to *ixHssAccPktPortConnect*() is *NULL*.

When HSS component finishes transmitting data, *IxHssAccPktTxDoneCallback*() is called. It passes the pointer of the Tx mbuf back to the client. The client can return the Tx mbuf back to mbuf pool by calling *IX_MBUF_POOL_PUT_CHAIN*(). *IxHssAccPktTxDoneCallback*() also passes the packet status and the number of errors that NPE receives. The packet status is used to determine if the packet is successfully sent out or not, and the client can call function *ixHssAccLastErrorRetrievalInitiate*() to initiate the retrieval of the last HSS error.

The process is show in Figure 61.

Figure 61. hsspacketizedAccess — HDLC Rx (Call-Back Implementation)



1. The codelet registers a function call-back on connection, to be triggered through an ISR when the Rx FREE queue is low.
2. The call-back function requests more mbuf.
3. The codelet provides free mbufs for specific HDLC ports.
4. Free mbufs are stored in the Rx FREE queue, within the HssPktAccess Rx descriptors.
5. 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.
6. The mbuf is transmitted to the codelet.

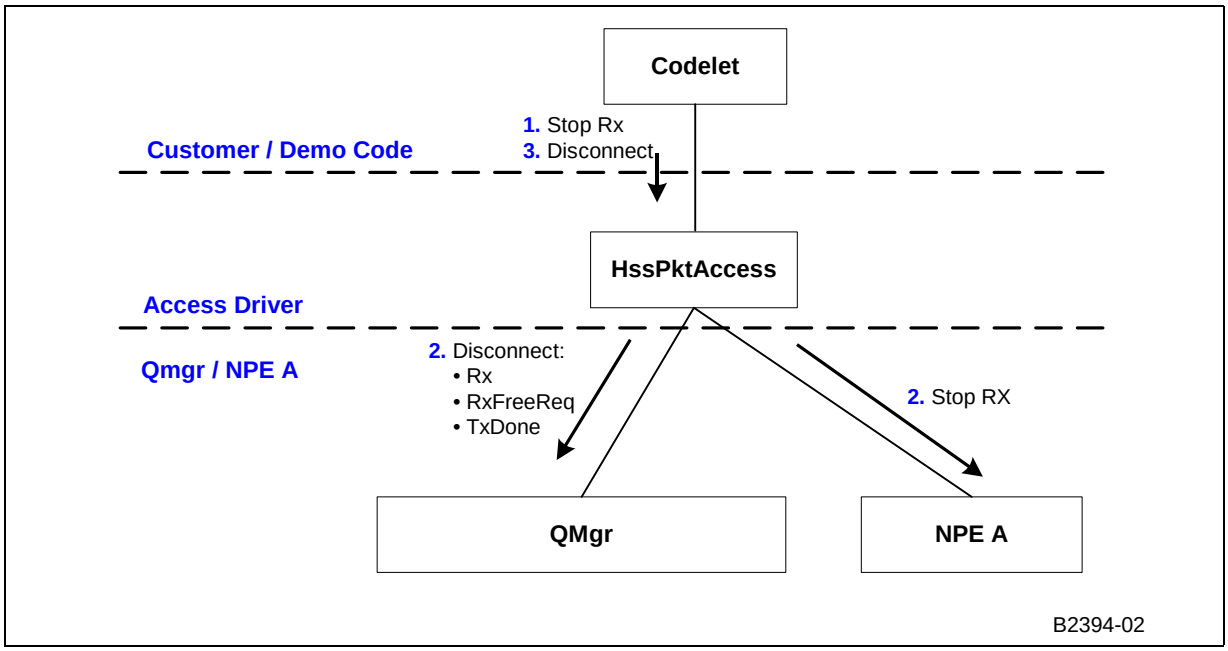
14.11.4 Packetized Disconnect

When packetized service is not needed any more, the function ixHssAccPktPortDisable() is called to stop the packetized service, and ixHssAccPktPortDisconnect() is called to disconnect the service.

This has to be done for each service. The client is responsible for ensuring all transmit activity ceases prior to disconnecting, and ensuring that the replenishment of the rxFreeQMQ ceases before trying to disconnect.

Figure 62 shows the disconnect process.

Figure 62. hsspacketizedAccess — 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.

Access-Layer Components: NPE-Downloader (IxNpeDl) API 15

This chapter describes the Intel® IXP400 Software v.1.3's "NPE-Downloader API" access-layer component.

15.1 What's New

Two new functions have been written to facilitate simple and efficient NPE initialization. They are replacing the two- to four-step process — required in software releases 1.2.2 and earlier — for loading an NPE and beginning execution.

- ixNpeDlInitAndStart — Starts execution on an NPE using a specified standard image.
- ixNpeDlCustomImageInitAndStart — Starts execution on a NPE using a custom image.

Note: The functions previously used for this task have become deprecated. See "Note About Deprecated APIs" on page 179 and refer to the ixNpeDl API for any additional information regarding the deprecated functions.

15.2 Overview

The NPE downloader (ixNpeDI) component is a stand-alone component providing a facility to download a microcode firmware image to NPE-A, NPE-B, or NPE-C in the system. The IxNpeDI component contains the default library of NPE microcode images, which contains up-to-date microcode for each NPE.

The IxNpeDl component also enables a client to supply a custom microcode image to use in place of the default images for each NPE. This "custom image" facility provides increased testability and flexibility, but is not intended for general use.

15.3 Microcode Images

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

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

15.4 Standard Usage Example

The initialization of an NPE has been made relatively easy. Only one function call is required.

Users call the *ixNpeDlNpeInitAndStart* function, which loads a specified image and begins execution on the NPE. Here is a sample function call, which starts NPE B with Ethernet and fast-path functionality:

```
ixNpeDlNpeInitAndStart(IX_NPEDL_NPEIMAGE_NPEB_ETH_FPATH);
```

The parameter is a UINT32 that is defined in the IXP425 NPE image ID definition. Table 35 lists the parameters for the standard images.

Table 35. NPE Images

Image Name	Description
IX_NPEDL_NPEIMAGE_NPEA_HSS0	NPE Image Id for NPE-A with HSS-0 Only feature. It supports 32 channelized and 4 packetized.
X_NPEDL_NPEIMAGE_NPEA_HSS0_ATM_SPHY_1_PORT	NPE Image Id for NPE-A with HSS-0 and ATM feature. For HSS, it supports 16/32 channelized and 4/0 packetized. For ATM, it supports AAL5, AAL0 and OAM for UTOPIA SPHY, 1 logical port, 32 VCs. It also has Fast Path support.
IX_NPEDL_NPEIMAGE_NPEA_HSS0_ATM_MPHY_1_PORT	NPE Image Id for NPE-A with HSS-0 and ATM feature. For HSS, it supports 16/32 channelized and 4/0 packetized. For ATM, it supports AAL5, AAL0 and OAM for UTOPIA MPHY, 1 logical port, 32 VCs. It also has Fast Path support.
IX_NPEDL_NPEIMAGE_NPEA_ATM_MPHY_12_PORT	NPE Image Id for NPE-A with ATM-Only feature. It supports AAL5, AAL0 and OAM for UTOPIA MPHY, 12 logical ports, 32 VCs. It also has Fast Path support.
IX_NPEDL_NPEIMAGE_NPEA_HSS_2_PORT	NPE Image Id for NPE-A with HSS-0 and HSS-1 feature. Each HSS port supports 32 channelized and 4 packetized.
IX_NPEDL_NPEIMAGE_NPEA_DMA	NPE Image Id for NPE-A with DMA-Only feature.
IX_NPEDL_NPEIMAGE_NPEB_ETH	NPE Image Id for NPE-B with Ethernet-Only feature.
IX_NPEDL_NPEIMAGE_NPEB_ETH_FPATH	NPE Image Id for NPE-B with Ethernet and Fast Path feature.
IX_NPEDL_NPEIMAGE_NPEB_DMA	NPE Image Id for NPE-B with DMA-Only feature.
IX_NPEDL_NPEIMAGE_NPEC_ETH	NPE Image Id for NPE-C with Eth-Only feature.
IX_NPEDL_NPEIMAGE_NPEC_CRYPTO	NPE Image Id for NPE-C with Crypto-Only feature. For Crypto, it supports DES, SHA-1, MD5.
IX_NPEDL_NPEIMAGE_NPEC_CRYPTO_AES	NPE Image Id for NPE-C with Crypto-Only feature. For Crypto, it supports AES, DES, SHA-1, MD5.
IX_NPEDL_NPEIMAGE_NPEC_CRYPTO_ETH	NPE Image Id for NPE-C with Crypto and Eth feature. For Crypto, it supports DES, SHA-1, MD5.
IX_NPEDL_NPEIMAGE_NPEC_CRYPTO_AES_ETH	NPE Image Id for NPE-C with Crypto and Eth feature. For Crypto, it supports AES, DES, SHA-1, MD5.
IX_NPEDL_NPEIMAGE_NPEC_DMA	NPE Image Id for NPE-C with DMA-Only feature.

15.5 Custom Usage Example

Using a custom image is the second option for starting an NPE. This allows the use of an external library of images, if needed.

Here is the function used for this procedure:

```
ixNpeDlCustomImageNpeInitAndStart(UINT32 *imagelibrary, UINT32 npeImageId);
```

15.6 IxNpeDl Uninitialization

After the first NPE has been started using one of the above methods, IxNpeDl will be initialized and the specified NPEs will begin execution.

The IxNpeDl should be uninitialized prior to unloading a kernel module. (This will unmap all memory that has been mapped by IxNpeDI.) If possible, IxNpeDl also should be uninitialized before a soft reboot.

Here is a sample function call to uninitialize IxNpeDl:

```
ixNpeDlUnload();
```

Note: IxNpeDl can only be initialized from an uninitialized state, and can only be uninitialized from an initialized state. If this order is not followed — for example, by uninitializing an uninitialized IxNpeDl — unpredictable behavior will result. Calling any other IxNpeDl API functions after unloading will also cause unpredictable results.

15.7 Note About Deprecated APIs

As of this software release, the functions ixNpeDlNpeInitAndStart and ixNpeDlCustomImageNpeInitAndStart have replaced the following functions, which should not be used in any new development because they will be removed in a future release:

- ixNpeDlImageDownload
- ixNpeDlAvailableImagesCountGet
- ixNpeDlAvailableImagesListGet
- ixNpeDlLatestImageGet
- ixNpeDlLoadedImageGet
- ixNpeDlMicrocodeImageLibraryOverride

This page intentionally left blank.

Access-Layer Components: NPE Message Handler (IxNpeMh) API 16

This chapter describes the Intel® IXP400 Software v.1.3's "NPE Message Handler API" access-layer component.

16.1 Overview

This chapter contains the necessary steps to start the NPE message-handler component. Additionally, information has been included about how the Message Handler functions from a high-level view.

This component acts a pseudo service layer to other access components such as EthAcc. In the sections that describe how the messaging works, the "client" is an access component such as EthAcc. An application programmer will not need to do any coding to directly control message handling, just the initialization and uninitialization of the component.

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. The solicited messages for a given ID are received in the same order that those soliciting messages are sent, and the first ID-matching callback in the list always corresponds to the next solicited message that is received.

16.2 Initializing the IxNpeMh

The IxNpeMh has two modes of operation, interrupted or polled. This refers to how the IxNpeMh will receive messages from the NPEs. When an NPE has a message for the message handler, it will always send an interrupt to the IxNpeMh, but the IxNpeMh must be set up for interrupt driven operation for it to service the interrupt automatically.

16.2.1 Interrupt-Driven Operation

This is the preferred method of operation for the message handler. Here is a sample function call to initialize the IxNpeMh component for interrupt driven operation:

```
ixNpeMhInitialize (IX_NPEMH_NPEINTERRUPTS_YES);
```

The function takes a yes/no value from an enum, and now all messages from all the NPEs will be serviced by IxNpeMH. The IxNpeMh handles messages from all NPEs and should only be initialized once.

16.2.2 Polled Operation

Here is a sample function call to initialize the IxNpeMh component for interrupt driven operation:

```
ixNpeMhInitialize (IX_NPEMH_NPEINTERRUPTS_NO);
```

The function takes a yes/no value from an enum, and now all messages from the NPEs must be manually checked. The IxNpeMh handles messages from all NPEs, and should only be initialized once.

After setting up polled operation the client must check for messages coming out of the NPEs. Here is a sample function call that will check to see if NPE-A has a message to send:

```
ixNpeMhMessagesReceive (IX_NPEMH_NPEID_NPEA);
```

Three separate function calls are required to check all three of the NPEs.

Note: This function call cannot be made for inside an interrupt service routine as it will use resource protection mechanisms.

16.3 Uninitializing IxNpeMh

The IxNpeMh should be uninitialized prior to unloading a kernel module (this will unmap all memory that has been mapped by IxNpeMh). If possible, IxNpeMh should also be uninitialized before a soft reboot.

Here is a sample function call to uninitialize IxNpeMh:

```
ixNpeMhUnload();
```

Note: IxNpeMh can only be initialized from an uninitialized state and can only be uninitialized from an initialized state. If this order is not followed, for example by uninitializing an uninitialized IxNpeMh, then unpredictable behavior will result. Calling any other IxNpeMh API functions after unloading will also cause unpredictable results.

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

Access-layer components — such as ixEthAcc and ixHssAcc — do all of their own message handling. This section describes the process of how messages are sent and processed so someone who is using IxNpeMh can understand what is going on in the background and gain insight into some performance issues.

There are two types of messages to send to an NPE: unsolicited and solicited. The first is just a simple message — that is, all it does is send a block of data. The second type sends data, but also registers a function to handle a response from the NPE.

The following sections give an overview of the process.

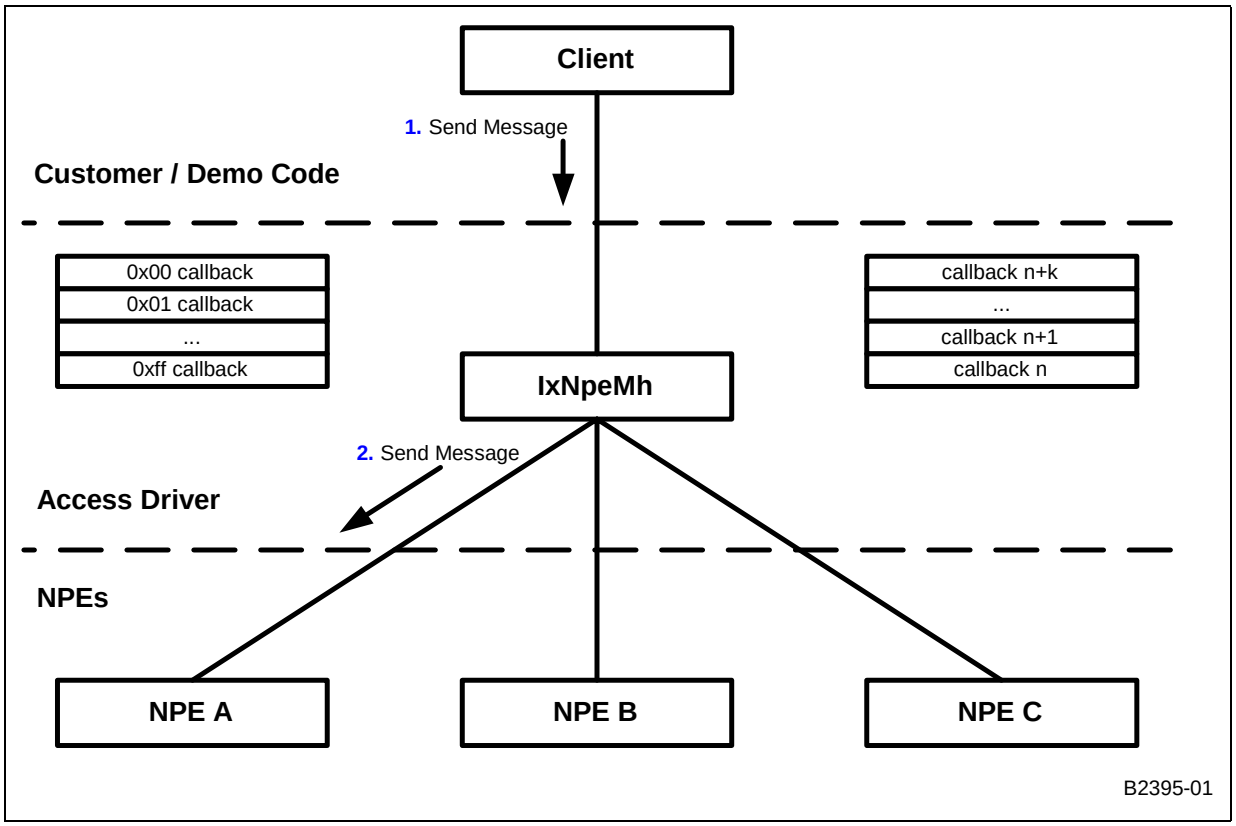
16.4.1 Sending an NPE Message

The scenario of sending a messages from an Intel XScale core software client to an NPE (as shown in Figure 63) is:

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

Note: If an NPE is busy, the message can be resent before the fail is returned. Because the action of rapidly messaging the NPE will consume the AHB bandwidth, the number of times the message will be sent is passed as a parameter to the send function; the default value is 3 (two retries).

Figure 63. Message from Intel XScale® Core Software Client to an NPE



16.4.2 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 64) is:

1. The client sends a message to the IxNpeMh component, specifying the destination NPE and a response callback.
2. The IxNpeMh component checks that the NPE can accept a message.

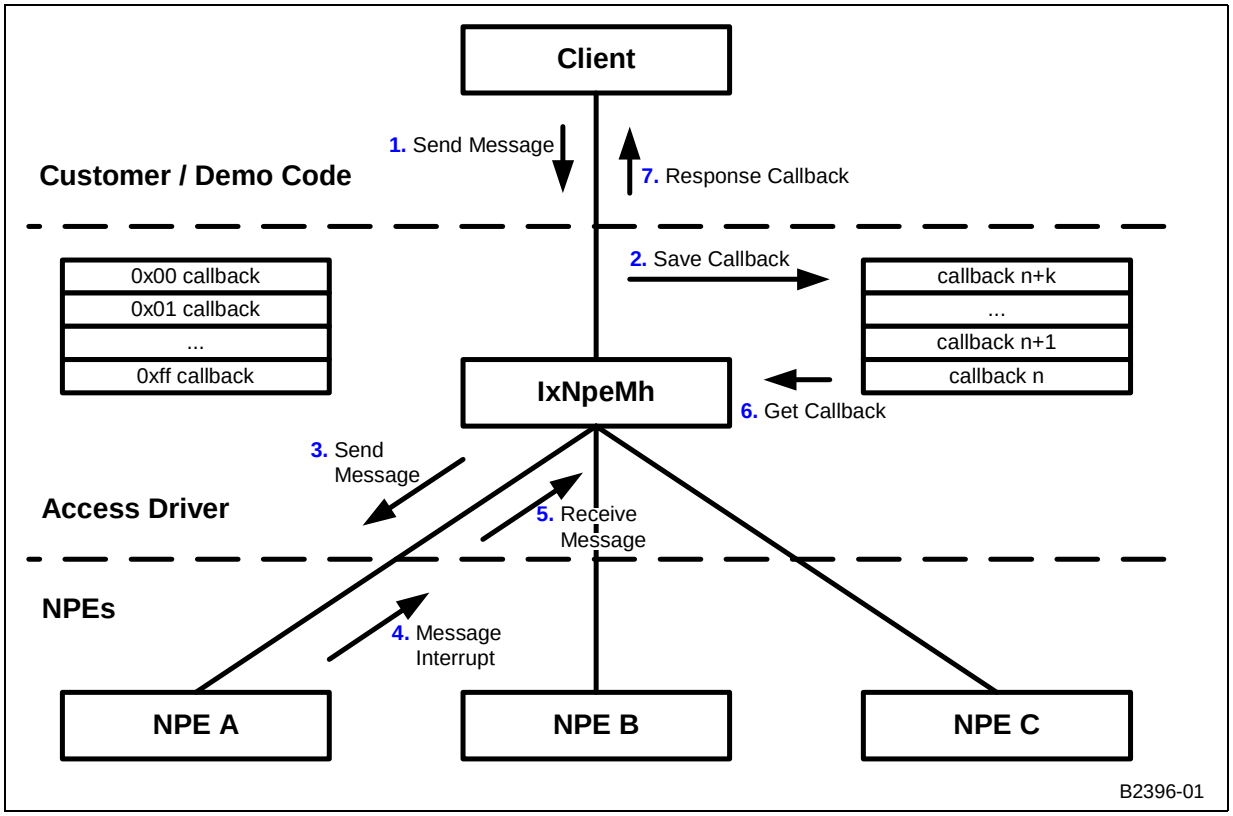
   If the component cannot accept a message, the send fails.
3. The IxNpeMh component adds the response callback to the end of the solicited callback list and sends the message to the NPE.
4. After some time, the NPEs "outFIFO not empty" interrupt invokes the IxNpeMh component's ISR.
5. Within the ISR, the IxNpeMh component receives a message from the specific NPE.
6. 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 185.)

7. 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 64. Message with Response from Intel XScale® Core Software Client to an NPE



16.5 Receiving Unsolicited Messages from an NPE to Client Software

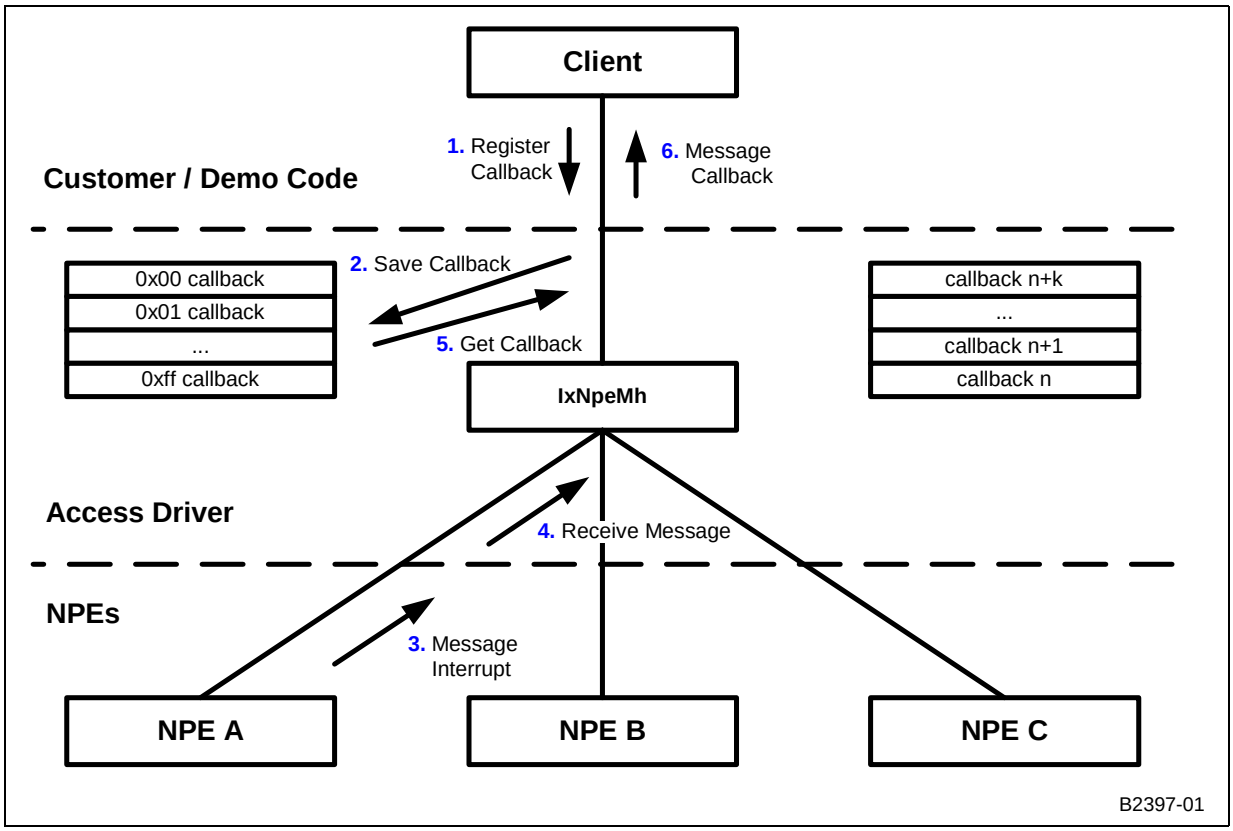
The scenario of receiving unsolicited messages from an NPE to client software (as shown in Figure 65) is:

1. At initialization, the client registers an unsolicited callback for a particular NPE and a message ID.
2. After some time, the NPEs “outFIFO not empty” interrupt invokes the IxNpeMh component’s ISR.
3. Within the ISR, the IxNpeMh component receives a message from the specific NPE.
4. 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 184.)

5. Since this is an unsolicited message, the IxNpeMh component invokes the corresponding unsolicited callback to pass the message back to the client.

Figure 65. Receiving Unsolicited Messages from NPE to Software Client



The IxNpeMh component does not interpret message IDs. It only uses message IDs for comparative purposes, and for passing 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.)

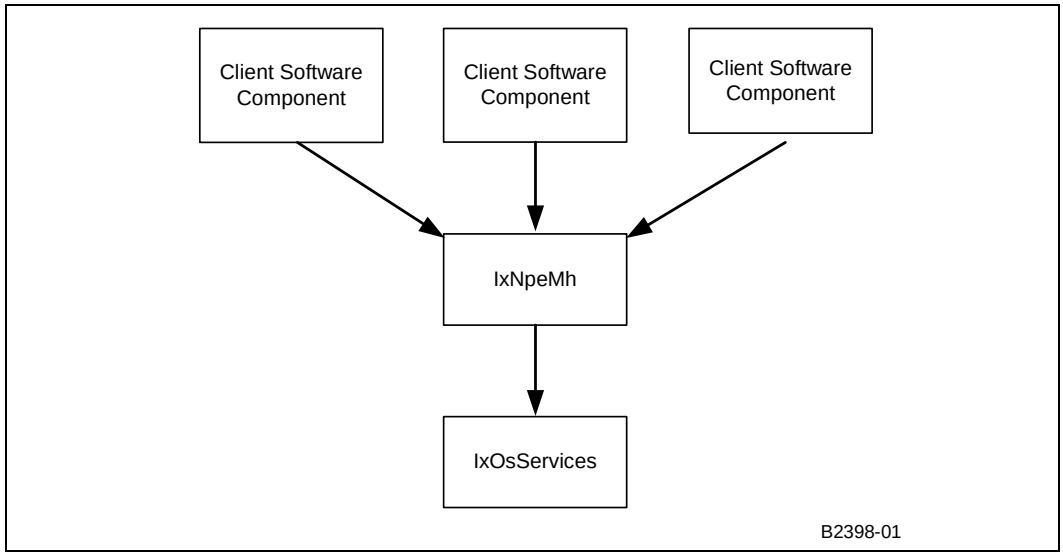
The IxNpeMh component handles messaging for the three NPEs independently. A problem or delay in interacting with one NPE will not impact interaction with the other NPEs.

16.6 Dependencies

The IxNpeMh component's dependencies (as shown in Figure 66) 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 66. ixNpeMh Component Dependencies



16.7 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 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.

This page intentionally left blank.

Access-Layer Components: Performance Profiling (IxPerfProfAcc) API 17

This chapter describes the Intel® IXP400 Software v.1.3's "Performance Profiling API" access-layer component.

17.1 Overview

The PerfProf Access module (IxPerfProfAcc) provides client access to the available performance statistics from the Intel® XScale™ core's PMU and the Internal Bus PMU as well as Xcycle, the idle-cycle counter utilities. These PMUs consist of programmable event counters, event select registers, and previous master/slave registers. Each counter is associated with an event by programming the event select registers.

The different features (Intel XScale core PMU, Bus PMU, and Xcycle) are not to be run at the same time as the PMU-enabling software may use a significant portion of the resources. In addition, the PMU-enabling software runs as an interrupt service routine while Xcycle disables interrupt during startup.

Utilizing only one PMU at a time will minimize the impact of the PerfProf Access module. Furthermore, the specific tasks for each PMU are not to be run in parallel. The PerfProf access layer component reads the registers for counter values, does the relevant calculations and presents the results to the client. All event and clock counters managed by the PerfProf access module are split into two, 32-bit-wide counters, to represent the upper and lower 32 bits of the count.

The access layer component itself will not contain any printf functions. Errors will be handled through error logging, but will store the calculated values in pointers to the output parameters — which can be accessed by the calling client.

Note: This component only work for big endian. It is planned that a future release will support little endian.

17.2 Intel XScale® Core PMU

The purpose of the Intel XScale core PMU is to enable performance measurement and to allow the client to identify the "hot spots" of a program. These hot spots are the sections of a program that consume the most number of cycles or cause process stalls due to events like cache misses, branches, and branch mispredictions.

The Intel XScale core PMU capabilities include clock counting, event counting, time-based sampling, and event-based sampling. A profiling period is defined as the length of time throughout which counting or sampling is done for a section of code. The results of this period are a profile summary.

Clock counting is used to measure the execution time of a program. The execution time of a block of code is measured by counting the number of processor clock cycles taken.

Event counting will be used to measure the number of specified performance events that occur in the system during the profiling period. The events monitored by the Intel XScale core's PMU are:

- Instruction cache miss requires fetch from external memory
- Instruction cache cannot deliver an instruction

  This could indicate an ICache miss or an ITLB miss. This event will occur every cycle in which the condition is present
- Stall due to a data dependency. This event will occur every cycle in which the condition is present
- Instruction TLB miss
- Data TLB miss
- Branch instruction executed, branch may or may not have changed program flow
- Branch mispredicted (B and BL instructions only)
- Instruction executed
- Stall because the data cache buffers are full (This event will occur every cycle in which the condition is present.)
- Stall because the data cache buffers are full (This event will occur once for each contiguous sequence of this type of stall.)
- Data cache access, not including cache operations
- Data cache miss, not including cache operations
- Data cache write-back (This event occurs once for each 1/2 line (four words) that are written back from the cache.)
- Software changed the PC

  This event occurs any time the PC is changed by software and there is not a mode change. For example, a MOV instruction with PC as the destination will trigger this event. Executing a SWI from Client mode will not trigger this event, because it will incur a mode change.

Time-based sampling is used to identify the most frequently executed lines of code for the client to focus performance analysis on. In this method, the sampling rate is the number of processor clock counts before a counter overflow interrupt is generated, at which a sample is taken. This sampling rate is defined by the client. The number of occurrences of each PC value determines the frequency with which the Intel XScale core's code is being executed.

Event-based sampling will allow the client to identify the "hot spots" of the program for further optimization. In this method, the sampling rate is the number of events before a counter overflow interrupt is generated. This sampling rate is defined by the client. As in time-based sampling, the PC value of each sample and frequency will be determined. This allows the client to identify the sections of code that cause each event.

Time-based sampling and event-based sampling, and event counting must not be performed concurrently. The client should be aware of the data memory required to perform each of these operations.

Event-based sampling allows the client to sample up to four events at a time. The maximum data memory required to store the results, in the event that the client chooses to perform event-based sampling with four events simultaneously, is about 4 Mbytes, and about 1 Mbytes for time-based sampling. In the event of an overflow in the results buffer, the client will be notified.

The PerfProf module provides the client with APIs to start and stop the collections of events. It will provide an API that reads and stores the value of all the counters. It will also enable the client to measure the latency (in clock cycles) between any two Intel XScale core instructions in a program.

Furthermore, the module will allow the client to determine the frequency with which Intel XScale core code is being executed.

17.2.1 Counter Buffer Overflow

The PerfProf module will allow the client to count up to four different events simultaneously and will also handle the overflow of these counters. In the case of overflow, the module will need to register an interrupt service routine. However, the handling of overflow will have a minimal impact on the running system.

The program shall keep track of the number of times a buffer has over flowed. The necessary adjustments will then be made to the final count value, to ensure an accurate value.

17.3 Internal Bus PMU

The internal bus PMU enables performance management of components accessing or utilizing the north and south bus. This includes statistics of the bus itself.

The counters monitor two types of events, which are occurrence events and duration events. The occurrence event causes the counter to increase by one, each time the event occurs. For duration events, the counter counts the number of clocks during which a particular condition or a set of conditions is true.

This PMU is able to monitor and gather statistics on SDRAM, north bus, south bus, north masters, north slaves, south masters, south slaves, and miscellaneous items like the cycle count. Among the details being monitored are:

- North bus usage — The north bus occupancy reported by the PMU.

  This is done by taking a snapshot of the total cycle count and subtracting the idle time.
- South bus usage — The south bus occupancy reported by the PMU.

  This is done by taking a snapshot of the total cycle count and subtracting the idle time.
- SDRAM controller usage — Usage monitored in all 8 pages of the SDRAM, i.e. the pages used and how often they are used.

  This also includes percentage usage and number of hits per second.
- SDRAM controller miss percentage — Identifies number of misses and rate of misses when accessing the SDRAM. A high miss rate would indicate a slow system.
- Previous Master Slave — Identifies the last master and slave on the respective buses.

This module has a Start API that obtains the register values at regular intervals. It only stops when a Stop API is called. User gets the desired results from the Get API.

17.4 Idle-Cycle Counter Utilities (Abbreviated to Xcycle)

Xcycle calculates the percentage of cycles that have been idle (not performing any processing) for a period of time.

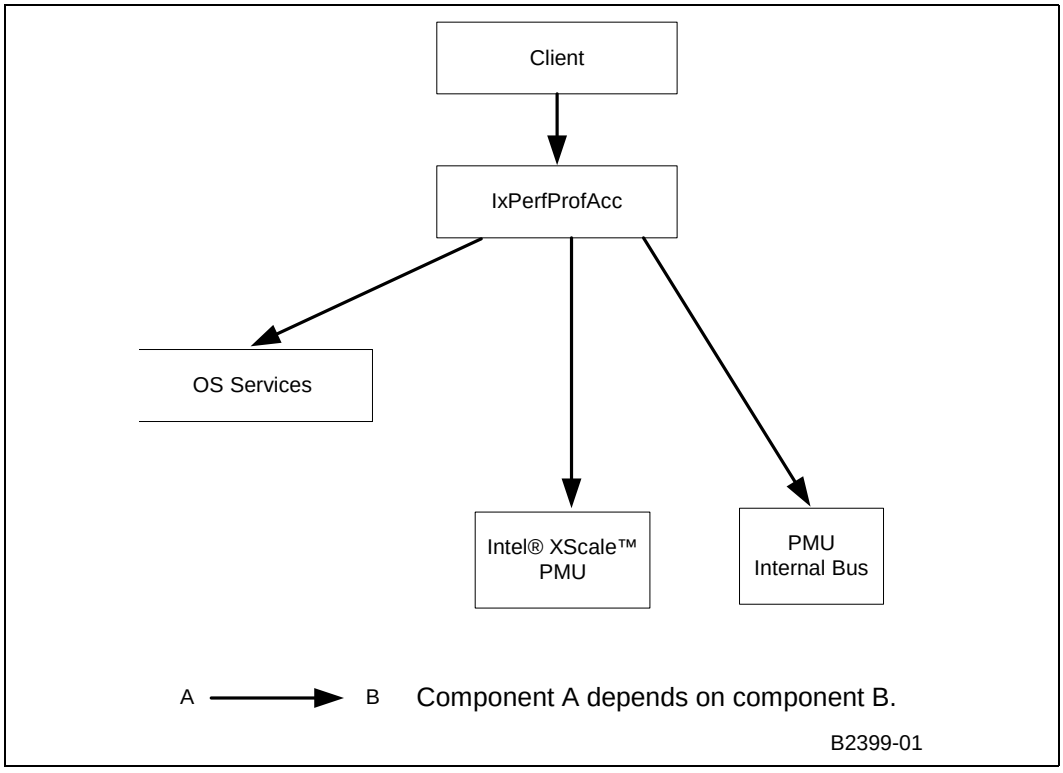
The client needs to calibrate the program by running *ixPerfProfAccXcycleBaselineRun*() when system is under low utilization. The client then starts the program he/she wants to measure. The *ixPerfProfAccXcycleStart*() API kicks off the idle cycle measurements. The client can select continuous Xcycle calculations, in which case calculations are stopped by calling the *ixPerfProfAccXcycleStop*(). Otherwise, the Xcycle measurements will occur for the number of times specified and will stop automatically.

The *ixPerfProfAccXcycleResultsGet*() API will calculate and prepare all the results to be sent to the calling function. The result contains maximum percentage of idle cycles, minimum percentage of idle cycles, average percentage of idle cycles, and total number of measurement made.

17.5 Dependencies

Figure 67 shows the functional dependencies of the IxPerfProfAcc component.

Figure 67. IxPerfProfAcc Dependencies



The client will call IxPerfProfAcc to access specific performance statistics of the Intel XScale core's PMU and internal bus PMU.

IxPerfProfAcc depends on the OS Services component for error handling and reporting, and for timer services like timestamp measurements.

17.6 Error Handling

IxPerfProfAcc returns an error type to the client and the client is expected to handle the error. Internal errors will be reported using the IxPerfProfAcc specific error handling mechanism as listed in IxPerfProfAccStatus. The Access Layer component will only return success or fail errors to its client. Any errors within the Access Layer will be logged and output to the screen using existing mechanisms.

17.7 Interrupt Handling

Both the PMUs generate interrupts when accessing the counters to obtain data. The Xcycle component on the other hand, disables the IRQ and FIQ during its calibration of the baseline. Any other components requiring interrupts during these periods may be affected.

17.8 Threading

The Xcycle component spawns a new task to work in the background. This task is spawned with the lowest priority. This is to avoid pre-emptying other tasks from running.

This task registers a dummy function that also triggers the measurement of idle cycles. The importance of starting a new thread at a low priority is that the task needs to run in the background whilst not preventing any other task from running, This is very important in obtaining the most accurate results.

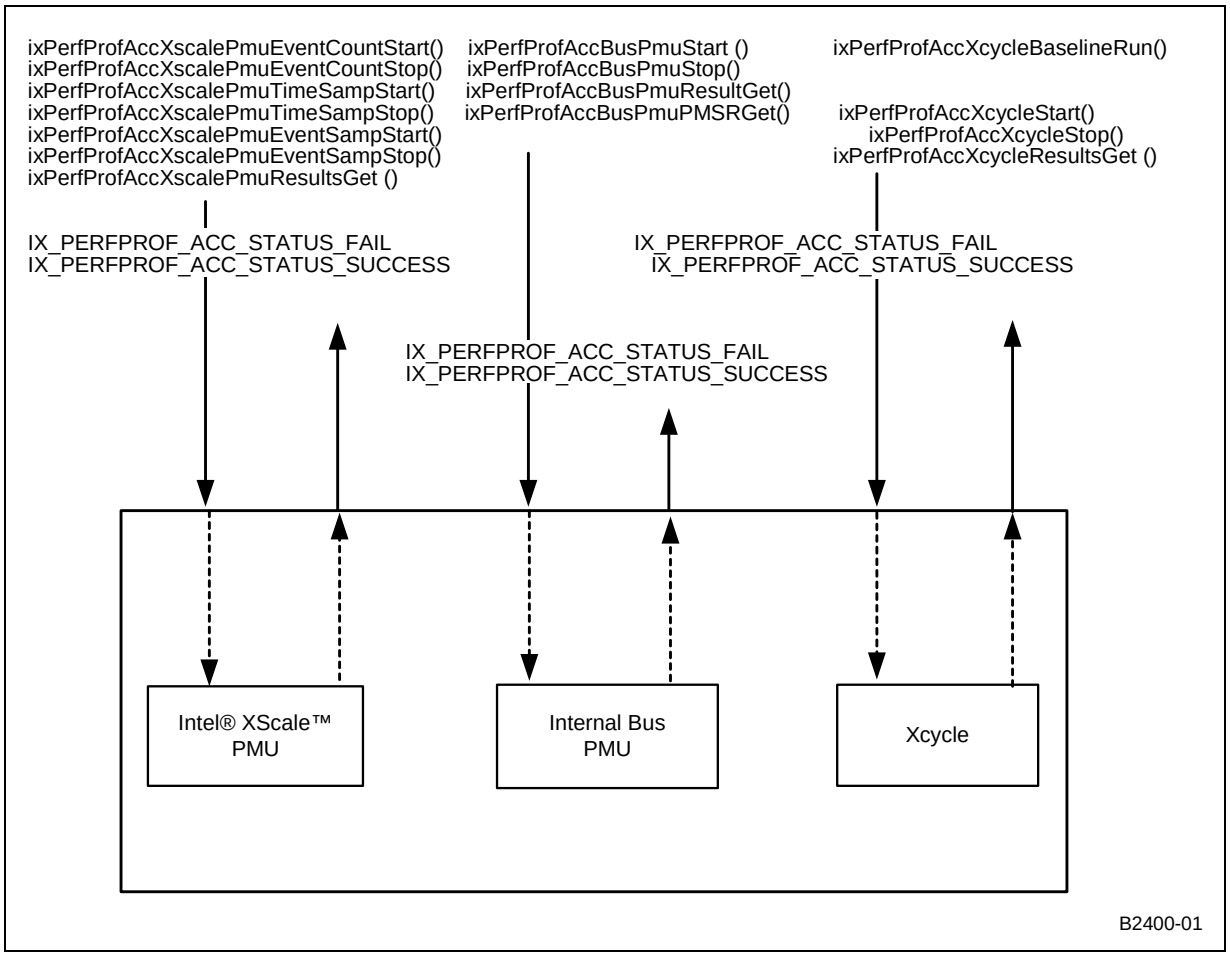
17.9 Using the API

This section will explain how to use the three utilities that make up the Performance Profiling Utilities component. It will also give practical usage examples of these utilities.

The examples provided here merely serve as a guide for the user. Users may choose to implement these utilities through their own methods.

Figure 68 shows all the APIs.

Figure 68. PerfProf Access Layer Component



17.9.1 API Usage for Intel XScale® Core PMU

The Intel XScale core's PMU utility provides three different capabilities, namely, event/clock counting, time-based sampling, and event-based sampling. The user may monitor their code/ program in two ways:

- From the CLI, call the appropriate Intel XScale core's PMU utility
- In the user's code itself, insert the appropriate Intel XScale core's PMU utility's start and stop functions.

17.9.1.1 Event and Clock Counting

This utility can be used to monitor clock counting and event counting in Intel XScale core's PMU. It tells the user how many processor cycles are taken and how many times an event has occurred.

The number of events that can be monitored simultaneously range from 0 to 4 at a time. When the number of event to monitor is set to 0, only clock counting is performed. The clock count can be set to be incremented by one at each 64th processor clock cycle or at every processor clock cycle.

The steps needed to run this utility are:

1. To begin the clock and event counting, call the start function with parameters:

```
ixPerfProfAccXscalePmuEventCountStart (
BOOL clkCntDiv,
UINT32 numEvents,
IxPerfProfAccXscalePmuEvent pmuEvent1,
IxPerfProfAccXscalePmuEvent pmuEvent2,
IxPerfProfAccXscalePmuEvent pmuEvent3,
IxPerfProfAccXscalePmuEvent pmuEvent4
```

   — BOOL [in] clkCntDiv — Enables/disables the clock divider. When true, the divider is enabled and the clock count will be incremented by one at each 64th processor clock cycle. When false, the divider is disabled and the clock count will be incremented at every processor clock cycle.

   — UINT32 [in] numEvents — Number of PMU events that are to be monitored as specified by the user. For clock counting only, this is set to zero.

   — pmuEvent1, pmuEvent2, pmuEvent3, pmuEvent4 — The specific PMU events to be monitored by counters as described in section 14.2 and defined in IxPerfProfAccXscalePmuEvent:

```
IxPerfProfAccXscalePmuEvent {
IX_PERFPROF_ACC_XSCALE_PMU_EVENT_CACHE_MISS = 0,
IX_PERFPROF_ACC_XSCALE_PMU_EVENT_CACHE_INSTRUCTION,
IX_PERFPROF_ACC_XSCALE_PMU_EVENT_STALL,
IX_PERFPROF_ACC_XSCALE_PMU_EVENT_INST_TLB_MISS,
IX_PERFPROF_ACC_XSCALE_PMU_EVENT_DATA_TLB_MISS,
IX_PERFPROF_ACC_XSCALE_PMU_EVENT_BRANCH_EXEC,
IX_PERFPROF_ACC_XSCALE_PMU_EVENT_BRANCH_MISPREDICT,
IX_PERFPROF_ACC_XSCALE_PMU_EVENT_INST_EXEC,
IX_PERFPROF_ACC_XSCALE_PMU_EVENT_FULL_EVERYCYCLE,
IX_PERFPROF_ACC_XSCALE_PMU_EVENT_ONCE,
IX_PERFPROF_ACC_XSCALE_PMU_EVENT_DATA_CACHE_ACCESS,
IX_PERFPROF_ACC_XSCALE_PMU_EVENT_DATA_CACHE_MISS,
IX_PERFPROF_ACC_XSCALE_PMU_EVENT_DATA_CACHE_WRITEBACK,
IX_PERFPROF_ACC_XSCALE_PMU_EVENT_SW_CHANGE_PC,
IX_PERFPROF_ACC_XSCALE_PMU_EVENT_MAX }
```

2. To end the counting, call the stop function with parameters:

```
ixPerfProfAccXscalePmuEventCountStop (
IxPerfProfAccXscalePmuResults *eventCountStopResults)
```

   This function can only be called once IxPerfProfAccEventCountStart has been called. It is the user's responsibility to allocate the memory for the results pointer before calling the function. The user may then read/print the values stored in this pointer to obtain the results of the clock/event counting process. It contains all values of counters and associated overflows.

   If the user has declared a variable `IxPerfProfAccXscalePmuResults eventCountStopResults`, the user may then print out the result for all the counters as shown in Figure 69.

Figure 69. Display Performance Counters

```
printf("Lower 32 bits of clock count = %u\n", eventCountStopResults.clk_value);
printf("Upper 32 bits of clock count = %u\n", eventCountStopResults.clk_samples);
printf("Lower 32 bits of event 1 count = %u\n", eventCountStopResults.event1_value);
printf("Upper 32 bits of event 1 count = %u\n", eventCountStopResults.event1_samples);
printf("Lower 32 bits of event 2 count = %u\n", eventCountStopResults.event2_value);
printf("Upper 32 bits of event 2 count = %u\n", eventCountStopResults.event2_samples);
printf("Lower 32 bits of event 3 count = %u\n", eventCountStopResults.event3_value);
printf("Upper 32 bits of event 3 count = %u\n", eventCountStopResults.event3_samples);
printf("Lower 32 bits of event 4 count = %u\n", eventCountStopResults.event4_value);
printf("Upper 32 bits of event 4 count = %u\n", eventCountStopResults.event4_samples);
```

3. If at any time before, during, or after the counting process, the user wishes to view the value of all four event counters and the clock counter, the user may call the following function with parameters:

   ```
   ixPerfProfAccXscalePmuResultsGet(IxPerfProfAccXscalePmuResults *results)
   ```

   The user may then read/print out the results of all the counters, as shown in Figure 69.

17.9.1.2 Time-Based Sampling

This utility can be used to profile the user's code through time sampling, which records PC addresses at fixed intervals. It tells the user which lines of code are most frequently executed, by creating a profile of the code which shows the PC addresses in the code that were sampled and the frequency of their occurrence.

The sampling rate is defined by the user and is the number of clock counts before a sample is taken. The steps needed to run this utility are:

1. To begin the time sampling, call the start function with parameters:

   ```
   ixPerfProfAccXscalePmuTimeSampStart(UINT32 samplingRate,
   BOOL clkCntDiv)
   ```

   — UINT32 [in] samplingRate — The number of clock counts before a sample is taken; the rate specified cannot be greater than the counter size of 32bits or set to zero.

   — BOOL [in] clkCntDiv — Enables/disables the clock divider. When true, the clock count will be incremented by one at each 64th processor clock cycle. When false, the clock count will be incremented at every processor clock cycle.

   This API starts the time based sampling to determine the frequency with which lines of code are being executed. Sampling is done at the rate specified by the user. At each sample, the value of the program counter is determined. Each of these occurrences are recorded to determine the frequency with which the Intel XScale core's code is being executed. This API has to be called before ixPerfProfAccXscalePmuTimeSampStop can be called.

2. To end the time sampling, call the stop function, with parameters:

   ```
   ixPerfProfAccXscalePmuTimeSampStop(
   IxPerfProfAccXscalePmuEvtCnt *clkCount,
   IxPerfProfAccXscalePmuSamplePcProfile *timeProfile)
   ```

   This function can only be called once *ixPerfProfAccXscalePmuTimeSampStart* has been called. It is the user's responsibility to allocate the memory for the pointers before calling this

function. The user may then read/print the values stored in these pointers to obtain the results of the time sampling process:

— *clkCount* — Indicates the number of clock cycles that elapsed,

— *timeProfile* — Contains the unique PC addresses and their occurencefrequencies.

For example, if the user has declared a pointer “IxPerfProfAccXscalePmuEvtCnt clkCount”, the user may then print out the value of the clock counter (which indicates the number of clock cycles that elapsed) as shown below.

Figure 70. Display Clock Counter

```
printf("\n Lower 32 bits of clock count: 0x%x", clkCount.lower32BitsEventCount);
printf("\n Upper 32 bits of clock count: 0x%x", clkCount.upper32BitsEventCount);
```

The following example shows how to process an array of *IxPerfProfAccXscalePmuSamplePcProfile:*

If the user has declared a pointer to an array...

```
IxPerfProfAccXscalePmuSamplePcProfile
timeProfile[IX_PERFPROF_ACC_XSCALE_PMU_MAX_PROFILE_SAMPLES],
```

...the user may then print out the top five PC addresses in the time profile as follows:

i. Obtain the number of samples which were taken. For example:

```
clkSamples = clkCount.upper32BitsEventCount
```

ii. Determine the number of elements in the timeProfile array, which is the number of unique PC addresses by adding up the elements in the array that contain results:

```
UINT32 test_freq;
UINT32 frequency;  /*total number of samples collected*/
UINT32 numPc = 0; /*number of unique PC addresses*/

for (frequency=0; frequency< =clkSamples; frequency+=test_freq)
{
    test_freq = timeProfile[numPc].freq;
    numPc ++;
}
```

iii. Use a sorting function of the user’s choice to sort the timeProfile array in descending order:

```
qsort(timeProfile, numPc, sizeof(IxPerfProfAccXscalePmuSamplePcProfile), (void *)compFreq)
```

Note: Note: compFreq is a compare function that is to be defined by the user

iv. Print out the first five elements:

```
for (i=0; i++; i<5)
{
    printf("timeprofile element %d pc value = 0x%x\n", i,
    timeProfile[i].programCounter);
    printf("timeprofile element %d freq value = %d\n", i,
    timeProfile[i].freq);
}
```

These profile results show the places in the user's code that are most frequently being executed and that are taking up the most processor cycles.

17.9.1.3 Event-Based Sampling

This utility can be used to profile the user's code through event sampling. The process is similar to that of time sampling. However, this utility tells the user which lines of codes trigger occurrences of the events specified by the user. The sampling rate is defined by the user and is the number of events before a sample is taken. Each event defined, may have its own sampling rate. The steps needed to run this utility are:

1. To begin the event sampling, call the start function with parameters:

```
ixPerfProfAccXscalePmuEventSampStart(
UINT32 numEvents,
IxPerfProfAccXscalePmuEvent pmuEvent1, UINT32 eventRate1,
IxPerfProfAccXscalePmuEvent pmuEvent2, UINT32 eventRate2,
IxPerfProfAccXscalePmuEvent pmuEvent3, UINT32 eventRate3,
IxPerfProfAccXscalePmuEvent pmuEvent4, UINT32 eventRate4)
```

This function starts the event-based sampling to determine the frequency with which events are being executed. The sampling rate is the number of events, as specified by the user, before a counter overflow interrupt is generated.

A sample is taken at each counter overflow interrupt. At each sample,the value of the program counter determines the corresponding location in the code. Each of these occurrences are recorded to determine the frequency with which the Intel XScale core's code in each event is executed.

This API has to be called before ixPerfProfAccXscalePmuEventSampStop can be called.

— UINT32 [in] <numEvents> — The number of PMU events that are to be monitored as specified by the user. The value should be between 1-4 events at a time.

— IxPerfProfAccXscalePmuEvent [in] pmuEvent1 — The specific PMU event to be monitored by counter 1

— UINT32 [in] eventRate1, eventRate2, eventRate3, eventRate4 — The number of events before a sample taken. If 0 is specified, the the full counter value (0xFFFFFFFF) is used. The rate must not be greater than the full counter value.

2. To end the event sampling, call the stop function, with parameters:

```
ixPerfProfAccXscalePmuEventSampStop(
IxPerfProfAccXscalePmuSamplePcProfile *eventProfile1,
IxPerfProfAccXscalePmuSamplePcProfile *eventProfile2,
IxPerfProfAccXscalePmuSamplePcProfile *eventProfile3,
IxPerfProfAccXscalePmuSamplePcProfile *eventProfile4)
```

It is the user's responsibility to allocate the memory for the pointers before calling this function. The user may then read/print the values stored in these pointers to obtain the results of the event sampling process. The user may obtain the number of samples for each event counter by calling the function *ixPerfProfAccXscalePmuResultsGet*().

Meanwhile, the event profiles will show the user the parts of the code that cause the specified events to occur.

In order to interpret the results, the user may want to map the PC addresses captured to their corresponding functions or locations in the user's code. This can be done as in the following examples:

i. In vxWorks, this can be done from the windshell. If the user obtains a PC address of "0x0008d448" the corresponding function(or location in the code) can be found by calling:

```
lkAddr 0x0008d448
```

ii. In Linux*, the corresponding function (or location in the code) can be inside the ksyms file that is produced in directory /proc/ksyms after running the perfProf test code. This file contains a list of PC addresses and their corresponding function names.

17.9.1.4 Using Intel XScale® Core PMU to Determine Cache Efficiency

In this example, the user would like to monitor the instruction cache efficiency mode. The user would use the event counting process to count the total number of instructions that were executed and instruction cache misses requiring fetch requests to external memory.

The remaining two counters will not provide relevant results in this example. The counters may be set to the appropriate default event value.

1. To begin the counting, call the start function, with parameters:

```
ixPerfProfAccXscalePmuEventCounting (FALSE, 2,
IX_PERFPROF_ACC_XSCALE_PMU_EVENT_INST_EXEC,
IX_PERFPROF_ACC_XSCALE_PMU_EVENT_CACHE_MISS,
IX_PERFPROF_ACC_XSCALE_PMU_EVENT_MAX,
IX_PERFPROF_ACC_XSCALE_PMU_EVENT_MAX)
```

2. Declare a results variable:

```
IxPerfProfAccXscalePmuResults results;
```

3. To end the counting, call the stop function, with parameters:

```
ixPerfProfAccXscalePmuEventCountStop (
IxPerfProfAccXscalePmuResults &results)
```

4. Print the total value (combining the upper and lower 32 bits) of all the counters:

```
printf("total clk count = 0x%x%x\n", results.clk_samples, results.clk_value);
printf("total event 1 count = 0x%x%x\n", results.event1_samples, results.event1_value);
printf("total event 2 count = 0x%x%x\n", results.event2_samples, results.event2_value);
printf("total event 3 count = 0x%x%x\n", results.event3_samples, results.event3_value);
printf("total event 4 count = 0x%x%x\n", results.event4_samples, results.event4_value);
```

Note: As only event counters one and two were configured to monitor events, the results of event counters 3 and 4 will remain at zero and will be irrelevant.

5. The appropriate statistics can be calculated from the results to determine the instruction cache efficiency. The instruction cache miss rate is the instruction cache misses (monitored by event counter two) divided by the total number of instructions executed (monitored by event counter one):

   Instruction cache miss rate

   — =instruction cache misses/total number of instructions executed

   — = total event count 2 / total event count 1

6. The average number of cycles it took to execute an instruction (also known as cycles-per-instruction), is the total clock count (monitored by the clock counter) divided by the total number of instructions executed (monitored by event counter 1):

```
cycles-per-instruction = total clock count  / total number of instructions executed
                       = total clk count  / total event count 1
```

17.9.2 Internal Bus PMU

The Internal Bus PMU utility enables performance monitoring of components accessing or utilizing the north and south bus, provides statistics of the north and south bus and SDRAM, and allows the user to read the value of the Previous Master Slave Register. The user may monitor their code/program in two ways:

- From the CLI, call the appropriate Internal Bus PMU utility
- In the user's code itself, insert the appropriate Internal Bus PMU utility's start and stop functions.

To run this utility:

1. Begin the measurements, call the start function with parameters:

```
ixPerfProfAccBusPmuStart(
IxPerfProfAccBusPmuMode mode,
IxPerfProfAccBusPmuEventCounters1 pecEvent1,
IxPerfProfAccBusPmuEventCounters2 pecEvent2,
IxPerfProfAccBusPmuEventCounters3 pecEvent3,
IxPerfProfAccBusPmuEventCounters4 pecEvent4,
IxPerfProfAccBusPmuEventCounters5 pecEvent5,
IxPerfProfAccBusPmuEventCounters6 pecEvent6,
IxPerfProfAccBusPmuEventCounters7 pecEvent7)
```

   This function initializes all the counters and assigns the events associated with the counters. Selecting HALT mode will generate error. User should use ixPerfProfAccBusPmuStop() to HALT.

2. To end the measurements, call the stop function, to stop all the counters:

```
ixPerfProfAccBusPmuStop()
```

3. If at any time before, during, or after the counting process, the user wishes to view the value of the counters, the user may call the following function, with parameter:

```
ixPerfProfAccBusPmuResultsGet (IxPerfProfAccBusPmuResults *busPmuResults)
```

It is the user's responsibility to allocate the memory for the pointer before calling this function. The user may then read/print the values stored in this pointer to obtain the results of the measurements.

`IxPerfProfAccBusPmuResults` has two arrays:

— For the lower 27-bit of counter values —

```
UINT32 statsToGetLower27Bit [IX_PERFPROF_ACC_BUS_PMU_MAX_PECS]
```

— For upper 32 Bit of counter values. The user should be aware that in the lower 27-bit counter, it only stores values up to 27 bits before causing an overflow —

```
UINT32 statsToGetUpper32Bit [IX_PERFPROF_ACC_BUS_PMU_MAX_PECS]
```

For example:

— If the user has declared a variable "IxPerfProfAccBusPmuResults busPmuResults", the user may then print out the value of all seven of the PEC counters. The user should be aware that in the lower 27-bit counter, it only stores values up to 27 bits before causing an overflow. Therefore, in order to combine the lower 27-bit value with the upper-32 bit value, the following calculations are done:

```
lower32Bits = (lower 27-bit counter value) +[(upper 32-bit counter value) & 0x1F ) << 27 ]
upper32Bits = (upper 32-bit counter value) >> 5
Total PEC counter value = (upper32Bits<<32) |lower32Bits
```

— If the user declares variables "UINT32 lower32Bits" and "UINT32 upper32Bits",and assigns them to the values calculated above, the user may print out the results as follows:

```
for (i = 0; i< IX_PERFPROF_ACC_BUS_PMU_MAX_PECS ; i++)
{
printf ("\n The value of  PEC %d = 0x%8x%8x ", i, upper32Bits, lower32Bits);
}
```

This will print out the entire value of the PC in hex.

Note: For the ixPerfProfAccBusPmuPMSRGet() function, the user may refer to the codelet for a detailed description.

17.9.2.1 Using the Internal Bus PMU Utility to Monitor Read/Write Activity on the North Bus

In this example, the user would like to monitor the number of cycles where the north bus is either idle, or is being written to or read from. In order to do so, the user selects the north mode. PECs 1, 2, and 3 will be set to monitor the number of cycles the bus is doing data writes/reads or is idle.

PEC 7 will be set to monitor the total number of cycles. Meanwhile, the remaining counters will not provide relevant results in this examples, therefore, they may be set to any appropriate north mode event.

1. To begin the measurements, call the start function with parameters:

```
ixPerfProfAccBusPmuStart (
IX_PERFPROF_ACC_BUS_PMU_MODE_NORTH,
IX_PERFPROF_ACC_BUS_PMU_PEC1_NORTH_BUS_IDLE_SELECT,
IX_PERFPROF_ACC_BUS_PMU_PEC2_NORTH_BUS_WRITE_SELECT,
IX_PERFPROF_ACC_BUS_PMU_PEC3_NORTH_BUS_READ_SELECT,
    IX_PERFPROF_ACC_BUS_PMU_PEC4_NORTH_ABB_SPLIT_SELECT,
IX_PERFPROF_ACC_BUS_PMU_PEC5_NORTH_PSMB_GRANT_SELECT,
IX_PERFPROF_ACC_BUS_PMU_PEC6_NORTH_PSMC_GRANT_SELECT,
IX_PERFPROF_ACC_BUS_PMU_PEC7_CYCLE_COUNT_SELECT)
```

2. After an appropriate amount of time, end the measurements by calling the stop function:

```
ixPerfProfAccBusPmuStop(void)
```

3. Declare a variable for the results:

```
IxPerfProfAccBusPmuResults results
```

4. Obtain the results by calling:

```
ixPerfProfAccBusPmuResultsGet (&results)
```

5. Print the value of all the PECs:

```
for (i = 0; i< IX_PERFPROF_ACC_BUS_PMU_MAX_PECS ; i++)
{
    printf ("\nPEC %d = upper 0x%x lower 0x%x ", i,
        results.statsToGetUpper32Bit[i], results.statsToGetLower27Bit[i]);
}
```

6. Print the total value of PECs 1-3, and PEC 7.

   The upper 32 bits reflect the number of times the lower 27-bit value overflowed:

```
printf ("Total value of PEC1 0x%8x%8x",
    results.statsToGetUpper32Bit[0],
    results.statsToGetLower27Bit[0]);
```

7. Perform the same calculation for the rest of the PEC's.

```
PEC1_total = total value of north bus idle cycles
PEC2_total = total value of north bus data write cycles
PEC3_total = total value of north bus date read cycles
PEC7_total = total value of cycles available
```

8. Determine the percentage of cycles that the bus was either idle or performing Data Writes/ Reads:

```
Percentage of idle cycles = (PEC1_total /PEC7_total) *100%
Percentage of data write cycles = (PEC2_total/PEC7_total) * 100%
Percentage of date read cycles = (PEC3_total/PEC7_total) * 100%
```

17.9.3 Xcycle (Idlecycle Counter)

The Xcycle utility calculates the cycles remaining compared with the cycles available during an idle period. The user may monitor the load of their program by obtaining the percentage of idle cycles available with their program running.

The user may monitor their code/program by creating a thread that runs the code being monitored. At the same time, on a separate thread, run the Xcycle utility. To run this utility:

1. Before creating any other threads, perform calibration and obtain the baseline (i.e. the total available cycles in the period of time specified) when there is no load:

   ```
   ixPerfProfAccXcycleBaselineRun (UINT32 *numBaselineCycle)
   ```

   It is the user's responsibility to allocate the memory for the pointer before calling this function. The user may then read/print this pointer to obtain the total available cycles when there is no load on the system.

   This pointer is interpreted as "the number of 66-Mhz clock ticks for one measurement." It is stored within the tool while it is being run and serves only as a reference for the user.

2. Create a thread that runs the code to be monitored. To begin the Xcycle measurements, call the start function, with parameter:

   ```
   ixPerfProfAccXcycleStart(UINT32 numMeasurementsRequested)
   ```

   This start the measurements immediately. numMeasurementsRequested specifies number of measurements to run.

   If numMeasurementsRequested is set to 0, the measurement will be performed continuously until IxPerfProfAccXcycleStop() is called. It is estimated that one measurement takes approximately 1 s during low CPU utilization, therefore 128 measurement takes approximately 128 s.

   When CPU utilization is high, the measurement will take longer. This function spawn a task the perform the measurement and returns. The measurement may continue even if this function returns.

   There are only IX_PERFPROF_ACC_XCYCLE_MAX_NUM_OF_MEASUREMENTS storage available so storing is wrapped around if measurements are more than IX_PERFPROF_ACC_XCYCLE_MAX_NUM_OF_MEASUREMENTS

3. If ixPerfProfAccXcycleStart() is called with an input of zero, this indicates continuous measurements. In this case, the measurements are stopped, by calling the stop function:

   ```
   ixPerfProfAccXcycleStop(void)
   ```

   As it takes the measurements some time to complete, the user should call the following function to determine if any measurements are still running:

   ```
   ixPerfProfAccXcycleInProgress(void)
   ```

4. To obtain the results of the measurements made, the user should call the results function, with parameter:

   ```
   ixPerfProfAccXcycleResultsGet(IxPerfProfAccXcycleResults *xcycleResult)
   ```

   The result contains:

   — float maxIdlePercentage — Maximum percentage of Idle cycles

— float minIdlePercentage — Minimum percentage of Idle cycles

— float aveIdlePercentage — Average percentage of Idle cycles

— UINT32 totalMeasurements — Total number of measurement made

If the user has declared a pointer `IxPerfProfAccXcycleResults *xcycleResult`, the user may then print out the results of the xcycle measurements as shown in Figure 71.

Figure 71. Display Xcycle Measurement

```
printf("Maximum percentage of idle cycles = %f\n", xcycleResult->maxIdlePercentage);
    printf("Minimum percentage of idle cycles = %f\n", xcycleResult->minIdlePercentage);
    printf("Average percentage of idle cycles = %f\n", xcycleResult->aveIdlePercentage);
    printf("Total number of measurements = %u\n", xcycleResult->totalMeasurements);
```

Access-Layer Components: Queue Manager (IxQMgr) API 18

This chapter describes the Intel® IXP400 Software v.1.3's "Queue Manager API" access-layer component.

18.1 Overview

The IxQMgr is a collection of software services responsible for configuring the Advanced High-Performance Bus (AHB) Queue Manager (also referred to by the combined acronym, AQM). IxQMgr is also responsible for managing messages between the NPEs and client Intel XScale® Core software. To do this, the IxQMgr API provides a low-level interface to the AHB queue manager hardware block of the IXP42X product line. Other Intel XScale core components can then use IxQMgr to pass and receive data to and from the NPEs through the AHB queue manager.

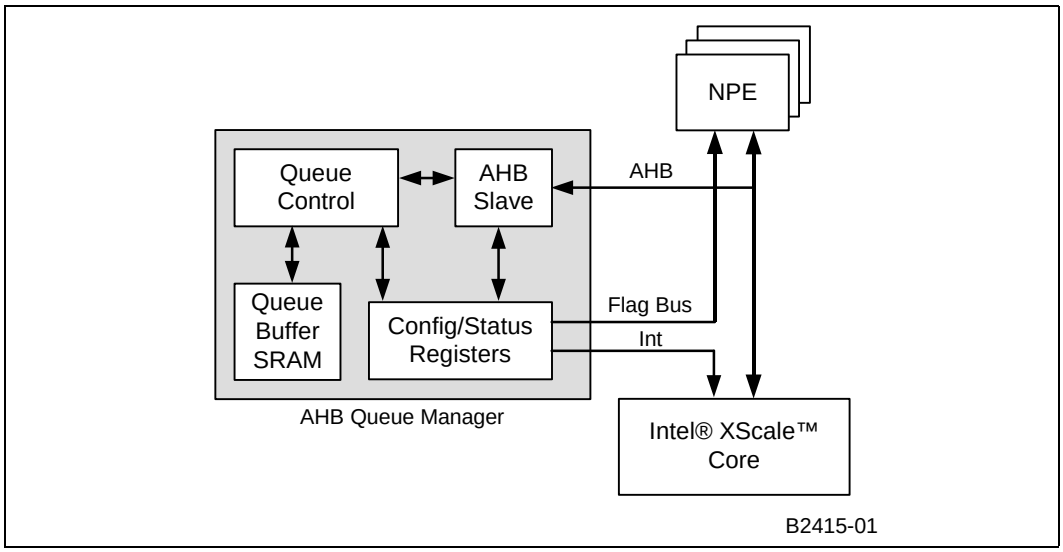
This chapter contains the necessary steps to start the IxQMgr component. Additionally, information has been included about how the AHB Queue Manager functions from a high-level view. The IxQMgr component acts a pseudo service layer to other access components such as EthAcc.

In the sections that describe how the queue manager works, the "client" is an access component such as EthAcc. An application programmer will not need to do any coding to directly control the queues and the AHB queue manager, just the initialization and uninitialization of the IxQMgr component.

Callback function registration must still be done, but it is done outside of the IxQMgr interface. For example, if an application programmer needed to register a callback function for EthAcc, EthAcc would make the appropriate function calls to the IxQMgr component.

18.2 Features and Hardware Interface

Figure 72. AHB Queue Manager Hardware Block



The IxQMgr provides a low-level interface for configuring the AHB queue manager, which contains the physical block of static RAM where all the data structures (queues) for the IxQMgr reside. The AHB queue manager provides 64 independent queues in which messages, pointers, and data 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.

18.3 IxQMgr Initialization and Uninitialization

The initialization of IxQMgr first requires a call to ixQMgrInit(), which takes no parameters and returns success or failure. No other ixQMgr functions may be called before this. After initialization, the queues will be configured, and the dispatcher will be started.

Sample code for starting the dispatcher is included in Figure 74 on page 210. This codelet is also a good reference to see how the queues are setup for an ethernet traffic application. The IxQMgr dispatcher API is a good source to get an additional level of insight into the performance of IxQMgr.

To uninitialize the IxQMgr component call the ixQMgrUnload() function, which also takes no parameters and returns success or failure. This uninitialization will unmap kernel memory mapped by the component and should be done before unloading a kernel module or before a soft reset if possible.

The ixQMgrUnload function should not be called twice in sequence before a call to ixQMgrInit, or unpredictable results will occur.

18.4 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.

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

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

  The amount of SRAM needed for the configuration is checked at run-time.

18.5 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 indentifiers are named as follows:

```
IX_QMGR_QUEUE_0, IX_QMGR_QUEUE_1, IX_QMGR_QUEUE_2,.....IX_QMGR_QUEUE63
```

18.6 Configuration Values

Table 36 details the attributes of a queue that can be configured and the possible values that these attributes can take on (word = 32 bits).

Table 36. AHB Queue Manager Configuration Attributes

Attribute	Description	Values
Queue Size	The maximum number of words that the queue can contain. 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

18.7 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.

A parameter passed to the ixQMgrDispatcherLoopRun() function determines which queues are serviced by the dispatcher each time ixQMgrDispatcherLoopRun() is called. The parameter will specify if queues 0-31 or 32-63 are serviced. The order in which queues are serviced depends on the priority specified by calling ixQMgrQDispatchPrioritySet().

Figure 73. AHB Queue Manager Dispatcher Operation

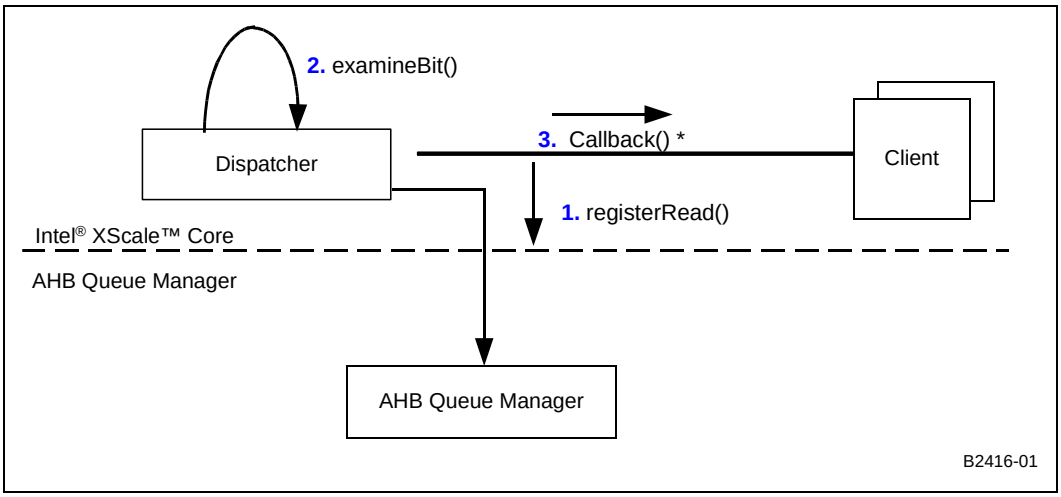


Figure 73 shows the operation of the dispatcher called in the context of an ISR. The interrupt register indicates which queues fired the interrupt. Each bit in this register is examined and, if set, the dispatcher will call each callback registered with this queue.

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 74. 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 74 is taken from the ixEthAcc codelet.

18.8 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.

18.9 Dependencies

The IxQMgr component is dependant on the ixOsServices component. IxQMgr uses ixOsServices in ixQMgrDispatcherInterruptConnect() to register AQM ISRs. The user is responsible for registering interrupt handlers, as shown in the code snippet in Figure 74.

This page intentionally left blank.

Access-Layer Components: UART-Access (IxUARTAcc) API 19

This chapter describes the Intel® IXP400 Software v.1.3's "UART-Access API" access-layer component.

The UARTs of the IXP42X product line's processors 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 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 is:

- 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 UART driver provides generic support for polled and loop back mode only.

19.1 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

19.2 UART / OS Dependencies

The UART device driver is an API than can be used to transmit/receive data from either of the two UART ports on the processor. However, it is expected that an RTOS will provide standard UART services independent from the IxUartAcc device driver. That is, the RTOS Uart services will configure and utilize the UART registers and FIFOs directly.

Users of the IxUartAcc component should ensure that the use of this device driver does not conflict with any UART services provided by the RTOS.

19.2.1 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 receive FIFO is wider to accommodate the potentially largest data word (i.e., including optional stop bits and parity 8+2+1=11).

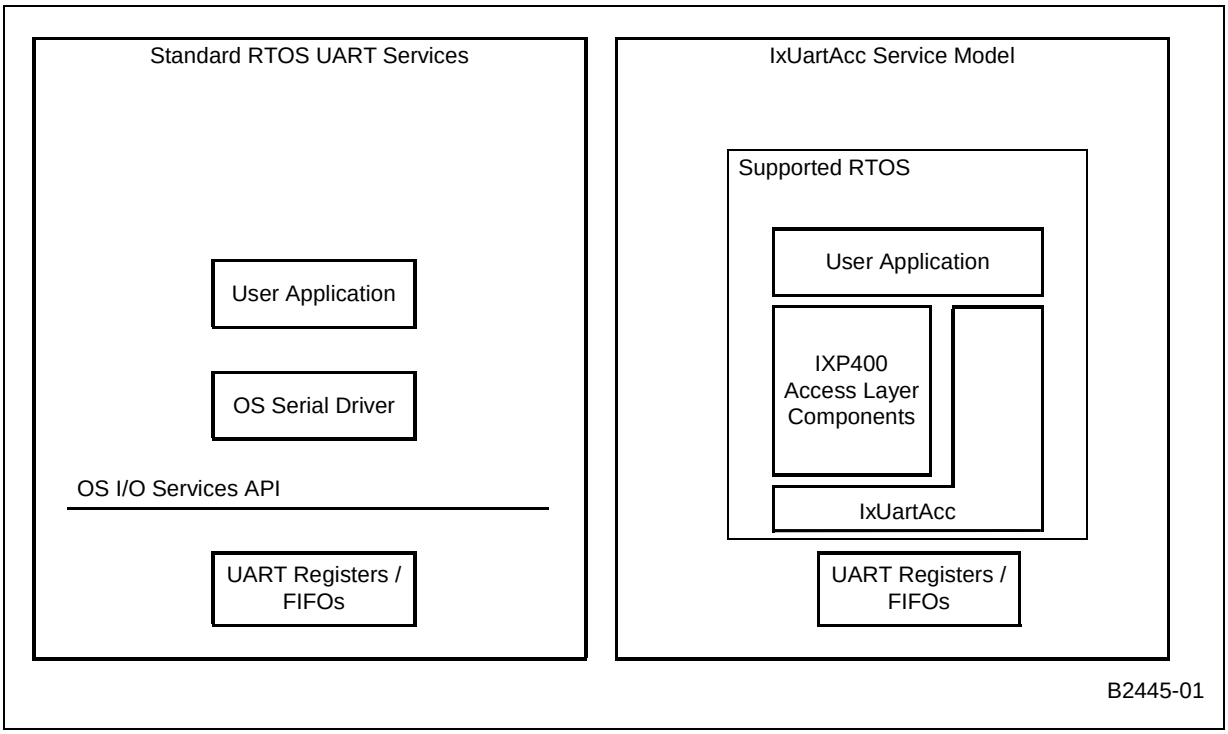
Interrupts can occur in one of two ways. One is when the FIFO has reached its programmed trigger level (set by the FIFO Control Register [FCR]). The other is when a character timeout has occurred (also set in the FCR). The driver will implement both modes of operation.

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)

19.3 Dependencies

Figure 75. UART Services Models



This page intentionally left blank.

Access-Layer Components: USB Access (ixUSB) API 20

This chapter describes the Intel® IXP400 Software v.1.3's "USB Access API" access-layer component.

20.1 Overview

The IXP42X product line's USB hardware 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 / IXCDP1100 Development Platform board-support package (BSP), available from the Wind River Systems* Web site. If this specific USB functionality is required on VxWorks, a USB developer's kit must be purchased directly from Wind River.

20.2 USB Controller Background

The IXP42X product line'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.

20.2.1 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 37. 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 38. The presence of SOF packets every 1ms prevents the UDC from going into suspend mode.

Table 38. 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 39. 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 39. 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 40 shows the format of a handshake packet.

Table 40. Handshake Packet Format

8 Bits	8 Bits
Sync	PID

20.2.2 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 41. (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 41. Bulk 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**.

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 42.

Table 42. 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 43 shows the four possible types of control transactions.

Table 43. 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. 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 44. Control Transaction Formats

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

NOTE: Packets 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 45 shows the eight possible types of interrupt transactions.

Table 45. 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**.

20.3 ixUSB API Interfaces

Table 46. 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.
ixUSBErrorStringGet	Convert an error code into a human-readable string error message.
ixUSBEndpointInfoShow	Display endpoint information table.

The ixUSB API components operate within a callback architecture. Initial device setup and configuration is controlled through the callback registered during the ixUSBSetupCallbackRegister function. Data reception occurs through the callback registered during the ixUSBReceiveCallbackRegister function. Special events are signalled to the callback registered during the ixUSBEventCallbackRegister function.

Prior to using any other ixUSB API, the ixUSB client must initialize the controller with the ixUSBDriverInit API call. After this call the driver is in a disabled state. The call to ixUSBDeviceEnable allows data, setup, and configuration transmissions to flow.

20.3.1 ixUSB Setup 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 are 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

The UDC decodes most commands with no intervention required by the ixUSB client. Other setup requests occur through the setup callback. The following data structure in Table 47, "USBSetupPacket" is passed to the setup callback function so the software can be configured properly.

Table 47. USBSetupPacket

```
typedef struct /* USBSetupPacket */
{
    UCHAR bmRequestType;
    UCHAR bRequest;
    UINT16 wValue;
    UINT16 wIndex;
    UINT16 wLength;
} USBSetupPacket;
```

Table 48 shows a summary of the setup device requests.

Table 48. Host-Device Request Summary (Sheet 1 of 2)

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.

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

Table 48. Host-Device Request Summary (Sheet 2 of 2)

Request	Name
SET_DESCRIPTOR	Sets existing descriptors or adds new descriptors. Existing descriptors include: † • Device • Configuration • String • 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: • Remote wake-up • Self-powered • Data direction • 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.

Via control endpoint 0, the user must decode and respond to the GET_DESCRIPTOR command.

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

20.3.1.1 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.

20.3.1.2 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 functions (see ixUSBFrameCounterGet).

20.3.2 ixUSB Send and Receive Requests

The USB access layer encodes and decodes data frames sending and receiving buffers to and from the client in the same format as IX_MBUF.

Buffers are sent from the UDC to the host with the ixUSBBufferSubmit API.

Data buffers are received from the host through the callback function registered with the access layer during the ixUSBReceiveCallbackRegister API call.

20.3.3 ixUSB Endpoint Stall Feature

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 76. STALL on IN Transactions

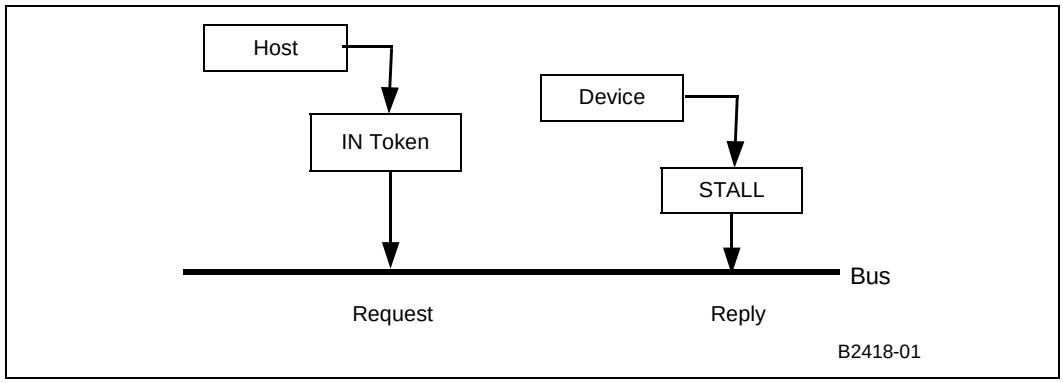
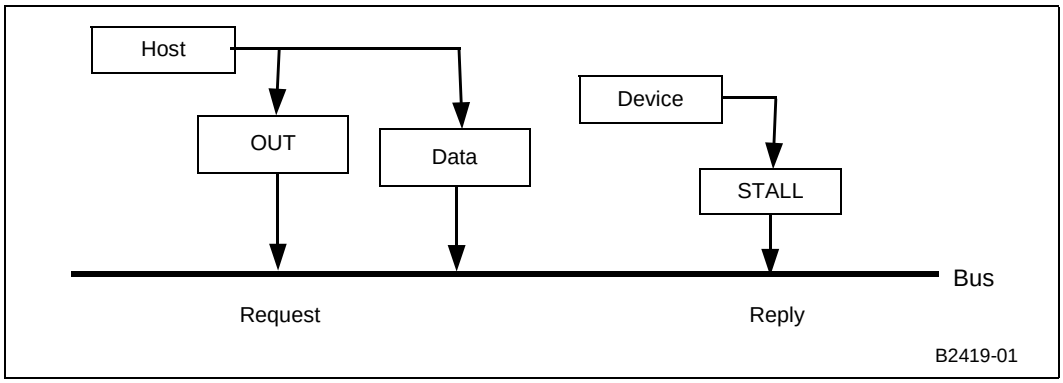


Figure 77. STALL on OUT Transactions



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.

20.3.4 ixUSB 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 USB Driver 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.

Table 49. 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.

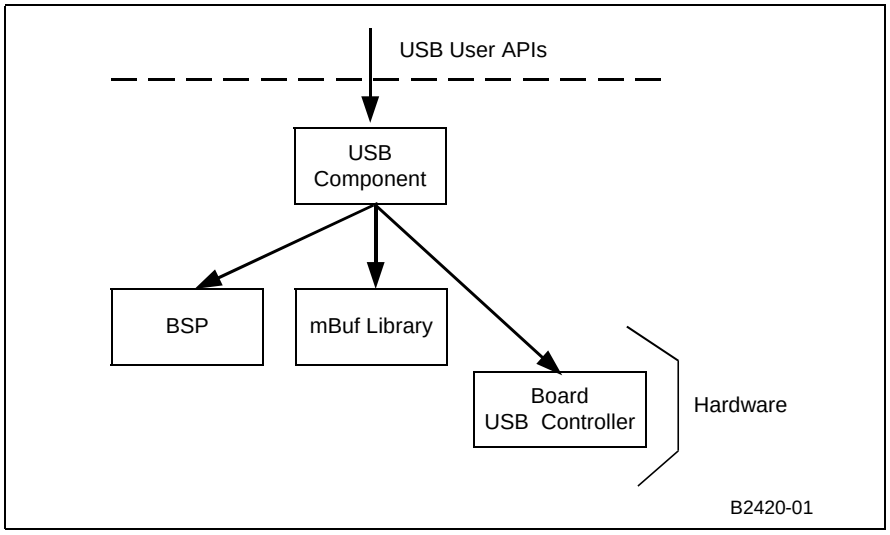
20.4 USB Data Flow

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.

20.5 USB Dependencies

The USB device driver is a self-contained component with no interactions with other data components. Figure 78 shows the dependencies for this USD component.

Figure 78. USB Dependencies



Codelets 21

This chapter describes the Intel® IXP400 Software v.1.3 codelets.

21.1 Overview

The codelets are example code that utilize the access-layer components and operating system abstraction layers discussed in the preceding chapters. Codelets, while not exhaustive examples of the functionality available to the developer, provide a good basis from which to begin their own code development for test harnesses, performance analysis code, or even functional applications to take to market.

This chapter describes the major features of the available in each codelet. For detailed information, refer to "Codelet Reference" on page 625 or the header and source files provided with software release 1.3 in the xscale_sw/src/codelets directory.

21.2 ATM Codelet (IxAtmCodelet)

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.

This codelet also demonstrates an example implementation of OAM F4 Segment, F4 End-To-End (ETE), F5 Segment and F5 ETE loopback. Aal5 or Aal0 (48 or 52 bytes) traffic types are available in this codelet, as well as the display of transmit and receive statistics.

This codelet is not supported in little-endian mode.

IxAtmCodelet makes use of the following access-layer components:

- IxAtmdAcc
- IxAtmm
- IxAtmSch

21.3 Crypto Access Codelet (IxCryptoAccCodelet)

This codelet demonstrates how to use the IxCrypto access layer component and the underlying security features in the IXP42X product line. IxCryptoAccCodelet runs through the scenarios of initializing the NPEs and Queue Manager, context registration, and performing a variety of encryption, decryption, and authentication operations.

The codelet also performs some performance measurements of the cryptographic operations.

21.4 DMA Access Codelet (IxDmaAccCodelet)

The DMA Access Codelet executes DMA transfer for various DMA transfer modes, addressing modes and transfer widths. The block sizes used in this codelet are 8; 1,024; 16,384; 32,768; and 65,528 bytes. For each DMA configuration, the performance is measured and the average rate (in Mbps) is displayed.

This codelet is not supported in little-endian mode.

21.5 Ethernet Aal5 Codelet (IxEthAal5App)

IxEthAal5App codelet is a mini-bridge application which bridges traffic between Ethernet and UTOPIA ports or Ethernet and ADSL ports. Two Ethernet ports and up to 8 UTOPIA PHYs are supported, which are initialized by default at the start of application.

Ethernet frames are transferred across ATM link (through Utopia interface) using AAL-5 protocol and Ethernet frame encapsulation described by RFC 1483. MAC address learning is performed on Ethernet frames, received by Ethernet ports and ATM interface (encapsulated). IxEthAal5App filters packets based on destination MAC addresses.

Forwarding is done only between Ethernet and UTOPIA port.

This codelet is not supported in little-endian mode.

IxEthAal5App makes use of the following access layer components:

- IxEthAcc
- IxAtmdAcc
- IxAtmm
- IxAtmSch
- IxQmgr

21.6 Ethernet Access Codelet (IxEthAccCodelet)

This codelet demonstrates both Ethernet data and control plane services and Ethernet management services.

- Ethernet data and control plane services:
  - — Configuring both ports as a receiver sink from an external source (such as Smartbits)
  - — Configuring Port-1 to automatically transmit frames and Port-2 to receive frames. Frames generated and transmitted in Port-1 are looped back into Port-2 by using cross-over cable.
  - — 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.
- Ethernet management services:
  - — Adding and removing static/dynamic entries.

— Calling the maintenance interface (run as a separate background task)

— Calling the show routine to display the MAC address filtering tables.

IxEthAccCodelet demonstrates the use of many of the access layer components.

21.7 Fast Path Access Codelet (IxFpathAccCodelet)

This codelet demonstrates two top-level scenarios: an Ethernet bridge forwarding fast-path Ethernet frames to a single VC, and an Ethernet router forwarding IP packets to a single VC.

Bridge Example

The Fast Path codelet initializes the software 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 codelet 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 codelet started, packets can be injected via the ATM equipment and will be received on the Ethernet test equipment.

Router Example

The Fast Path codelet initializes the software 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 AAL 5 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 four IP flows per VC and this is shown in this codelet.

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 codelet starts, packets can be injected via the ATM equipment and will be received on the Ethernet test equipment.

This codelet is not supported in little-endian mode.

IxFpathAccCodelet makes use of the following access layer components:

- IxFpathAcc
- IxEthAcc
- IxAtmdAcc

21.8 HSS Access Codelet (IxHssAccCodelet)

IxHssAccCodelet tests packetized and channelized services, with the codelet acting as data source/sink and HSS as loopback. The codelet will transmit data and will optionally verify that data received is the same as that transmitted.

Codelet runs for a user selectable amount of time. This codelet provides a good example of different Intel XScale core-to-NPE data transfer techniques, by using mbuf pools for packetized services and circular buffers for channelized services.

21.9 Performance Profiling Codelet (IxPerfProfAccCodelet)

IxPerfProfAccCodelet is a useful utility that demonstrates how to access performance related data provided by IxPerfProfAcc. The codelet provides an interface to view north, south, and SDRAM bus activity, event counting and idle cycles from the Intel XScale core PMU and other performance attributes of the processor.

This codelet is not supported in little-endian mode.

21.10 USB RNDIS Codelet (IxUSBRNDIS)

The IxUSBRNDIS codelet is a sample VxWorks* END driver implementation of an RNDIS client.

RNDIS (Remote Network Driver Interface Specification) is a specification for Ethernet-like interface compatible with Microsoft* operating systems. This codelet allows a properly configured IXP42X product line processor running VxWorks to communicate IP traffic over USB to a Microsoft Windows* system.

Operating System Abstraction Layers22

22.1 Overview

There are two operating-system abstraction layers provided as part of the Intel® IXP400 Software v.1.3 architecture.

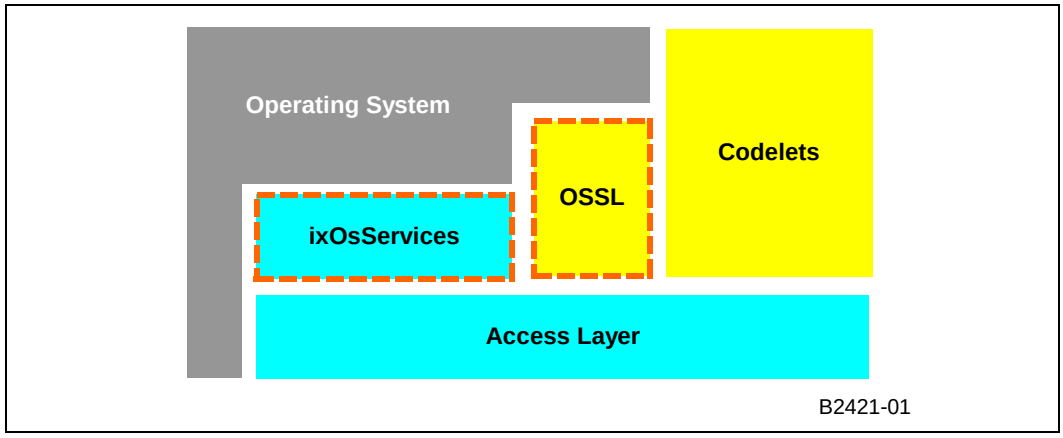
The OsServices layer provides a very thin set of abstracted operating-system (OS) services. All other access-layer components abstract their OS dependencies to this module.

The Operating System Service Layer (OSSL) defines an extended, more fully featured interface for operating system services.

Though primarily intended for use by the software release 1.3 access layer, these services are also made available to the codelets and to application layer software.

There is some overlap in the functionality provided in the OsServices layer and OSSL layer. Where possible, it is recommended that the software release 1.3 developer use an OsServices layer API as opposed to a corresponding, similar OSSL function.

Figure 79. Intel® IXP400 Software v.1.3 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.

22.2 What's New

Changes since software release version 1.1 include:

- **Cache aligned memory management—** ixOsServCacheDmaAlloc and ixOsServCacheDmaFree APIs have been added.
- **Physical to virtual memory conversions—** IX_OSSERV_MEM_MAP and IX_OSSERV_MEM_UNMAP APIs have been added.

22.3 ixOsServices

The `IxOServices.h` file contains the API to the functions that 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.

22.3.1 Mutual Exclusion Services

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.
- **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.

  If there is task contention for a resource, a mutex lock will suspend the current task. Once the contention disappears, the lock and will automatically resume the task. 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 caller is notified without being suspended. The caller has the opportunity to retry to obtain the lock or to abort.

  The fast semaphore lock is very simple and very efficient. If resource contention is expected to be infrequent, this type of lock should be used.

22.3.2 Trace Services

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*.

22.3.3 Memory Services

The memory services provide application access to physical device memory and support DMA setup.

The memory services include

- Application memory allocation suitable for DMA descriptor layout
- Application access to memory mapped hardware devices

22.3.4 Timer Services

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

22.3.5 ixOsServices Functions

The ixOsServices layer implements 17 functions. Basic ixOsServices provided are shown in Table 50.

Table 50. Intel® IXP400 Software v.1.3 ixOsServices Function Descriptions (Sheet 1 of 2)

	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.

Table 50. Intel® IXP400 Software v.1.3 ixOsServices Function Descriptions (Sheet 2 of 2)

	Function	Description
Memory	ixOsServCacheDmaAlloc	Allocates cache aligned memory.
	ixOsServCacheDmaFree	De-allocates memory allocated with ixOsServCacheDmaAlloc.
	IX_OSSERV_MEM_MAP	Allocates an application visible address for a given physical address.
	IX_OSSERV_MEM_UNMAP	De-allocates an application visible address.

22.4 OSSL

The Operating System Services Layer (OSSL) defines a portable interface for operating-system services. This OSSL abstraction layer is a set of functions that provide further 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

22.4.1 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.

22.4.2 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.

Counting semaphores are not supported.

22.4.3 Mutual Exclusion

A critical section is a part of the program that needs exclusive access to a shared resource such as a buffer or an I/O device. Mutual exclusion (mutex) is used to guard a critical section

A critical section must be protected by a mutex for correct behavior, otherwise corrupted data is likely. Prior to the first execution of a critical section, a mutex should be allocated using `ix_ossl_mutex_init` service. When any thread wants to access the critical section, it must first lock the mutex for that critical section. When the thread is finished with execution in the critical section, it unlocks the mutex thus allowing another thread to execute in the critical section.

22.4.4 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.

22.4.5 Timers

OSSL timers provide primitives to get system time and cause thread delays. System time is provided in nanosecond-level resolution.

22.4.6 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; i.e. dynamic memory must be de-allocated using `ix_ossl_free` when it is no longer required.

22.4.7 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. `ix_ossl_message_log` is used to log error messages.

22.4.8 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 51.

Table 51. Intel® IXP400 Software v.1.3 ixOSSL Function Descriptions

	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.

22.5 Software Component Dependencies on Operating System Services

Table 52. Access Layer Component OS Service Dependencies (Sheet 1 of 2)

		Access Layer Components															
		ixNpeMh	ixNpeDl	ixQmgr	ixAtmdAcc	ixHssAcc	ixEthAcc	ixEthDB	ixFPathAcc	ixCryptoAcc	ixDmaAcc	ixUsbAcc	ixAdsl	ixUartAcc	ixPerfProfAcc	ixTimerCtrl	ixFeatureCtrl
ixOsServices	ixOsServIntBind	X										X			X		
	ixOsServIntUnBind														X		
	ixOsServIntLock			X		X	X	X		X	X	X			X		
	ixOsServIntUnLock			X		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					X				
	ixOsServFastMutexTryLock			X				X					X				
	ixOsServFastMutexUnlock			X				X					X				
	ixOsServLog	X	X	X	X	X	X	X	X	X	X				X	X	X
	ixOsServLogLevelSet					X			X								
	ixOsServSleep	X							X			X					
	ixOsServTaskSleep						X	X	X	X			X				
	ixOsServTimestampGet																
	ixOsServCacheDmaAlloc				X	X		X	X		X						
	ixOsServCacheDmaFree																
	IX_OSSERV_MEM_MAP	X	X	X			X						X				X
	IX_OSSERV_MEM_UNMAP	X		X			X						X				

Table 52. Access Layer Component OS Service Dependencies (Sheet 2 of 2)

		Access Layer Components														
		ixNpeMh	ixNpeDl	ixQmgr	ixAtmdAcc	ixHssAcc	ixEthAcc	ixEthDB	ixFPathAcc	ixCryptoAcc	ixDmaAcc	ixUsbAcc	ixAdsl	ixUartAcc	ixPerfProfAcc	ixTimerCtrl	ixFeatureCtrl
OSSL	ix_ossl_thread_create							X					X		X	X	
	ix_ossl_thread_get_id														X		
	ix_ossl_thread_exit																
	ix_ossl_thread_kill																
	ix_ossl_thread_set_priority														X		
	ix_ossl_sem_init							X								X	
	ix_ossl_sem_fini																
	ix_ossl_sem_take							X								X	
	ix_ossl_sem_give							X								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															X	
	ix_ossl_tick_get															X	
	ix_ossl_free																
	ix_ossl_memcpy																
	ix_ossl_memset																
	ix_ossl_message_log_init																
	ix_ossl_message_log																

ADSL Driver for the Intel® IXDP425 / IXCDP1100 Development Platform 23

This chapter describes the ADSL driver for the IXDP425 / IXCDP1100 platform that supports the STMicroelectronics* (formally Alcatel*) MTK-20150 ADSL chipset in the ADSL Termination Unit-Remote (ATU-R) mode of operation.

The ADSL driver is provided as part of the Intel® IXP400 Software and is only supported when the processor is in big endian mode.

23.1 What's New

The following major change have been implemented since the Intel® IXP400 Software v1.1:

- The driver no longer supports the SNMP/MIB interface.

23.2 Device Support

- STMicroelectronics MTK-20150 on the IXDP425 / IXCDP1100 platform. The MTK-20150 chipset is made up of MTC-20154 integrated analog front end and the MTC-20156 DMT/ ATM digital modem and ADSL transceiver controller.

23.3 ADSL Driver Overview

The two main interfaces to the ADSL chipset are the parallel CTRL-E interface (via the IXP42X product line processor's expansion bus) and the ATM UTOPIA data path interface (via the IXP42X product line's UTOPIA interface).

The ADSL driver only supports communication with the ADSL chipset via the CTRL-E interface. All data path communication (ATM UTOPIA) must be performed via the ATM Access Layer component of the IXP400 software.

The driver uses the CTRL-E interface to download the STMicroelectronics firmware, configure and monitor the status of the ADSL chipset. The advantage of downloading the firmware via the CTRL-E interface is that it removes the requirement for a separate flash for the STMicroelectronics ADSL chipset.

The driver provides an API to bring the ADSL line up in ATU-R mode. The line is configured to negotiate the best possible line rate, given the conditions of the local loop when the line is opened. The line rate is not renegotiated once the modems are in the "show-time" mode.

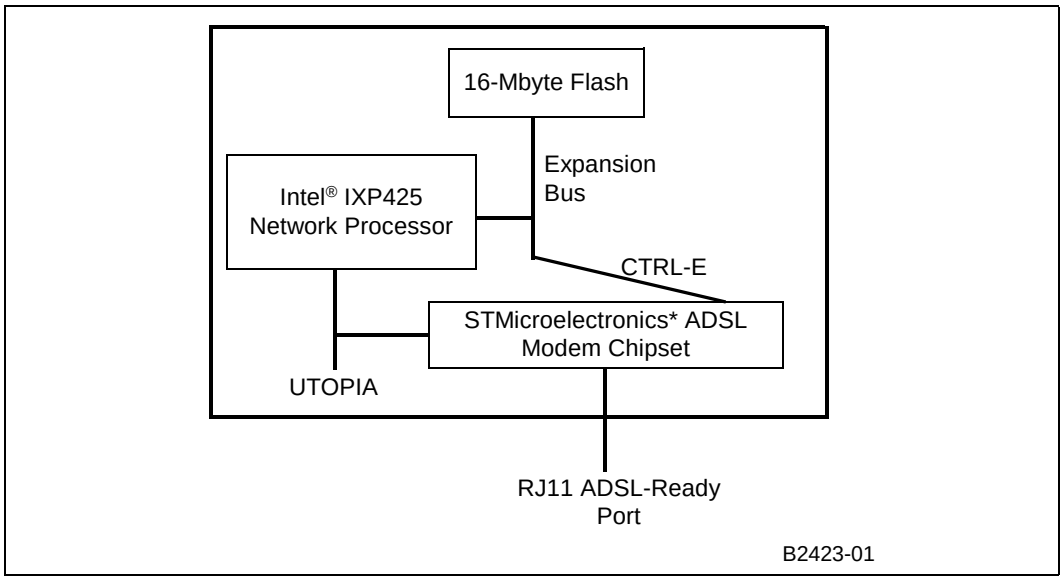
There is very little configuration information required to open an ATU-R line. Almost all line configuration parameters are supplied by the ATU-C side.

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.

23.3.1 Controlling STMicroelectronics* ADSL Modem Chipset Through CTRL-E

The STMicroelectronics ADSL chipset CTRL-E interface is memory-mapped into the processor’s expansion bus address space. Figure 80 shows how the chipset is connected to the processor.

Figure 80. STMicroelectronics* ADSL Chipset on the Intel® IXDP425 / IXCDP1100 Development Platform



The CTRL-E interface is used for all non-data-path communication between the processor and the ADSL chipset. The ADSL driver public APIs use private driver utilities to convert client requests into CTRL-E commands to the ADSL chipset.

23.4 ADSL API

The ADSL driver provides a number of API that provide several general types of functionality. The details of these APIs can be found in Appendix A, “Application Programming Interfaces.” APIs are provided in the following areas:

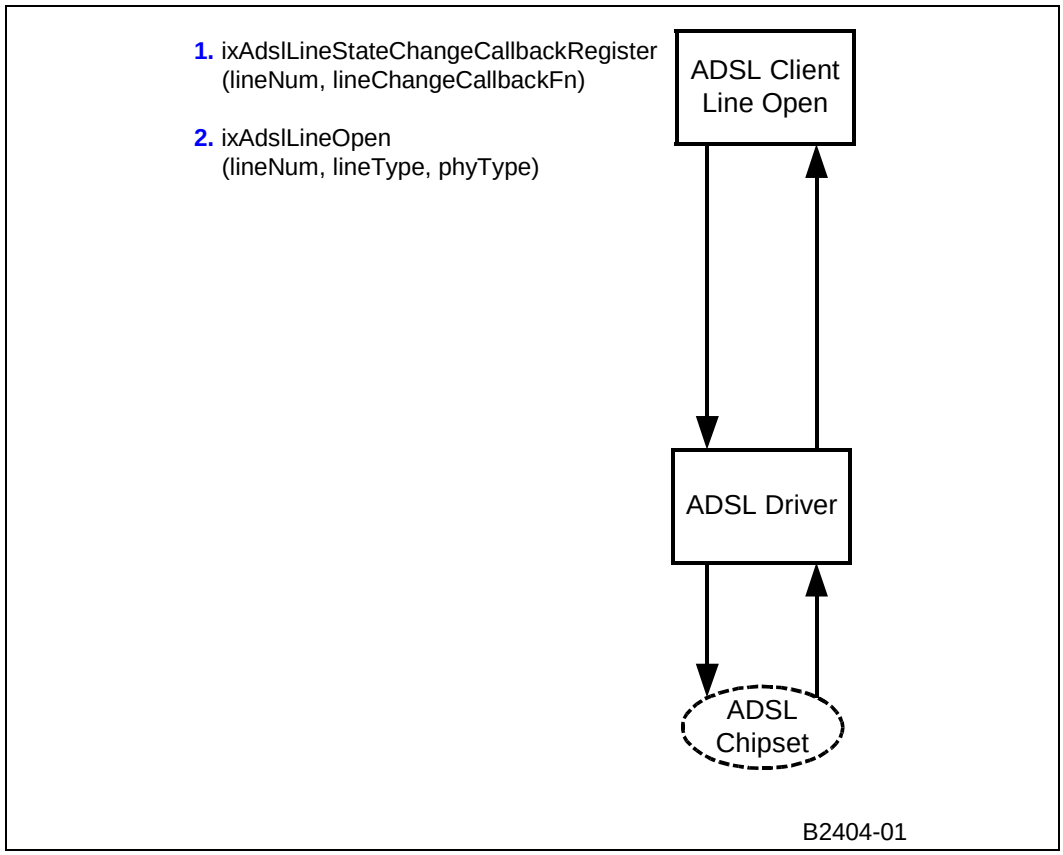
- Firmware download to the ADSL chipset
- Initialization of the ADSL devices
- Opening, closing and monitoring an ADSL line.
- Soft reset

23.5 ADSL Line Open/Close Overview

Note: Before calling the ADSL driver line open function the ATM Access Layer must be started.

Figure 81 on page 243 provides an example of the ADSL driver functions that the client application code will call to open an ADSL line.

Figure 81. Example of ADSL Line Open Call Sequence



Step 1 of Figure 81 is only required if the client application wants to be notified when a line state changes occurs.

Step 2 of Figure 81 is called by the client application to establish an ATU-R ADSL connection with another modem. This function call performs the following actions within the private context of the ADSL driver:

a. Invokes the private ixAdslDriverInit function which creates an ixAdslLineSupervisoryTask. This task invokes the ixAdslLineStateMachine.

b. Invokes the private ixAdslUtilDeviceDownload function which downloads the STMicroelectronics* ADSL firmware and configures the chipset.

c. Invokes the private ixAdslCtrleEnableModem function which enables the ADSL chipset to start opening the line.

The client application can close an ADSL line by calling the ixAdslLineClose( ) API which will disable the modem (i.e. close the line) but not kill the ixAdslLineSupervisoryTask.

23.6 Limitations and Constraints

- The driver is only supported when the processor is operating in big-endian mode.
- The driver only supports the ATU-R mode of operation.
- The driver can operate in single PHY mode only.

Application Programming Interfaces A

This appendix documents the Intel® IXP400 Software v.1.3's access-layer APIs.

This reference chapter was generated automatically using the most recent source code available when this document was generated, and is useful for reference purposes. Be advised that the software may have been subsequently updated and the user should use the source code for the most accurate reference information.

A.1 Access-Layer API and Codelet Module Index

Access-Layer API and Codelet Module Index 245
IXP425 Operating System Services Library (IxOSSL) API 246
IXP425 ADSL Driver API 268
IXP425 Assertion Macros (IxAssert) API 278
IXP425 ATM Driver Access (IxAtmdAcc) API 278
IXP425 ATM Driver Access (IxAtmdAcc) Control API 297
IXP425 ATM Driver Access (IxAtmdAcc) UTOPIA Control API 315
IXP425 ATM Manager (IxAtmm) API 317
IXP425 ATM Transmit Scheduler (IxAtmSch) API 329
IXP425 ATM Types (IxAtmTypes) 336
IXP425 Security (IxCryptoAcc) API 341
IXP425 DMA Types (IxDmaTypes) 356
IXP425 DMA Access Driver (IxDmaAcc) API 359
IXP425 Ethernet Access (IxEthAcc) API. 361
IXP425 Ethernet Access Fast Path (IxEthAccFpathDep) API 391
IXP425 Ethernet Database (IxEthDB) API 392
IXP425 Ethernet Database Port Definitions (IxEthDBPortDefs). 401
IXP425 Ethernet PHY Access (IxEthMii) API 403
IXP425 Ethernet NPE (IxEthNpe) API 407
IXP425 Feature Control (IxFeatureCtrl) API 418
Software Configuration for Access Component. 430
IXP425 Fast Path Access (IxFpathAcc) API 431
IXP425 Fast Path Classifier Opcode Definitions 440
IXP425 Fast Path Modifier Template. 443
IXP425 HSS Access (IxHssAcc) API 444
IXP425 NPE-A (IxNpeA) API 465
IXP425 NPE-Downloader (IxNpeDl) API 481
IXP425 NPE Image ID Definition 495
IXP425 NPE Message Handler (IxNpeMh) API 501
IXP425 OS Memory Buffer Management (IxOsBuffMgt) API. 509
IXP425 OS Memory Buffer Pool Management (IxOsBuffPoolMgt) API 510
IXP425 OS Cache MMU (IxOsCacheMMU) API. 521
IXP425 OS Services (IxOsServices) API. 527
IXP425 Performance Profiling (IxPerfProfAcc) API. 539
IXP425 Queue Manager (IxQMgr) API. 567

IXP425 Timer Control (IxTimerCtrl) API 593
IXP425 Types (IxTypes) 597
IXP425 UART Access (IxUARTAcc) API 598
Defines for Default Values 601
Defines for IOCTL Commands 601
Defines for IOCTL Arguments 603
IXP400 Version ID (IxVersionId) 605
IXP425 USB Driver Public API 606

A.2 IXP425 Operating System Services Library (IxOSSL) API

IXP425 Operating System Services Library (IxOSSL) API. IXP425 Operating System Services Library (IxOSSL) APIIXP425 Operating System Services Library (IxOSSL) APIIXP425 Operating System Services Library (IxOSSL) API. This service provides a layer of OS dependency services.

A.2.0.1 Data Structures

- struct ix_ossl_thread_main_info_t
  This type defines thread main info.
- struct ix_ossl_time_t
  This type defines OSSL time.

A.2.0.2 Defines

- #define IX_OSSL_ERROR_SUCCESS (ix_error)0UL
  This symbol defines an error token that indicates the successful completion of the OSSL calls.
- #define IX_OSSL_WAIT_FOREVER OS_WAIT_FOREVER
  This symbol is useful for specifying an 'indefinite wait' timeout value in ix_ossl_sem_take and ix_ossl_mutex_lock function calls.
- #define IX_OSSL_WAIT_NONE OS_WAIT_NONE
  This symbol is useful for specifying a 'no wait' timeout value in ix_ossl_sem_take and ix_ossl_mutex_lock function calls.
- #define BILLION 1000000000
  Define a constant for the value 1 billion, used by OSSL TIME macros. Equivalent to (1000 million nanoseconds / second).
- #define IX_OSSL_TIME_EQ(a, b) ((a).tv_sec == (b).tv_sec && (a).tv_nsec == (b).tv_nsec)
  Compares a operand with b operand. Returns true if they are equal and false otherwise.
- #define IX_OSSL_TIME_GT(a, b)
  Compares a operand with b operand. Returns true if a > b, and false otherwise.
- #define IX_OSSL_TIME_LT(a, b)
  Compares a operand with b operand. Returns true if a < b, and false otherwise.

- #define IX_OSSL_TIME_ISZERO(a) ((a).tv_sec == 0 && (a).tv_nsec == 0)
  This macro checks if the operand a is zero. Returns true if a is zero (both sec and nsec fields must be zero) and false otherwise.
- #define IX_OSSL_TIME_SET(a, b) (a).tv_sec = (b).tv_sec; (a).tv_nsec = (b).tv_nsec
  This macro sets operand a to the value of operand b.
- #define IX_OSSL_TIME_ADD(a, b)
  This macro performs a += b operation.
- #define IX_OSSL_TIME_SUB(a, b)
  This macro performs a -= b operation.
- #define IX_OSSL_TIME_NORMALIZE(a)
  This macro normalizes the value of a. If 'a.nsec' > 10^9, it is decremented by 10^9 and 'a.sec' is incremented by 1.
- #define IX_OSSL_TIME_VALID(a) ((a).tv_nsec >= 0 && (a).tv_nsec < BILLION)
  This macro checks whether a is a valid ix_ossl_time_t i.e. 0 =< a.nsec < 10^9. Returns true if a is valid and false otherwise.
- #define IX_OSSL_TIME_ZERO(a) (a).tv_sec = 0; (a).tv_nsec = 0
  This macro sets the value of a to zero.
- #define IX_OSSL_TIME_CONVERT_TO_TICK(a, b) TV_CONVERT_TO_TICK(a,b)
  This macro converts b value in ix_ossl_time_t to a value in os ticks.

A.2.0.3 Typedefs

- typedef os_thread_t ix_ossl_thread_t
  This type defines OSSL thread type.
- typedef os_sem_t ix_ossl_sem_t
  This type defines OSSL semaphore type.
- typedef os_mutex_t ix_ossl_mutex_t
  This type defines OSSL mutex type.
- typedef ix_error(* ix_ossl_thread_entry_point_t)(void *arg, void **ptrRetObj)
  This function type defines OSSL thread entry point function. .
- typedef unsigned int ix_ossl_size_t
  ix_ossl_size_t This type describes a generic size type.

A.2.0.4 Enumerations

- enum ix_ossl_error_code { IX_OSSL_ERROR_SUCCESS = 0,
  IX_OSSL_ERROR_INVALID_ARGUMENTS = OS_INVALID_ATTR,
  IX_OSSL_ERROR_INVALID_OPERATION,
  IX_OSSL_ERROR_THREAD_CALL_FAILURE, IX_OSSL_ERROR_INVALID_PID,
  IX_OSSL_ERROR_INVALID_TID, IX_OSSL_ERROR_OS_CALL_FAILURE =
  OS_FAILURE, IX_OSSL_ERROR_TIMEOUT = OS_TIMEDOUT,
  IX_OSSL_ERROR_NOMEM = OS_NOMEM, IX_OSSL_ERROR_NOSYS = OS_NOSYS
  }
  This type defines error codes returned by OSSL calls.

- enum ix_ossl_sem_state { IX_OSSL_SEM_UNAVAILABLE = OS_SEM_UNVAILABLE, IX_OSSL_SEM_AVAILABLE = OS_SEM_AVAILABLE }
  This type defines OSSL binary semaphore states.
- enum ix_ossl_mutex_state { IX_OSSL_MUTEX_UNLOCK = OS_MUTEX_UNLOCK, IX_OSSL_MUTEX_LOCK = OS_MUTEX_LOCK }
  This type defines OSSL mutex states.
- enum ix_ossl_thread_priority { IX_OSSL_THREAD_PRI_HIGH = OS_THREAD_PRI_HIGH, IX_OSSL_THREAD_PRI_MEDIUM = OS_THREAD_PRI_MEDIUM, IX_OSSL_THREAD_PRI_LOW = OS_THREAD_PRI_LOW }
  This type define OSSL thread priority levels.

A.2.0.5 Functions

- IX_EXPORT_FUNCTION ix_error ix_ossl_thread_create (ix_ossl_thread_entry_point_t entryPoint, void *arg, ix_ossl_thread_t *ptrTid)
  create a thread
- IX_EXPORT_FUNCTION ix_error ix_ossl_thread_get_id (ix_ossl_thread_t *ptrTid)
  get id of calling thread
- IX_EXPORT_FUNCTION void * ix_ossl_thread_main_wrapper (void *ptrThreadInfo)
  wrapper for user-provided thread function
- IX_EXPORT_FUNCTION void ix_ossl_thread_exit (ix_error retError, void *retObj)
  Causes the calling thread to exit.
- IX_EXPORT_FUNCTION ix_error ix_ossl_thread_kill (ix_ossl_thread_t tid)
  kills the specified thread
- IX_EXPORT_FUNCTION ix_error ix_ossl_thread_set_priority (ix_ossl_thread_t tid, ix_ossl_thread_priority priority)
  sets the priority of the specified thread
- IX_EXPORT_FUNCTION ix_error ix_ossl_thread_delay (int ticks)
  delay the current task for specified number of OS ticks
- IX_EXPORT_FUNCTION int os_thread_create (void *(*start_routine)(void *), ix_ossl_thread_main_info_t *ptrThreadInfo, ix_ossl_thread_t *ptrTid, os_error *osError)
  creates a thread (use ix_ossl_thread_create instead)
- IX_EXPORT_FUNCTION int os_thread_get_id (ix_ossl_thread_t *ptrTid)
  get thread id (use ix_ossl_thread_get_id instead)
- IX_EXPORT_FUNCTION int os_thread_exit (void *ptrRetObj)
  exits the calling thread (use ix_ossl_thread_exit instead)
- IX_EXPORT_FUNCTION int os_thread_kill (ix_ossl_thread_t tid, os_error *osError)
  kill the specified thread (use ix_ossl_thread_kill instead)
- IX_EXPORT_FUNCTION ix_error os_thread_set_priority (ix_ossl_thread_t *tid, ix_ossl_thread_priority priority, os_error *osError)
  sets priority of a thread (use ix_ossl_thread_set_priority instead)

- IX_EXPORT_FUNCTION ix_error ix_ossl_tick_get (int *pticks)
  gets number of OS ticks per second
- IX_EXPORT_FUNCTION int os_sleep (ix_uint32 sleeptime_ms, os_error *osError)
  causes the calling thread to sleep for specified time (milliseconds)
- IX_EXPORT_FUNCTION int os_sleep_tick (ix_uint32 sleeptime_ticks, os_error *osError)
  causes the calling thread to sleep for specified time (os ticks)
- IX_EXPORT_FUNCTION int os_time_get (ix_ossl_time_t *ptime, os_error *osError)
  returns the system time with nano-second resolution
- IX_EXPORT_FUNCTION ix_error ix_ossl_sem_init (int start_value, ix_ossl_sem_t *sid)
  initialises a new semaphore
- IX_EXPORT_FUNCTION ix_error ix_ossl_sem_take (ix_ossl_sem_t sid, ix_uint32 timeout)
  take a semaphore, and block if semaphore not available
- IX_EXPORT_FUNCTION ix_error ix_ossl_sem_give (ix_ossl_sem_t sid)
  give back a semaphore
- IX_EXPORT_FUNCTION ix_error ix_ossl_sem_flush (ix_ossl_sem_t sid, int *result)
  unblocks all threads pending on the semaphore
- IX_EXPORT_FUNCTION ix_error ix_ossl_sem_fini (ix_ossl_sem_t sid)
  terminate the semaphore (free semaphore resources)
- IX_EXPORT_FUNCTION int os_thread_sema_create (int value, ix_ossl_sem_t *sid, os_error *osError)
  create a thread semaphore
- IX_EXPORT_FUNCTION int os_thread_sema_P (ix_ossl_sem_t *sid, ix_uint32 timeout, os_error *osError)
  pend on a semaphore (with timeout)
- IX_EXPORT_FUNCTION int os_thread_sema_V (ix_ossl_sem_t *sid, os_error *osError)
  release a semaphore
- IX_EXPORT_FUNCTION int os_thread_sema_destroy (ix_ossl_sem_t *sid, os_error *osError)
  destroy semaphore object
- IX_EXPORT_FUNCTION ix_error ix_ossl_mutex_init (ix_ossl_mutex_state start_state, ix_ossl_mutex_t *mid)
  initialises a new mutex object
- IX_EXPORT_FUNCTION ix_error ix_ossl_mutex_lock (ix_ossl_mutex_t mid, ix_uint32 timeout)
  lock a mutex
- IX_EXPORT_FUNCTION ix_error ix_ossl_mutex_unlock (ix_ossl_mutex_t mid)
  unlocks a mutex
- IX_EXPORT_FUNCTION ix_error ix_ossl_mutex_fini (ix_ossl_mutex_t mid)
  free a mutex
- IX_EXPORT_FUNCTION int os_thread_mutex_create (ix_ossl_mutex_state start_state, ix_ossl_mutex_t *mid, os_error *osError)

creates a thread mutex object

- IX_EXPORT_FUNCTION int os_thread_mutex_lock (ix_ossl_mutex_t *mutex, ix_uint32 timeout, os_error *osError)
  lock a mutex
- IX_EXPORT_FUNCTION int os_thread_mutex_unlock (ix_ossl_mutex_t *mutex, os_error *osError)
  unlock a mutex
- IX_EXPORT_FUNCTION int os_thread_mutex_destroy (ix_ossl_mutex_t *mutex, os_error *osError)
  destroy a mutex object
- ix_error ix_ossl_sleep (ix_uint32 sleeptime_ms)
  causes calling thread to sleep for specified time (milliseconds)
- IX_EXPORT_FUNCTION ix_error ix_ossl_sleep_tick (ix_uint32 sleeptime_ticks)
  causes calling thread to sleep for specified time (os ticks)
- IX_EXPORT_FUNCTION ix_error ix_ossl_time_get (ix_ossl_time_t *ptime)
  gets current system time with nano-second resolution
- IX_EXPORT_FUNCTION void * ix_ossl_malloc (ix_ossl_size_t arg_Size)
  allocate a block of memory
- IX_EXPORT_FUNCTION void ix_ossl_free (void *arg_pMemory)
  free a block of memory
- IX_EXPORT_FUNCTION void * ix_ossl_memcpy (void *arg_pDest, const void *arg_pSrc, ix_ossl_size_t arg_Count)
  copy number of bytes from one memory location to another
- IX_EXPORT_FUNCTION void * ix_ossl_memset (void *arg_pDest, int arg_pChar, ix_ossl_size_t arg_Count)
  fill a region of memory with a specified byte value
- IX_EXPORT_FUNCTION ix_error ix_ossl_message_log_init (void)
  initialise the error message logging facility
- IX_EXPORT_FUNCTION ix_error ix_ossl_message_log (char *arg_pFmtString,...)
  log a printf-style message

A.2.0.6 Detailed Description

This service provides a layer of OS dependency services.

This file contains the prototypes of OS-independent wrapper functions which allow the programmer not to be tied to a specific operating system. The OSSL functions can be divided into three classes:

- Synchronization-related wrapper functions around thread system calls
- Thread-related wrapper functions around thread calls
- Transactor/workbench osapi calls -- defined in osApi.h

Both 1 and 2 classes of functions provide Thread Management, Thread Synchronization, Mutual Exclusion and Timer primitives. Namely, creation and deletion functions as well as the standard "wait" and "exit". Additionally, a couple of utility functions which enable to pause the execution of a thread are also provided.

The 3rd class provides a slew of other OSAPI functions to handle Transactor/WorkBench OS calls.

Note: Wherever possible, please use the equivalent API function from the IXP425 OS Services (IxOsServices) API component instead if one exists.

A.2.0.7 Define Documentation

A.2.0.8 #define BILLION 1000000000

Define a constant for the value 1 billion, used by OSSL TIME macros. Equivalent to (1000 million nanoseconds / second).

Definition at line 393 of file ix_ossl.h.

A.2.0.9 #define IX_OSSL_ERROR_SUCCESS (ix_error)0UL

This symbol defines an error token that indicates the successful completion of the OSSL calls.

Definition at line 339 of file ix_ossl.h.

A.2.0.10 #define IX_OSSL_TIME_ADD(a, b)

Value

```
(a).tv_sec += (b).tv_sec;   \
        (a).tv_nsec += (b).tv_nsec; \
        if ((a).tv_nsec >= BILLION) \
    { \
        (a).tv_sec++; \
        (a).tv_nsec -= BILLION; }
```

This macro performs a += b operation.

Parameters

- *a* ix_ossl_time_t (in|out) - operand a
- *b* ix_ossl_time_t (in) - operand b
  Definition at line 486 of file ix_ossl.h.

A.2.0.11 #define IX_OSSL_TIME_CONVERT_TO_TICK(a, b) TV_CONVERT_TO_TICK(a,b)

This macro converts b value in ix_ossl_time_t to a value in os ticks.

Parameters

- *a* unsigned int (out) - operand a
- *b* ix_ossl_time_t (in) - operand b

Definition at line 573 of file ix_ossl.h.

A.2.0.12 #define IX_OSSL_TIME_EQ(a, b) ((a).tv_sec == (b).tv_sec && (a).tv_nsec == (b).tv_nsec)

Compares a operand with b operand. Returns true if they are equal and false otherwise.

Parameters

- *a* ix_ossl_time_t (in) - operand a
- *b* ix_ossl_time_t (in) - operand b

Returns

- true if a == b and false otherwise.
  Definition at line 417 of file ix_ossl.h.

A.2.0.13 #define IX_OSSL_TIME_GT(a, b)

Value

```
((a).tv_sec  > (b).tv_sec ||    \
        ((a).tv_sec == (b).tv_sec && (a).tv_nsec > (b).tv_nsec))
```

Compares a operand with b operand. Returns true if a > b, and false otherwise.

Parameters

- *a* ix_ossl_time_t (in) - operand a
- *b* ix_ossl_time_t (in) - operand b

Returns

true if a > b and false otherwise.

Definition at line 432 of file ix_ossl.h.

A.2.0.14 #define IX_OSSL_TIME_ISZERO(a) ((a).tv_sec == 0 && (a).tv_nsec == 0)

This macro checks if the operand a is zero. Returns true if a is zero (both sec and nsec fields must be zero) and false otherwise.

Parameters

a ix_ossl_time_t (in) - operand a

Returns

true if a is zero and false otherwise.

Definition at line 462 of file ix_ossl.h.

A.2.0.15 #define IX_OSSL_TIME_LT(a, b)

Value

```
((a).tv_sec  < (b).tv_sec ||    \
        ((a).tv_sec == (b).tv_sec && (a).tv_nsec < (b).tv_nsec))
```

Compares a operand with b operand. Returns true if a < b, and false otherwise.

Parameters

- *a* ix_ossl_time_t (in) - operand a
- *b* ix_ossl_time_t (in) - operand b

Returns

true if a < b and false otherwise.

Definition at line 447 of file ix_ossl.h.

A.2.0.16 #define IX_OSSL_TIME_NORMALIZE(a)

Value

```
if ((a).tv_nsec >= BILLION) \
        { (a).tv_sec++; (a).tv_nsec -= BILLION; }   \
    else if ((a).tv_nsec < 0)   \
        { (a).tv_sec--; (a).tv_nsec += BILLION; }
```

This macro normalizes the value of a. If 'a.nsec' > 10^9, it is decremented by 10^9 and 'a.sec' is incremented by 1.

Parameters

a ix_ossl_time_t (in|out) - operand a

Definition at line 524 of file ix_ossl.h.

A.2.0.17 #define IX_OSSL_TIME_SET(a, b) (a).tv_sec = (b).tv_sec; (a).tv_nsec = (b).tv_nsec

This macro sets operand a to the value of operand b.

Parameters

- *a* ix_ossl_time_t (out) - operand a
- *b* ix_ossl_time_t (in) - operand b

Definition at line 474 of file ix_ossl.h.

A.2.0.18 #define IX_OSSL_TIME_SUB(a, b)

Value

```
if ((a).tv_nsec >= (b).tv_nsec) \
        { \
          (a).tv_sec -= (b).tv_sec; \
          (a).tv_nsec -= (b).tv_nsec; \
```

```
} \
else \
{ \
  (a).tv_sec -= ((b).tv_sec + 1); \
  (a).tv_nsec += BILLION - (b).tv_nsec; \
}
```

This macro performs a -= b operation.

Parameters

- *a* ix_ossl_time_t (in|out) - operand a
- *b* ix_ossl_time_t (in) - operand b
  Definition at line 503 of file ix_ossl.h.

A.2.0.19 #define IX_OSSL_TIME_VALID(a) ((a).tv_nsec >= 0 && (a).tv_nsec < BILLION)

This macro checks whether a is a valid ix_ossl_time_t i.e. 0 =< a.nsec < 10^9. Returns true if a is valid and false otherwise.

Parameters

- *a* ix_ossl_time_t (in) - operand a

Returns

true if a is valid ix_ossl_time_t and false otherwise

Definition at line 542 of file ix_ossl.h.

A.2.0.20 #define IX_OSSL_TIME_ZERO(a) (a).tv_sec = 0; (a).tv_nsec = 0

This macro sets the value of a to zero.

Parameters

a ix_ossl_time_t (in|out) - operand a

Definition at line 554 of file ix_ossl.h.

A.2.0.21 Typedef Documentation

A.2.0.22 ix_ossl_thread_entry_point_t

This function type defines OSSL thread entry point function. .

Parameters

- *void* * arg (in) - pointer to a custom thread argument
- *void* ** ptrRetObj (out) - address where a pointer to a custom data structure or object will be returned by the thread on exit.
  Definition at line 244 of file ix_ossl.h.

A.2.0.23 Enumeration Type Documentation

A.2.0.24 enum ix_ossl_error_code

This type defines error codes returned by OSSL calls.

Enumeration Values

- ***IX_OSSL_ERROR_SUCCESS*** success
- ***IX_OSSL_ERROR_INVALID_ARGUMENTS*** invalid arguments
- ***IX_OSSL_ERROR_INVALID_OPERATION*** invalid operation
- ***IX_OSSL_ERROR_THREAD_CALL_FAILURE*** thread operation failed
- ***IX_OSSL_ERROR_INVALID_PID*** invalid process id
- ***IX_OSSL_ERROR_INVALID_TID*** invalid thread id
- ***IX_OSSL_ERROR_OS_CALL_FAILURE*** invalid arguments
- ***IX_OSSL_ERROR_TIMEOUT*** OS operation failed.
- ***IX_OSSL_ERROR_NOMEM*** memory unavailable
- ***IX_OSSL_ERROR_NOSYS*** system resource unavailable

Definition at line 274 of file ix_ossl.h.

A.2.0.25 enum ix_ossl_mutex_state

This type defines OSSL mutex states.

Enumeration Values

- ***IX_OSSL_MUTEX_UNLOCK*** Mutex unlocked.
- ***IX_OSSL_MUTEX_LOCK*** Mutex locked.

Definition at line 309 of file ix_ossl.h.

A.2.0.26 enum ix_ossl_sem_state

This type defines OSSL binary semaphore states.

Enumeration Values

- ***IX_OSSL_SEM_UNAVAILABLE*** Semaphore unavailable.
- ***IX_OSSL_SEM_AVAILABLE*** Semaphore available.

Definition at line 295 of file ix_ossl.h.

A.2.0.27 enum ix_ossl_thread_priority

This type define OSSL thread priority levels.

Enumeration Values

- ***IX_OSSL_THREAD_PRI_HIGH*** High priority thread.

- ***IX_OSSL_THREAD_PRI_MEDIUM*** Medium priority thread.
- ***IX_OSSL_THREAD_PRI_LOW*** Low priority thread.
  Definition at line 322 of file ix_ossl.h.

A.2.0.28 Function Documentation

A.2.0.29 IX_EXPORT_FUNCTION void ix_ossl_free (void * *arg_pMemory*)

free a block of memory

This function will free a memory block specified by the passed address. The ix_ossl_free function deallocates a memory block (arg_pMemory) that was previously allocated by a call to ix_ossl_malloc.If arg_pMemory is NULL, the pointer is ignored and ix_ossl_free immediately returns. Attempting to free an invalid pointer (a pointer to a memory block that was not allocated by ix_ossl_malloc) may affect subsequent allocation requests and cause errors.

See ixOsServCacheDmaFree for an alternative.

Parameters

arg_pMemory void* (in) - address of the memory block to be deallocated.

A.2.0.30 IX_EXPORT_FUNCTION void* ix_ossl_malloc (ix_ossl_size_t *arg_Size*)

allocate a block of memory

This function will allocate a memory block. the function returns a void pointer to the allocated space, or NULL if there is insufficient memory available. To return a pointer to a type other than void, use a type cast on the return value. The storage space pointed to by the return value is guaranteed to be suitably aligned for storage of any type of object. If size is 0, ix_ossl_malloc allocates a zero-length item in the heap and returns a valid pointer to that item. Always check the return from ix_ossl_malloc, even if the amount of memory requested is small.

See ixOsServCacheDmaAlloc for an alternative.

Parameters

arg_Size ix_ossl_size_t (in) - the size of the memory block requested.

Returns

Returns a valid address if successful or a NULL for failure.

A.2.0.31 IX_EXPORT_FUNCTION void* ix_ossl_memcpy (void * *arg_pDest*, const void * *arg_pSrc*, ix_ossl_size_t *arg_Count*)

copy number of bytes from one memory location to another

This function will copy memory bytes between buffers. The ix_ossl_memcpy function copies count bytes of arg_pSrc to arg_pDest. If the source and destination overlap, this function does not ensure that the original source bytes in the overlapping region are copied before being overwritten.

Parameters

- *arg_pDest* void* (in|out) - destination buffer address
- *arg_pSrc* const void* (in) - source buffer address
- *arg_Count* ix_ossl_size_t - number of bytes to copy

Returns

Returns the address of the destination buffer.

A.2.0.32 IX_EXPORT_FUNCTION void* ix_ossl_memset (void * *arg_pDest*, int *arg_pChar*, ix_ossl_size_t *arg_Count*)

fill a region of memory with a specified byte value

This function sets buffers to a specified character. The ix_ossl_memset function sets the first arg_Count bytes of arg_pDest to the character arg_Char.

Parameters

- *arg_pDest* void* (in|out) - destination buffer address
- *arg_pChar* int (in) - character to set
- *arg_Count* ix_ossl_size_t (in) - number of characters to set

Returns

Returns the address of the destination buffer.

A.2.0.33 IX_EXPORT_FUNCTION ix_error ix_ossl_message_log (char * *arg_pFmtString*, ...)

log a printf-style message

This routine is used to log a specified message. This routine's syntax is similar to printf() - a format string is followed by a variable number of arguments which will be interpreted and formated according to the fmt_string passed as the first argument. Further details will be provided on where the messages will be logged!

See ixOsServLog for an alternative.

Parameters

arg_pFmtString char* (in) - format string for the log message

Returns

IX_ERROR_SUCCESS if successful or a valid ix_error token for failure.

A.2.0.34 IX_EXPORT_FUNCTION ix_error ix_ossl_message_log_init (void)

initialise the error message logging facility

This function is used to initialize the error message logging. 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! This function should be called before any call to ix_ossl_message_log().

See ixOsServLogLevelSet for an alternative.

Returns

IX_ERROR_SUCCESS if successful or a valid ix_error token for failure.

A.2.0.35 IX_EXPORT_FUNCTION ix_error ix_ossl_mutex_fini (ix_ossl_mutex_t *mid*)

free a mutex

This function frees a mutex.'mid' is the id mutex id. The mutex is deleted, all resources are freed. Any threads pending on this mutex will be unblocked and return an error.

Parameters

mid ix_ossl_mutex_t (in) - mutex id

Returns

IX_OSSL_ERROR_SUCCESS if successful or a valid ix_error token for failure.

A.2.0.36 IX_EXPORT_FUNCTION ix_error ix_ossl_mutex_init (ix_ossl_mutex_state *start_state*, ix_ossl_mutex_t * *mid*)

initialises a new mutex object

This function initializes a new mutex. 'mid' is a pointer to an ix_ossl_mutex_t. Upon success, '*mid' will contain the mutex id. 'start_state' is the initial locking state.

Parameters

- *start_state* ix_ossl_mutex_state (in) - 'start_state' is initial locking state. Valid values are :- 'OSSL_MUTEX_LOCK': Lock the mutex 'OSSL_MUTEX_UNLOCK': Leave the mutex unlocked
- *mid* ix_ossl_mutex_t* (out) - pointer to id of the mutex created

Returns

IX_OSSL_ERROR_SUCCESS if successful or a valid ix_error token for failure.

A.2.0.37 IX_EXPORT_FUNCTION ix_error ix_ossl_mutex_lock (ix_ossl_mutex_t *mid*, ix_uint32 *timeout*)

lock a mutex

This function locks the mutex. If the mutex is already locked, the thread will block. If the time indicated in 'timeout' is reached, the thread will unblock and return error indication. If the timeout is set to 'IX_OSSL_WAIT_NONE', the thread will never block (this is trylock). If the timeout is set to 'IX_OSSL_WAIT_FOREVER', the thread will block until the mutex is unlocked.

Parameters

- *mid* ix_ossl_mutex_t (in) - mutex id.
- *timeout* ix_uint32 (in) - timeout value of type ix_uint32 expressed in miliseconds.

Returns

IX_OSSL_ERROR_SUCCESS if successful or a valid ix_error token for failure.

A.2.0.38 IX_EXPORT_FUNCTION ix_error ix_ossl_mutex_unlock (ix_ossl_mutex_t *mid*)

unlocks a mutex

This function unlocks the mutex. 'mid' is the mutex id. If there are threads pending on the mutex, the next one is given the lock. If there are no pending threads, then the mutex is unlocked.

Parameters

mid ix_ossl_mutex_t (in) - mutex id

Returns

IX_OSSL_ERROR_SUCCESS if successful or a valid ix_error token for failure.

A.2.0.39 IX_EXPORT_FUNCTION ix_error ix_ossl_sem_fini (ix_ossl_sem_t *sid*)

terminate the semaphore (free semaphore resources)

This function frees a semaphore.'sid' is semaphore id. The semaphore is terminated, all resources are freed. The threads pending on this semaphore will be released and return an error.

Parameters

sid ix_ossl_sem_t (in) - semaphore id

Returns

IX_OSSL_ERROR_SUCCESS if successful or a valid ix_error token for failure.

A.2.0.40 IX_EXPORT_FUNCTION ix_error ix_ossl_sem_flush (ix_ossl_sem_t *sid*, int * *result*)

unblocks all threads pending on the semaphore

This function unblocks all pending threads without altering the semaphore count. 'sid' is the id for the semaphore. '*result' will be non-zero if a thread was unblocked during this call.

Parameters

- *sid* ix_ossl_sem_t (in) - semaphore id
- *result* int* (out) - the value referred will be non-zero if a thread was unblocked during this call

Returns

IX_OSSL_ERROR_SUCCESS if successful or a valid ix_error token for failure.

A.2.0.41 IX_EXPORT_FUNCTION ix_error ix_ossl_sem_give (ix_ossl_sem_t *sid*)

give back a semaphore

This function causes the next available thread in the pend queue to be unblocked. If no thread is pending on this semaphore, the semaphore becomes 'full'.

Parameters

sid ix_ossl_sem_t (in) - semaphore id.

Returns

IX_OSSL_ERROR_SUCCESS if successful or a valid ix_error token for failure.

A.2.0.42 IX_EXPORT_FUNCTION ix_error ix_ossl_sem_init (int *start_value*, ix_ossl_sem_t * *sid*)

initialises a new semaphore

This function initializes a new semaphore. 'sid' is a pointer to an ix_ossl_sem_t. Upon success, '*sid' will be the semaphore id used in all other ix_ossl_sem functions. The newly created semaphore will be initialized the value of 'start_value'.

Parameters

- *start_value* int (in) - initial value of the semaphore
- *sid* ix_ossl_sem_t* (out) - Address where the newly created semaphore id will returned.

Returns

IX_OSSL_ERROR_SUCCESS if successful or a valid ix_error token for failure.

A.2.0.43 IX_EXPORT_FUNCTION ix_error ix_ossl_sem_take (ix_ossl_sem_t *sid*, ix_uint32 *timeout*)

take a semaphore, and block if semaphore not available

If the semaphore is 'empty', the calling thread is blocked. If the semaphore is 'full', it is taken and control is returned to the caller. If the time indicated in 'timeout' is reached, the thread will unblock and return an error indication. If the timeout is set to 'IX_OSSL_WAIT_NONE', the thread will never block; if it is set to 'IX_OSSL_WAIT_FOREVER', the thread will block until the semaphore is available.

Parameters

- *sid* ix_ossl_sem_t (in) - semaphore id.
- *timeout* ix_uint32 (in) - timeout value of type ix_uint32 expressed in miliseconds

Returns

IX_OSSL_ERROR_SUCCESS if successful or a valid ix_error token for failure.

A.2.0.44 ix_error ix_ossl_sleep (ix_uint32 *sleeptime_ms*)

causes calling thread to sleep for specified time (milliseconds)

This function causes the calling thread to sleep for the specified time.

Parameters

sleeptime_ms ix_uint32 (in) - sleep time specified in milliseconds.

Returns

IX_OSSL_ERROR_SUCCESS if successful or a valid ix_error token for failure.

: In VxWorks this function has a resolution dictated by sysClkRateGet(). Very small delays could be truncated to 0 i.e. no delay at all. See ixOsServTaskSleep() for an alternative.

A.2.0.45 IX_EXPORT_FUNCTION ix_error ix_ossl_sleep_tick (ix_uint32 *sleeptime_ticks*)

causes calling thread to sleep for specified time (os ticks)

This function causes the calling thread to sleep for the time specified in OS ticks.

Parameters

sleeptime_ticks ix_uint32 (in) - sleep time specified in os ticks.

Returns

IX_OSSL_ERROR_SUCCESS if successful or a valid ix_error token for failure.

A.2.0.46 IX_EXPORT_FUNCTION ix_error ix_ossl_thread_create (ix_ossl_thread_entry_point_t *entryPoint*, void * *arg*, ix_ossl_thread_t * *ptrTid*)

create a thread

This function creates a cancellable thread that will execute the user-provided entry point function. Custom arguments can be passed to this function using the "arg" argument.

Parameters

- *entryPoint* ix_ossl_thread_entry_point_t (in) - thread's entry point function.
- *arg* void* (in) - pointer to custom arguments that will be passed to entry point function as the first argument.
- *ptrTid* ix_ossl_thread_t* (out) - address at which the thread id of the newly created thread will be returned

Returns

IX_OSSL_ERROR_SUCCESS if successful or a valid ix_error token for failure.

A.2.0.47 IX_EXPORT_FUNCTION ix_error ix_ossl_thread_delay (int *ticks*)

delay the current task for specified number of OS ticks

This function causes the current task to delay for the specified number of OS ticks. Control of the CPU is relinquished during this time allowing other system tasks a chance to execute.

Parameters

ticks int (in) - number of OS ticks to delay task.

Returns

IX_OSSL_ERROR_SUCCESS if successful or IX_OSSL_ERROR_FAILURE for failure.

A.2.0.48 IX_EXPORT_FUNCTION void ix_ossl_thread_exit (ix_error *retError*, void * *retObj*)

Causes the calling thread to exit.

This function causes the calling thread to exit. It gives the opportunity to the caller to pass back to a waiting parent a pointer to a custom data structure and an ix_error token.

Parameters

- *retError* ix_error (in) - ix_error token to be returned to the waiting parent (0 if no error is returned)
- *retObj* void* (in) - pointer to custom data structure returned to the waiting parent on thread exit.It is used for post-mortem debugging.(null if no data structure is returned)

Returns

none

A.2.0.49 IX_EXPORT_FUNCTION ix_error ix_ossl_thread_get_id (ix_ossl_thread_t * *ptrTid*)

get id of calling thread

This function returns id of the calling thread.

Parameters

ptrTid ix_ossl_thread_t* (out) - address at which the id of the calling thread will be returned

Returns

IX_OSSL_ERROR_SUCCESS if successful or a valid ix_error token for failure.

A.2.0.50 IX_EXPORT_FUNCTION ix_error ix_ossl_thread_kill (ix_ossl_thread_t *tid*)

kills the specified thread

Kills the running thread specified by its thread id. Because the thread will be killed instantly, the caller must be extremely careful when using this function as the thread will not have time to release any of the resources it is currently owning. ix_ossl_thread_exit should be used to delete a thread and its resources instead!.

Parameters

tid ix_ossl_thread_t (in) - id of the thread to be killed

Returns

IX_OSSL_ERROR_SUCCESS if successful or a valid ix_error token for failure.

A.2.0.51 IX_EXPORT_FUNCTION void* ix_ossl_thread_main_wrapper (void * *ptrThreadInfo*)

wrapper for user-provided thread function

This function provides the needed pthread-compliant entry point function. It basically acts as a wrapper around the user-provided function. It still does one important thing which is to set the cancellation type of the thread to ASYNCHRONOUS (which translates into instantaneous")

Parameters

ptrThreadInfo void* (in) - pointer to our temporary structure containing pointers to the thread's entry point function and its argument structure

Returns

void *

A.2.0.52 IX_EXPORT_FUNCTION ix_error ix_ossl_thread_set_priority (ix_ossl_thread_t *tid*, ix_ossl_thread_priority *priority*)

sets the priority of the specified thread

This function sets the priority of the indicated thread. Possible values for 'priority' are IX_OSSL_THREAD_PRI_HIGH, IX_OSSL_THREAD_PRI_MED, and IX_OSSL_THREAD_PRI_LOW.

Parameters

- *tid* ix_ossl_thread_t (in) - id of the thread
- *priority* ix_ossl_thread_priority (in) - valid priority values are: IX_OSSL_THREAD_PRI_HIGH, IX_OSSL_THREAD_PRI_MED, and IX_OSSL_THREAD_PRI_LOW.

Returns

IX_OSSL_ERROR_SUCCESS if successful or a valid ix_error token for failure.

A.2.0.53 IX_EXPORT_FUNCTION ix_error ix_ossl_tick_get (int * *pticks*)

gets number of OS ticks per second

This function returns the number of os ticks per second.

Parameters

pticks int* (out) - pointer to location where data will be returned.

Returns

IX_OSSL_ERROR_SUCCESS.

A.2.0.54 IX_EXPORT_FUNCTION ix_error ix_ossl_time_get (ix_ossl_time_t * *ptime*)

gets current system time with nano-second resolution

This function places the current value of a timer, in seconds and nano-seconds, into an ix_ossl_time_t structure pointed by 'ptime'. This function does not provide a time-of-day. The intention is to provide a nano-second resolution time.

Parameters

ptime ix_ossl_time_t* (out) - pointer to 'ix_ossl_time_t' structure where data will be returned.

Returns

IX_OSSL_ERROR_SUCCESS if successful or a valid ix_error token for failure.

A.2.0.55 IX_EXPORT_FUNCTION int os_sleep (ix_uint32 *sleeptime_ms*, os_error * *osError*)

causes the calling thread to sleep for specified time (milliseconds)

This function causes the calling thread to sleep for the specified time.

Parameters

- *sleeptime_ms* ix_uint32 (in) - sleep time specified in milliseconds.
- *osError* os_error* (out) - pointer to the datastructure where the error conditions are returned.

A.2.0.56 IX_EXPORT_FUNCTION int os_sleep_tick (ix_uint32 *sleeptime_ticks*, os_error * *osError*)

causes the calling thread to sleep for specified time (os ticks)

This function causes the calling thread to sleep for the time specified in OS ticks.

Parameters

- *sleeptime_ticks* ix_uint32 (in) - sleep time specified in OS ticks.
- *osError* os_error* (out) - pointer to the datastructure where the error conditions are returned.

A.2.0.57 IX_EXPORT_FUNCTION int os_thread_create (void *(* *start_routine*)(void *), ix_ossl_thread_main_info_t * *ptrThreadInfo*, ix_ossl_thread_t * *ptrTid*, os_error * *osError*)

creates a thread (use ix_ossl_thread_create instead)

Create a cancellable thread that will execute the user-provided entry point function. Custom arguments can be passed to this function using the "arg" argument.

Parameters

- *start_routine* void * (*)(void *) (in) - pointer to thread's entry point function
- *ptrThreadInfo* ix_ossl_thread_main_info_t* (in) - pointer to custom argument structure that will be passed to entry point function
- *ptrTid* ix_ossl_thread_t* (out) - address at which the thread id of the newly created thread will be returned
- *osError* os_error* (out) - pointer to the datastructure where OS error codes are returned.

A.2.0.58 IX_EXPORT_FUNCTION int os_thread_exit (void * *ptrRetObj*)

exits the calling thread (use ix_ossl_thread_exit instead)

This function causes the calling thread to exit. It gives the opportunity (this is not a requirement!) to the caller to pass back to a waiting parent a pointer to a custom data structure and an ix_error token.

Parameters

ptrRetObj void* (out) - pointer to custom data structure (null if no data structure is returned)

A.2.0.59 IX_EXPORT_FUNCTION int os_thread_get_id (ix_ossl_thread_t * *ptrTid*)

get thread id (use ix_ossl_thread_get_id instead)

Returns the thread id of the calling thread

Parameters

ptrTid ix_ossl_thread_t* (out) - address at which the thread id of the inquiring thread will be returned

A.2.0.60 IX_EXPORT_FUNCTION int os_thread_kill (ix_ossl_thread_t *tid*, os_error * *osError*)

kill the specified thread (use ix_ossl_thread_kill instead)

Kills the running thread specified by its thread id.

Parameters

- *tid* ix_ossl_thread_t (in) - id of the thread to be killed

- *osError* os_error* (out) - pointer to the datastructure where OS error conditions are returned.

A.2.0.61 IX_EXPORT_FUNCTION int os_thread_mutex_create (ix_ossl_mutex_state *start_state*, ix_ossl_mutex_t * *mid*, os_error * *osError*)

creates a thread mutex object

Create a thread mutex object.

See ixOsServMutexInit for an alternative.

Parameters

- *start_state* ix_ossl_mutex_state (in) - the value that the mutex state should be initialized to.
- *mid* ix_ossl_mutex_t* (out) - id of the thread mutex created.
- *osError* os_error* (out) - pointer to the datastructure where the error conditions are returned.

A.2.0.62 IX_EXPORT_FUNCTION int os_thread_mutex_destroy (ix_ossl_mutex_t * *mutex*, os_error * *osError*)

destroy a mutex object

Destroy the thread mutex object.

see ixOsServMutexDestroy for an alternative

Parameters

- *mutex* ix_ossl_mutex_t* (in) - pointer to the thread mutex object.
- *osError* os_error* (out) - pointer to the datastructure where the error conditions are returned.

A.2.0.63 IX_EXPORT_FUNCTION int os_thread_mutex_lock (ix_ossl_mutex_t * *mutex*, ix_uint32 *timeout*, os_error * *osError*)

lock a mutex

This function locks the mutex. If the mutex is already locked, the task will block. If the time indicated in 'timeout' is reached, the task will unblock and retun error indication. If timeout is set to '0', the task will never block.

See ixOsServMutexLock for an alternative.

Parameters

- *mutex* ix_ossl_mutex_t* (in) - pointer to the thread mutex object.
- *timeout* ix_uint32 (in) - 'timeout' value
- *osError* os_error* (out) - pointer to the datastructure where the error conditions are returned.

A.2.0.64 IX_EXPORT_FUNCTION int os_thread_mutex_unlock (ix_ossl_mutex_t * *mutex*, os_error * *osError*)

unlock a mutex

This function unlocks the mutex. If there are tasks pending on the mutex, the next one is given the lock. If there are no pending tasks, then the mutex is unlocked.

See ixOsServMutexUnlock for an alternative.

Parameters

- *mutex* ix_ossl_mutex_t* (in) - pointer to the thread mutex object.
- *osError* os_error* (out) - pointer to the datastructure where the error conditions are returned.

A.2.0.65 IX_EXPORT_FUNCTION int os_thread_sema_create (int *value*, ix_ossl_sem_t * *sid*, os_error * *osError*)

create a thread semaphore

Create a thread semaphore object.

Parameters

- *value* int (in) - value that the semaphore should be initialize to.
- *sid* ix_ossl_sem_t* (out) - id of the thread semaphore created.
- *osError* os_error* (out) - pointer to the datastructure where the error conditions are returned.

A.2.0.66 IX_EXPORT_FUNCTION int os_thread_sema_destroy (ix_ossl_sem_t * *sid*, os_error * *osError*)

destroy semaphore object

Destroy the thread semaphore object.

Parameters

- *sid* ix_ossl_sem_t* (in) - pointer to the thread semaphore object.
- *osError* os_error* (out) - pointer to the datastructure where the error conditions are returned.

A.2.0.67 IX_EXPORT_FUNCTION int os_thread_sema_P (ix_ossl_sem_t * *sid*, ix_uint32 *timeout*, os_error * *osError*)

pend on a semaphore (with timeout)

Waits on the specified semaphore until the units are available or timeout occurs.

Parameters

- *sid* ix_ossl_sem_t* (in) - pointer to the semaphore object.
- *timeout* ix_uint32 (in) - timeout value.
- *osError* os_error* (out) - pointer to the datastructure where the error conditions are returned.

A.2.0.68 IX_EXPORT_FUNCTION int os_thread_sema_V (ix_ossl_sem_t * *sid*, os_error * *osError*)

release a semaphore

This function releases the specified semaphore and the semaphore state becomes available after this function call.

Parameters

- *sid* ix_ossl_sem_t* (in) - pointer to the thread semaphore object.
- *osError* os_error* (out) - pointer to the datastructure where the error conditions are returned.

A.2.0.69 IX_EXPORT_FUNCTION ix_error os_thread_set_priority (ix_ossl_thread_t * *tid*, ix_ossl_thread_priority *priority*, os_error * *osError*)

sets priority of a thread (use ix_ossl_thread_set_priority instead)

This function sets the priority of the indicated thread. possible values for 'priority' are IX_OSSL_PRI_HIGH, IX_OSSL_PRI_MED, and IX_OSSL_PRI_LOW. The effect of priority is OS dependant.

Parameters

- *tid* ix_ossl_thread_t* (in) - pointer to the thread object
- *priority* ix_ossl_thread_priority (in) - priority priority level.
- *osError* os_error* (out) - pointer to the datastructure where OS error conditions are returned.

A.2.0.70 IX_EXPORT_FUNCTION int os_time_get (ix_ossl_time_t * *ptime*, os_error * *osError*)

returns the system time with nano-second resolution

This function places the current value of time, in (seconds, nanoseconds), into the '*ptime' structure. This function does not provide a time-of-day. The purpose is to provide a nano-second resolution time.

Parameters

- *ptime* ix_ossl_time_t* (out) - address to the time structure.
- *osError* os_error* (out) - pointer to the datastructure where the error conditions are returned.

A.3 IXP425 ADSL Driver API

IXP425 ADSL Driver API. IXP425 ADSL Driver APIIXP425 ADSL Driver APIIXP425 ADSL Driver APIThe public API for the IXP425 ADSL Driver.

A.3.0.1 Typedefs

- typedef void(* IxAdslStateChangeCallback)(UINT32 lineNum, IxAdslLineState lineState)
  Callback function to indicate of the changes on the line state.

A.3.0.2 Enumerations

- enum IxAdslStatus { IX_ADSL_STATUS_SUCCESS = IX_SUCCESS, IX_ADSL_STATUS_FAIL = IX_FAIL, IX_ADSL_STATUS_UNSUPPORTED_MODE, IX_ADSL_STATUS_ALREADY_DOWN }
  These status will be used by the APIs to return to the client.
- enum IxAdslLineState { IX_ADSL_LINE_STATE_UP_DUAL_LATENCY = 0, IX_ADSL_LINE_STATE_WAIT_FOR_ACTIVATING, IX_ADSL_LINE_STATE_ACTIVATING, IX_ADSL_LINE_STATE_DOWN, IX_ADSL_LINE_STATE_UP_FASTCHANNEL, IX_ADSL_LINE_STATE_UP_INTERLEAVECHANNEL, IX_ADSL_LINE_STATE_INVALID }
  These status will be used to indicate the line state.
- enum IxAdslLineType { IX_ADSL_AUTOSELECT = 0, IX_ADSL_GLITE = 1, IX_ADSL_DMT = 2, IX_ADSL_ANSI = 3, IX_ADSL_LOOPBACK = 4, IX_ADSL_INVALID_MODE = 5 }
  Used to indicate the type of ADSL line type used.
- enum IxAdslPhyType { IX_ADSL_PHY_CPE = 0, IX_ADSL_PHY_INVALID }
  Used to indicate the ADSL physical type - CPE.

A.3.0.3 Functions

- PUBLIC IxAdslStatus ixAdslLineOpen (UINT32 lineNum, IxAdslLineType lineType, IxAdslPhyType phyType)
  Open the given ADSL line in the specified mode and type.
- PUBLIC IxAdslStatus ixAdslLineClose (UINT32 lineNum)
  Closes a previously opened ADSL line.
- PUBLIC IxAdslStatus ixAdslLineStateChangeCallbackRegister (UINT32 lineNum, IxAdslStateChangeCallback lineChangeCallbackFn)
  This is a notification registration procedure that gets called if the line state of the given ADSL line changes. The maximum callbacks that can be registered is defined as IX_ADSL_SIZEOF_CALLBACK_LIST (The default value is 10).
- PUBLIC IxAdslLineState ixAdslLineStateGet (UINT32 lineNum)
  Returns the current state of the given ADSL line.
- PUBLIC UINT32 ixAdslLineRateUpstreamGet (UINT32 lineNum)
  Return the current upstream line speed of the given ADSL line.
- PUBLIC UINT32 ixAdslLineRateDownstreamGet (UINT32 lineNum)
  Return the current downstream line speed of the given ADSL line.
- PUBLIC IxAdslStatus ixAdslDyingGaspEnable (UINT32 lineNum)

Enables the function that informs ATU-C when condition that leads to shutdown of the given Adsl line has been detected.

- PUBLIC IxAdslStatus ixAdslVendorCodeSet (UINT32 lineNum, UINT8 ixAdslItuVendoridCountrycode, UINT8 ixAdslItuVendoridVendorcode1, UINT8 ixAdslItuVendoridVendorcode2, UINT8 ixAdslItuVendoridVendorcode3, UINT8 ixAdslItuVendoridVendorcode4, UINT8 ixAdslItuVendoridVendorspecific1, UINT8 ixAdslItuVendoridVendorspecific2)

  Set the vendor specific bytes in the given ADSL line.
- PUBLIC void ixAdslShow (UINT32 lineNum)

  This function will show the current statistics associated with the given ADSL Line.
- PUBLIC void ixAdslMemoryUnmap (void)

  This function will unmap the dynamically allocated addresses.

A.3.0.4 Detailed Description

The public API for the IXP425 ADSL Driver.

A.3.0.5 Typedef Documentation

A.3.0.6 typedef void(* IxAdslStateChangeCallback)(UINT32 lineNum, IxAdslLineState lineState)

Callback function to indicate of the changes on the line state.

Definition at line 171 of file IxAdsl.h.

A.3.0.7 Enumeration Type Documentation

A.3.0.8 enum IxAdslLineState

These status will be used to indicate the line state.

Enumeration Values

- ***IX_ADSL_LINE_STATE_UP_DUAL_LATENCY*** The line is in showtime state.
  Fast & Interleaved Channel
- ***IX_ADSL_LINE_STATE_WAIT_FOR_ACTIVATING*** The line is waiting for the peer to activate.
- ***IX_ADSL_LINE_STATE_ACTIVATING*** The line is negotiating with its peer.
- ***IX_ADSL_LINE_STATE_DOWN*** The line is down.
- ***IX_ADSL_LINE_STATE_UP_FASTCHANNEL*** The line is in showtime state.
  Fast Channel
- ***IX_ADSL_LINE_STATE_UP_INTERLEAVECHANNEL*** The line is in showtime state.
  Interleaved Channel
- ***IX_ADSL_LINE_STATE_INVALID*** ADSL line in an unknown state.
  Definition at line 84 of file IxAdsl.h.

A.3.0.9 enum IxAdslLineType

Used to indicate the type of ADSL line type used.

Enumeration Values

- ***IX_ADSL_AUTOSELECT*** This is an auto-select mode for the CPE to auto-configure based on the CO/DSLAM line type; DMT, ANSI or G.lite.
- ***IX_ADSL_GLITE*** G.lite line type.
- ***IX_ADSL_DMT*** DMT line type.
- ***IX_ADSL_ANSI*** ANSI line type.
- ***IX_ADSL_LOOPBACK*** Utopia Loopback line type.
- ***IX_ADSL_INVALID_MODE*** Used internally to indicate last valid enum.

  Definition at line 125 of file IxAdsl.h.

A.3.0.10 enum IxAdslPhyType

Used to indicate the ADSL physical type - CPE.

Note: IxAdslPhyType is declared as an enum due to forward compatibility to support CO (fast and interleaved mode) in the future.

Enumeration Values

- ***IX_ADSL_PHY_CPE*** Adsl type is CPE.
- ***IX_ADSL_PHY_INVALID*** Adsl type is invalid.

  Definition at line 154 of file IxAdsl.h.

A.3.0.11 enum IxAdslStatus

These status will be used by the APIs to return to the client.

Enumeration Values

- ***IX_ADSL_STATUS_SUCCESS*** Successful API execution.
- ***IX_ADSL_STATUS_FAIL*** Failed API execution.
- ***IX_ADSL_STATUS_UNSUPPORTED_MODE*** Unsupported mode type for IxAdslLineOpen function.
- ***IX_ADSL_STATUS_ALREADY_DOWN*** Line is already down.

  Definition at line 63 of file IxAdsl.h.

A.3.0.12 Function Documentation

A.3.0.13 ixAdslDyingGaspEnable (UINT32 *lineNum*)

Enables the function that informs ATU-C when condition that leads to shutdown of the given Adsl line has been detected.

Note: The parameter lineNum exists for future Multi-PHY support. Only lineNum = 0 is valid.

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.

Parameters

lineNum [in] is the parameter showing which ADSL line is being used.

Returns

- IX_ADSL_STATUS_SUCCESS - Dying Gasp is enabled. successfully.
- IX_ADSL_STATUS_FAILED - Failed to enable Dying Gasp.

A.3.0.14 ixAdslLineClose (UINT32 *lineNum*)

Closes a previously opened ADSL line.

The line will closed and put in the idle state.

Note: The parameter lineNum exists for future Multi-PHY support. Only lineNum = 0 is valid.

Blocking : Non-blocking

Impacts On Global Data: Sets the lineEnable State . Notifies a callback routine.

Pre-Conditions: Code should only be called from task level.

Post-Conditions: No cleanup after this call is required.

Parameters

lineNum [in] is the parameter showing which ADSL line is being used and to be closed.

Returns

- IX_ADSL_STATUS_SUCCESS - Line was closed successfully.
- IX_ADSL_STATUS_FAILED - Line failed to close properly.
- IX_ADSL_STATUS_ALREADY_DOWN - Line was not open before close.

A.3.0.15 ixAdslLineOpen (UINT32 *lineNum*, IxAdslLineType *lineType*, IxAdslPhyType *phyType*)

Open the given ADSL line in the specified mode and type.

Opens the given ADSL line in the specified mode and puts it in the 'Showtime' state, i.e. available to carry user data.

Notes:

- The parameter lineNum exists for future Multi-PHY support. Only lineNum = 0 is valid.
- The parameter phyType exists for future CO support.

  Blocking : This call may block for several seconds while the link is established.

  Impacts On Global Data: Sets the lineEnable 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 such as Atmd, Atmm, Atm scheduler and Utopia. The code should only be called from task level.

  Post-Conditions: No cleanup after this call is required.

  Exceptions: None.

Parameters

- *lineNum* [in] is the parameter showing which ADSL line is being used.
- *lineType* [in] indicates type of ADSL to be opened.
- *phyType* [in] is the type of Phy used - CPE.

Returns

- IX_ADSL_STATUS_SUCCESS - Line was opened and is in 'showtime' state
- IX_ADSL_STATUS_FAILED - Line failed to open properly.
- IX_ADSL_STATUS_UNSUPPORTED_MODE - Illegal ADSL type for IxAdslLineOpen function.

A.3.0.16 ixAdslLineRateDownstreamGet (UINT32 *lineNum*)

Return the current downstream line speed of the given ADSL line.

The data returned by this API is the received (Rx) rate of the line the ATU device.

Note: The parameter lineNum exists for future Multi-PHY support. Only lineNum = 0 is valid.

Blocking : This call is a non-blocking

Impacts On Global Data: None

Pre-Conditions: Task level calls only.

Post-Conditions: None.

Exceptions: None.

Parameters

lineNum [in] is the parameter showing which ADSL line is being used.

Returns

- Integer - bit rate in kbits/second. N.B. Returns zero if the line is not in 'Showtime' state.

A.3.0.17 ixAdslLineRateUpstreamGet (UINT32 *lineNum*)

Return the current upstream line speed of the given ADSL line.

Note: The parameter lineNum exists for future Multi-PHY support. Only lineNum = 0 is valid.

The data returned by this API represents the transmit (Tx) rate of the line from the ATU device.

Blocking : This call is a non-blocking

Impacts On Global Data: None

Pre-Conditions: Task level calls only.

Post-Conditions: None.

Exceptions: None.

Parameters

lineNum [in] is the parameter showing which ADSL line is being used.

Returns

- Integer - bit rate in kbits/second. N.B. Returns zero if the line is not in 'Showtime' state.

A.3.0.18 ixAdslLineStateChangeCallbackRegister (UINT32 *lineNum*, IxAdslStateChangeCallback *lineChangeCallbackFn*)

This is a notification registration procedure that gets called if the line state of the given ADSL line changes. The maximum callbacks that can be registered is defined as IX_ADSL_SIZEOF_CALLBACK_LIST (The default value is 10).

Note: The parameter lineNum exists for future Multi-PHY support. Only lineNum = 0 is valid

Blocking : This call is a non-blocking function.

Impacts On Global Data: This sets a global callback handler.

Pre-Conditions: There are no pre conditions to this call.

Post-Conditions: A global line state handler for ADSL line state changes shall be registered. It is advisable to register the callback before the given ADSL line is opened.

Exceptions: None.

Parameters

- *lineNum* [in] is the parameter showing which ADSL line is being used.
- *lineChangeCallbackFn* [in] is the callback function that will be invoked when there is a state change

Returns

- IX_ADSL_STATUS_SUCCESS - The callback function was registered successfully.
- IX_ADSL_STATUS_FAILED - Internal error, registration of the callback function failed.

A.3.0.19 ixAdslLineStateGet (UINT32 *lineNum*)

Returns the current state of the given ADSL line.

Note: The parameter lineNum exists for future Multi-PHY support. Only lineNum = 0 is valid.

Blocking : This call is a non-blocking

Impacts On Global Data: None

Pre-Conditions: Task level calls only.

Post-Conditions: None.

Exceptions: None.

Parameters

lineNum [in] is the parameter showing which ADSL line is being used.

Returns

- IX_ADSL_LINE_STATE_UP_FAST - The line is in show time state.
- IX_ADSL_LINE_STATE_WAIT_FOR_ACTIVATING - The line is waiting fo the peer to activate.
- IX_ADSL_LINE_STATE_ACTIVATING - The line is negotiating with its peer.
- IX_ADSL_LINE_STATE_DOWN - The line is idle.
- IX_ADSL_LINE_STATE_INVALID - The line is in an unknown state.

A.3.0.20 ixAdslMemoryUnmap (void)

This function will unmap the dynamically allocated addresses.

Blocking : This call is non-blocking.

Impacts On Global Data: None

Pre-Conditions: Task level calls only.

Post-Conditions: None.

Exceptions: None.

Parameters

None

Returns

None

A.3.0.21 ixAdslShow (UINT32 *lineNum*)

This function will show the current statistics associated with the given ADSL Line.

Note: The parameter lineNum exists for future Multi-PHY support. Only lineNum = 0 is valid.

The list of statistics to be shown by IxAdslShow:

- Controller SW Version
- ADSL Line State
- Line Number
- Upstream and Downstream Rate
- Training Statistics for Upstream and Downstream Rates
- ADSL Near End Operational Data such as Upstream Relative Capacity Occupancy, Noise Margin Upstream, Output Pwr Downstream, Attenuation Upstream, Downstream Fast Bitrate, Downstream Interleaved Bitrate, Near-end defect bitmap, Loss of Frame (secs), Loss of Cell delineation (secs), Loss of Signal (secs), Loss of Margin (secs), Errored seconds, HEC and FEC Errors.
- ADSL Far End Operational Data such as Downstream Relative Capacity Occupancy, Noise Margin Downstream, Output Pwr Upstream, Attenuation Downstream, Upstream Fast Bitrate, Upstream Interleaved Bitrate, Far-end defect bitmap
- Tx and Rx ATM Cell Counters

Blocking : This call is non-blocking.

Impacts On Global Data: None

Pre-Conditions: Task level calls only.

Post-Conditions: None.

Exceptions: None.

Parameters

lineNum [in] is the parameter showing which ADSL line is being used.

Returns

None

A.3.0.22 ixAdslVendorCodeSet (UINT32 *lineNum*, UINT8 *ixAdslItuVendoridCountrycode*, UINT8 *ixAdslItuVendoridVendorcode1*, UINT8 *ixAdslItuVendoridVendorcode2*, UINT8 *ixAdslItuVendoridVendorcode3*, UINT8 *ixAdslItuVendoridVendorcode4*, UINT8 *ixAdslItuVendoridVendorspecific1*, UINT8 *ixAdslItuVendoridVendorspecific2*)

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.

Notes:

- Vendor specific values are taken by the phy and linked together to form a single code
- Note that the parameter lineNum exists for future Multi-PHY support. Only lineNum = 0 is valid.

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.

Parameters

- *lineNum* [in] is the parameter showing which ADSL line is being used.
- *ixAdslItuVendoridCountrycode* [in] is the vendor country code that are predefined in standards.
- *ixAdslItuVendoridVendorcode1* [in] is the vendor code 1 that are predefined in standards.
- *ixAdslItuVendoridVendorcode2* [in] is the vendor code 2 that are predefined in standards.
- *ixAdslItuVendoridVendorcode3* [in] is the vendor code 3 that are predefined in standards.
- *ixAdslItuVendoridVendorcode4* [in] is the vendor code 4 that are predefined in standards.
- *ixAdslItuVendoridVendorspecific1* [in] is the vendor specific 1 that are predefined in standards.
- *ixAdslItuVendoridVendorspecific2* [in] is the vendor specific 2 that are predefined in standards.

Returns

- IX_ADSL_STATUS_SUCCESS - Set Vendor Code successful.
- IX_ADSL_STATUS_FAILED - Set Vendor Code failed because the line is up.

A.4 IXP425 Assertion Macros (IxAssert) API

IXP425 Assertion Macros (IxAssert) APIIXP425 Assertion Macros (IxAssert) APIIXP425 Assertion Macros (IxAssert) APIIXP425 Assertion Macros (IxAssert) APIAssertion support.

A.4.0.1 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.
- #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.

A.4.0.2 Detailed Description

Assertion support.

A.4.0.3 Define Documentation

A.4.0.4 #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.

Definition at line 69 of file IxAssert.h.

A.4.0.5 #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.

Definition at line 77 of file IxAssert.h.

A.5 IXP425 ATM Driver Access (IxAtmdAcc) API

IXP425 ATM Driver Access (IxAtmdAcc) API. IXP425 ATM Driver Access (IxAtmdAcc) APIIXP425 ATM Driver Access (IxAtmdAcc) APIIXP425 ATM Driver Access (IxAtmdAcc) APIThe public API for the IXP425 Atm Driver Data component.

A.5.0.1 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 dsicriminate the OAM channel, and should be chosen so that it does not coencide 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 dsicriminate the OAM channel, and should be chosen so that it does not coencide 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 value used for the dedicated OAM Rx connection. This value should be chosen so that it does not coencide 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 value used for the dedicated OAM Rx connection. This value should be chosen so that it does not coencide 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.5.0.2 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.5.0.3 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.5.0.4 Functions

- PUBLIC IX_STATUS ixAtmdAccInit (void)

  Initialise 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 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.5.0.5 Detailed Description

The public API for the IXP425 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 independantly 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 descriminate the dedicated OAM channels and do not identify a particular OAM F4/F5 flow. F4/F5 flows may be realised for particluar 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.5.0.6 Define Documentation

A.5.0.7 #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

Definition at line 136 of file IxAtmdAcc.h.

A.5.0.8 #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

Definition at line 167 of file IxAtmdAcc.h.

A.5.0.9 #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.

Definition at line 208 of file IxAtmdAcc.h.

A.5.0.10 #define IX_ATMDACC_OAM_RX_VCI 0

The reserved value value used for the dedicated OAM Rx connection. This value should be chosen so that it does not coencide 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.

Definition at line 232 of file IxAtmdAcc.h.

A.5.0.11 #define IX_ATMDACC_OAM_RX_VPI 0

The reserved value value used for the dedicated OAM Rx connection. This value should be chosen so that it does not coencide 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.

Definition at line 220 of file IxAtmdAcc.h.

A.5.0.12 #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 dsicriminate the OAM channel, and should be chosen so that it does not coencide 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.

Definition at line 196 of file IxAtmdAcc.h.

A.5.0.13 #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 dsicriminate the OAM channel, and should be chosen so that it does not coencide 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.

Definition at line 183 of file IxAtmdAcc.h.

A.5.0.14 #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.

Definition at line 152 of file IxAtmdAcc.h.

A.5.0.15 #define IX_ATMDACC_WARNING 2

Warning return code.

This constant is used to tell IxAtmDAcc user about a special case.

Definition at line 121 of file IxAtmdAcc.h.

A.5.0.16 Typedef Documentation

A.5.0.17 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

Definition at line 401 of file IxAtmdAcc.h.

A.5.0.18 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 partilcular connection. The callback is likely to push the mbufs to a protocol layer, and recycle the mbufs for a further use.

Note: 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 possibile value is AAL0_VALID, in this case the client may optionally determine that an rx timeout occured by checking if the mbuf is compleletly 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 servive 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.

Definition at line 363 of file IxAtmdAcc.h.

A.5.0.19 typedef void(* IxAtmdAccTxVcBufferReturnCallback)(IxAtmdAccUserId userId, IX_MBUF * mbufPtr)

Buffer callback prototype.

This function is called to relinguish 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. Definition at line 426 of file IxAtmdAcc.h.

A.5.0.20 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

Definition at line 315 of file IxAtmdAcc.h.

A.5.0.21 Enumeration Type Documentation

A.5.0.22 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

Definition at line 270 of file IxAtmdAcc.h.

A.5.0.23 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.

Definition at line 291 of file IxAtmdAcc.h.

A.5.0.24 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

Definition at line 247 of file IxAtmdAcc.h.

A.5.0.25 Function Documentation

A.5.0.26 ixAtmdAccInit (void)

Initialise the IxAtmdAcc Component.

This function initialise the IxAtmdAcc component. This function shall be called before any other function of the API. Its role is to initialise all internal resources of the IxAtmdAcc component.

The ixQmgr component needs to be initialized prior the use of *ixAtmdAccInit()*

Parameters

none

Failing to initilialize the IxAtmdAcc API before any use of it will result in a failed status. If the specified component is not present, a success status will still be returned, however, a warning indicating the NPE to download to is not present will be issued.

Returns

- IX_SUCCESS initialisation is complete (in case of component not being present, a warning is clearly indicated)

- IX_FAIL unable to process this request either because this IxAtmdAcc is already initialised or some unspecified error has occrred.

A.5.0.27 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 succesful call to a connect function. This identifier is invalid after a sucessful 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 prioriy 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 descriminate the dedicated OAM channels and do not identify a particular OAM F4/F5 flow. F4/F5 flows may be realised for particluar 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 identifieds which of two Qs the VC should use.when icoming 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
- 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 initialised, or some other error occurs during processing.

A.5.0.28 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 succesfull call to *IxAtmdAccRxVcConnect()*

Returns

- IX_SUCCESS successful call to *ixAtmdAccRxVcDisable()* .
- IX_ATMDACC_WARNING the channel is already disabled
- IX_FAIL invalid parameters or some unspecified internal error occured

A.5.0.29 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 succesfull call to *IxAtmdAccRxVcConnect()*

Returns

- IX_SUCCESS successful call to ixAtmdAccRxVcEnable
- IX_ATMDACC_WARNING the channel is already enabled
- IX_FAIL invalid parameters or some unspecified internal error occured.

A.5.0.30 ixAtmdAccRxVcFreeEntriesQuery (IxAtmConnId *connId*, unsigned int * *numberOfMbufsPtr*)

Get the number of rx mbufs the system can accept to replenish the 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 succesfull 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.5.0.31 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 curent 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 succesfull 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 occrred.

Note: the callback will be called when the threshold level will drop from exactly (numberOfMbufs + 1) to (numberOfMbufs).

A.5.0.32 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 efficients.

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 succesfull 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.

Note: 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.5.0.33 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 succesfull 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.5.0.34 ixAtmdAccShow (void)

Show IxAtmdAcc configuration on a per port basis.

Parameters

none

Returns

none

Note: Display use printf() and are redirected to stdout.

A.5.0.35 ixAtmdAccStatsReset (void)

Reset all IxAtmdAcc stats.

Parameters

none

Returns

None

A.5.0.36 ixAtmdAccStatsShow (void)

Show all IxAtmdAcc stats.

Parameters

none

Returns

none

Note: Stats display use printf() and are redirected to stdout

A.5.0.37 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 succesful call to a connect function. This identifier is invalid after a sucessful 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 descriminate the dedicated OAM channels and do not identify a particular OAM F4/F5 flow. F4/F5 flows may be realised for particluar 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 initialised, or some other error occurs during processing.

Note: Unscheduled mode is not supported in ixp425 1.0. Therefore, the function *ixAtmdAccPortTxScheduledModeEnable()* need to be called for this port before any establishing a Tx Connection

A.5.0.38 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 sufficent 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 independantly of the AAL0/AAL5 traffic for that connection.

When transmission is complete, the TX Done mechanism will give the owmnership 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 succesfull 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 (e.g. : 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.5.0.39 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 succesfull 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 TxDone 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. Diabling the port and retry a disconnect will free the resources associated with this channel.

See also:

ixAtmdAccPortTxProcess

A.6 IXP425 ATM Driver Access (IxAtmdAcc) Control API

IXP425 ATM Driver Access (IxAtmdAcc) Control API. IXP425 ATM Driver Access (IxAtmdAcc) Control APIIXP425 ATM Driver Access (IxAtmdAcc) Control APIIXP425 ATM Driver Access (IxAtmdAcc) Control APIThe public API for the IXP425 Atm Driver Control component.

A.6.0.1 Modules

- IXP425 ATM Driver Access (IxAtmdAcc) Utopia Control API
  The public API for the IXP425 Atm Driver Control component.

A.6.0.2 Defines

- #define IX_ATMDACC_PORT_DISABLE_IN_PROGRESS 5
  Port enable return code.
- #define IX_ATMDACC_ALLPDUS 0xffffffff
  All PDUs.

A.6.0.3 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.6.0.4 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.6.0.5 Detailed Description

The public API for the IXP425 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.6.0.6 Define Documentation

A.6.0.7 #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

Definition at line 103 of file IxAtmdAccCtrl.h.

A.6.0.8 #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.

Definition at line 86 of file IxAtmdAccCtrl.h.

A.6.0.9 Typedef Documentation

A.6.0.10 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.

Definition at line 217 of file IxAtmdAccCtrl.h.

A.6.0.11 typedef IX_STATUS(* IxAtmdAccRxDispatcher)(IxAtmRxQueueId rxQueueId, unsigned int numberOfPdusToProcess, unsigned int *reservedPtr)

Callback prototype for notification of available PDUs for an Rx Q.

This a protoype 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 aprropriate 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.

  Definition at line 137 of file IxAtmdAccCtrl.h.

A.6.0.12 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 sytem-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 occured. This is provided to make this prototype compatible with *ixAtmdAccTxDoneDispatch()*

  Definition at line 188 of file IxAtmdAccCtrl.h.

A.6.0.13 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

  Definition at line 317 of file IxAtmdAccCtrl.h.

A.6.0.14 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

Definition at line 282 of file IxAtmdAccCtrl.h.

A.6.0.15 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 the function is registering the cell demand for this VC.
- IX_FAIL the function cannot register cell for this VC : the scheduler maybe overloaded or misconfigured

Definition at line 248 of file IxAtmdAccCtrl.h.

A.6.0.16 Function Documentation

A.6.0.17 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 synchrnisation with tx resources. Therefore, the txDone mechanism needs to be serviced to make a PortDisable complete.

See also:

ixAtmdAccPortEnable
ixAtmdAccPortDisableComplete
ixAtmdAccTxDoneDispatch

A.6.0.18 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.6.0.19 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 initialise 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.6.0.20 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 initialisation, outside an interrupt context

A.6.0.21 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 occured.

A.6.0.22 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 ouside 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. If the

Obsolete or invalid connection Id will be silently discarded.

This function is not reentrant for the same port.

This functions doesn't 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 occured. 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.6.0.23 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 faciltates 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 regulary 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 sheduling 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 ixp425 1.0

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.6.0.24 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 particluar 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 maxiumum 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 fuction is invoked. Because of real-time constraints, there is no guarantee thatthe 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 occured.

A.6.0.25 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 threhold 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.6.0.26 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.6.0.27 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.6.0.28 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 oustanding 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 sufficently 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) maxiumum 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 occured.

A.6.0.29 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 sytem-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 initialisation 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 ust not be a null pointer.

Returns

- IX_SUCCESS Successful call to ixAtmdAccTxDoneDispatcherRegister
- IX_FAIL error in the Parameters

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 preffered 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.6.0.30 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
- IX_FAIL wrong parameter (null pointer as parameter).or unspecified rocessing error occurs..The value in numberOfPdusPtr is unspecified.

A.6.0.31 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 the number of pdus the system is able to queue after transmission
- IX_FAIL wrong parameter (null pointer as parameter).or unspecified rocessing 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.6.0.32 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
- initialise the NPE
- reset the Utopia cell counters and status registers to known values

  This action has to be done once at initialisation. A lock is preventing the concurrent use of *ixAtmdAccUtopiaStatusGet()* and 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.6.0.33 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 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.6.1 IXP425 ATM Driver Access (IxAtmdAcc) UTOPIA Control API

IXP425 ATM Driver Access (IxAtmdAcc) Utopia Control API. IXP425 ATM Driver Access (IxAtmdAcc) Utopia Control APIIXP425 ATM Driver Access (IxAtmdAcc) Utopia Control APIIXP425 ATM Driver Access (IxAtmdAcc) Utopia Control APIThe public API for the IXP425 Atm Driver Control component.

A.6.1.1 Data Structures

- struct IxAtmdAccUtopiaConfig
  Utopia configuration.
- struct IxAtmdAccUtopiaConfig::UtRxConfig
  Utopia Rx config Register.
- struct IxAtmdAccUtopiaConfig::UtRxDefineIdle
  Utopia Rx idle cells config Register.
- struct IxAtmdAccUtopiaConfig::UtRxEnableFields
  Utopia Rx enable Register.
- struct IxAtmdAccUtopiaConfig::UtRxStatsConfig
  Utopia Rx stats config 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 IxAtmdAccUtopiaConfig::UtTxConfig
  Utopia Tx Config Register.
- struct IxAtmdAccUtopiaConfig::UtTxDefineIdle
  Utopia Tx idle cells Register.
- struct IxAtmdAccUtopiaConfig::UtTxEnableFields
  Utopia Tx ienable fields Register.
- struct IxAtmdAccUtopiaConfig::UtTxStatsConfig
  Utopia Tx stats 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 IxAtmdAccUtopiaStatus
  Utopia status.
- struct IxAtmdAccUtopiaStatus::UtRxCellConditionStatus
  Utopia Rx Status Register.
- struct IxAtmdAccUtopiaStatus::UtTxCellConditionStatus
  Utopia Tx Status Register.

A.6.1.2 Detailed Description

The public API for the IXP425 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.7 IXP425 ATM Manager (IxAtmm) API

IXP425 ATM Manager (IxAtmm) API. IXP425 ATM Manager (IxAtmm) APIIXP425 ATM Manager (IxAtmm) APIIXP425 ATM Manager (IxAtmm) APIIXP425 ATM Manager component Public API.

A.7.0.1 Data Structures

- struct IxAtmmPortCfg

  Structure contains port-specific information required to initialize IxAtmm, and specifically, the IXP425 UTOPIA Level-2 device.

- struct IxAtmmVc

  This structure describes the required attributes of a virtual connection.

A.7.0.2 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.7.0.3 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.7.0.4 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 AAAAAAAACJ interface to indicate that UTOPIA loopback should be enabled or disabled on initialisation.

- 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.7.0.5 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 initialisation, all the ports are disabled.
- IX_STATUS ixAtmmPortDisable (IxAtmLogicalPort port)
  The client call this interface to disable transmit for an ATM port. At initialisation, 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.7.0.6 Detailed Description

IXP425 ATM Manager component Public API.

A.7.0.7 Define Documentation

A.7.0.8 #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.

Definition at line 96 of file IxAtmm.h.

A.7.0.9 Typedef Documentation

A.7.0.10 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

  Definition at line 212 of file IxAtmm.h.

A.7.0.11 Enumeration Type Documentation

A.7.0.12 enum IxAtmmPhyMode

Definitions for use with ixAtmmUtopiaInit interface to indicate that UTOPIA multi-phy/single-phy mode is used.

Enumeration Values

- ***IX_ATMM_MPHY_MODE*** Atmm phy mode mphy.
- ***IX_ATMM_SPHY_MODE*** Atmm phy mode sphy.
- ***IX_ATMM_PHY_MODE_INVALID*** Atmm phy mode invalid.

  Definition at line 173 of file IxAtmm.h.

A.7.0.13 enum IxAtmmUtopiaLoopbackMode

Definitions for use with AAAAAAAACJ interface to indicate that UTOPIA loopback should be enabled or disabled on initialisation.

Enumeration Values

- ***IX_ATMM_UTOPIA_LOOPBACK_DISABLED*** Atmm Utopia loopback mode disabled.
- ***IX_ATMM_UTOPIA_LOOPBACK_ENABLED*** Atmm Utopia loopback mode enabled.
- ***IX_ATMM_UTOPIA_LOOPBACK_INVALID*** Atmm Utopia loopback mode invalid.

Definition at line 149 of file IxAtmm.h.

A.7.0.14 enum IxAtmmVcChangeEvent

Definition for use with IxAtmmVcChangeCallback callback. Indicates that the event type represented by the callback for this VC.

Enumeration Values

- ***IX_ATMM_VC_CHANGE_EVENT_REGISTER*** Atmm Vc event register.
- ***IX_ATMM_VC_CHANGE_EVENT_DEREGISTER*** Atmm Vc event de-register.
- ***IX_ATMM_VC_CHANGE_EVENT_INVALID*** Atmm Vc event invalid.

Definition at line 139 of file IxAtmm.h.

A.7.0.15 enum IxAtmmVcDirection

Definition for use in the IxAtmmVc structure. Indicates the direction of a VC.

Enumeration Values

- ***IX_ATMM_VC_DIRECTION_TX*** Atmm Vc direction transmit.
- ***IX_ATMM_VC_DIRECTION_RX*** Atmm Vc direction receive.
- ***IX_ATMM_VC_DIRECTION_INVALID*** Atmm Vc direction invalid.

Definition at line 129 of file IxAtmm.h.

A.7.0.16 Function Documentation

A.7.0.17 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.7.0.18 ixAtmmPortDisable (IxAtmLogicalPort *port*)

The client call this interface to disable transmit for an ATM port. At initialisation, 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.7.0.19 ixAtmmPortEnable (IxAtmLogicalPort *port*)

The client call this interface to enable transmit for an ATM port. At initialisation, all the ports are disabled.

Parameters

IxAtmLogicalPort port Value identifies the port

Returns

- IX_SUCCESS : Transmission over this port is started.
- 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.7.0.20 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 IXP425 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 specificed 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.7.0.21 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.7.0.22 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.

A.7.0.23 ixAtmmUtopiaCfgShow (void)

Display utopia information(config registers and status registers).

Parameters

none

Returns

- IX_SUCCESS : Show function was successful
- IX_FAIL : Internal failure

A.7.0.24 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 : The supplied parameters are invalid or have been rejected by the UTOPIA-NPE device.

Warning: This interface may only be called once. 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.7.0.25 ixAtmmUtopiaStatusShow (void)

Display utopia status counters.

Parameters

none

Returns

- IX_SUCCESS : Show function was successful
- IX_FAIL : Internal failure

A.7.0.26 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.7.0.27 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 calbacks 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 appropiate 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 connot 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.7.0.28 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.7.0.29 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.7.0.30 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.7.0.31 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 becuase 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.8 IXP425 ATM Transmit Scheduler (IxAtmSch) API

IXP425 ATM Transmit Scheduler (IxAtmSch) API. IXP425 ATM Transmit Scheduler (IxAtmSch) APIIXP425 ATM Transmit Scheduler (IxAtmSch) APIIXP425 ATM Transmit Scheduler (IxAtmSch) APIIXP425 ATM scheduler component Public API.

A.8.0.1 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.8.0.2 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.8.0.3 Detailed Description

IXP425 ATM scheduler component Public API.

A.8.0.4 Define Documentation

A.8.0.5 #define IX_ATMSCH_RET_NOT_ADMITTED 2

Indicates that CAC function has rejected VC registration due to insufficient line capacity.

Definition at line 81 of file IxAtmSch.h.

A.8.0.6 #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.

Definition at line 99 of file IxAtmSch.h.

A.8.0.7 #define IX_ATMSCH_RET_QUEUE_FULL 3

Indicates that the VC queue is full, no more demand can be queued at this time.

Definition at line 90 of file IxAtmSch.h.

A.8.0.8 Function Documentation

A.8.0.9 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
  - The ATM scheduler component has been successfully initialized.
  - The scheduler is ready to accept Port modelling requests.
- **IX_FAIL :** Some internal error has prevented the scheduler component from initialising.

A.8.0.10 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 IX_UTOPIA_MAX_PORTS - 1, representing a maximum of IX_UTOPIA_MAX_PORTS possible ports.
- *unsigned int portRate* Value indicating the upstream capacity of the indicated port. The value should be supplied in units of ATM (53 bytes) cells per second. A port rate of 800Kbits/s is the equivalent of 1886 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
  - The ATM scheduler has been successfully initialized.
  - The requested port model has been established.
  - The scheduler is ready to accept VC modelling requests on the ATM port.
- **IX_FAIL :** indicates the requested port could not be initialized.

A.8.0.11 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. A port rate of 800Kbits/s is the equivalent of 1886 cells per second

Returns

- **IX_SUCCESS :** The port rate has been successfully modified.
- **IX_FAIL :** The 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.8.0.12 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.8.0.13 ixAtmSchStatsClear (void)

Utility function which will reset all counter statistics in the ATM scheduler to zero.

Parameters

none

Returns

none

A.8.0.14 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 :** The 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 :** The input are invalid. No action is taken.

  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.
- 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.8.0.15 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 communicatations from the Scheduler to the scheduler user e.g. schedule table entries.

Returns

- **IX_SUCCESS :** The id has successfully been set.
- **IX_FAIL :** Input data are invalid. connId id is not established.

A.8.0.16 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 :** The 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.8.0.17 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 :** The VC has been successfully established on this port. The client may begin to submit demand on this VC.
- **IX_ATMSCH_RET_NOT_ADMITTED :** The 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.8.0.18 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 IXP425 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 :** The VC queue has been successfully cleared.
- **IX_FAIL :** The 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.8.0.19 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 IXP425 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 :** The VC queue has been successfully updated.
- **IX_ATMSCH_RET_QUEUE_FULL :** The 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 :** The 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.9 IXP425 ATM Types (IxAtmTypes)

IXP425 ATM Types (IxAtmTypes). IXP425 ATM Types (IxAtmTypes)IXP425 ATM Types (IxAtmTypes)IXP425 ATM Types (IxAtmTypes)The common set of types used in many Atm components.

A.9.0.1 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.9.0.2 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
  Size of a OAM cell, excluding HEC byte.
- #define IX_ATM_AAL0_48_CELL_PAYLOAD_SIZE IX_ATM_CELL_PAYLOAD_SIZE

Size of a AAL0 48 Cell payload.

- #define IX_ATM_AAL5_CELL_PAYLOAD_SIZE IX_ATM_CELL_PAYLOAD_SIZE
  Size of a AAL5 Cell payload.
- #define IX_ATM_AAL0_52_CELL_SIZE_NO_HEC IX_ATM_CELL_SIZE_NO_HEC
  Size of a AAL0 52 Cell, excluding HEC byte.
- #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
  Maximum number of active AAL5/AAL0 VCs in the system.
- #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 compatiblity. For current software release 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 accross 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_CELL_HEADER_VCI_GET(cellHeader) (((cellHeader) >> 4) & IX_OAM_VCI_BITS_MASK);
  get the VCI field from a cell header
- #define IX_ATM_CELL_HEADER_VPI_GET(cellHeader) (((cellHeader) >> 20) & IX_OAM_VPI_BITS_MASK);
  get the VPI field from a cell header
- #define IX_ATM_CELL_HEADER_PTI_GET(cellHeader) ((cellHeader) >> 1) & IX_OAM_PTI_BITS_MASK;
  get the PTI field from a cell header

A.9.0.3 Typedefs

- typedef unsigned int IxAtmCellHeader
  ATM Cell Header, does not contain 4 byte HEC, added by NPE-A.
- 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.9.0.4 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 informatoin 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.9.0.5 Detailed Description

The common set of types used in many Atm components.

A.9.0.6 Define Documentation

A.9.0.7 #define IX_ATM_CELL_HEADER_PTI_GET(cellHeader) ((cellHeader) >> 1) & IX_OAM_PTI_BITS_MASK;

get the PTI field from a cell header

Definition at line 209 of file IxAtmTypes.h.

A.9.0.8 #define IX_ATM_CELL_HEADER_VCI_GET(cellHeader) (((cellHeader) >> 4) & IX_OAM_VCI_BITS_MASK);

get the VCI field from a cell header

Definition at line 195 of file IxAtmTypes.h.

A.9.0.9 #define IX_ATM_CELL_HEADER_VPI_GET(cellHeader) (((cellHeader) >> 20) & IX_OAM_VPI_BITS_MASK);

get the VPI field from a cell header

Definition at line 202 of file IxAtmTypes.h.

A.9.0.10 #define IX_ATM_IDLE_CELLS_CONNID 0

VC Id used to indicate idle cells in the returned schedule table.

Definition at line 188 of file IxAtmTypes.h.

A.9.0.11 #define IX_ATM_MAX_NUM_OAM_RX_VCS 1

Maximum number of active OAM Rx VCs in the system, 1 OAM VC shared accross all ports.

Definition at line 170 of file IxAtmTypes.h.

A.9.0.12 #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.

Definition at line 163 of file IxAtmTypes.h.

A.9.0.13 Typedef Documentation

A.9.0.14 IxAtmConnId

ATM VC data connection identifier.

This is 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.

Definition at line 305 of file IxAtmTypes.h.

A.9.0.15 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.

Definition at line 330 of file IxAtmTypes.h.

A.9.0.16 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.

Definition at line 319 of file IxAtmTypes.h.

A.9.0.17 Enumeration Type Documentation

A.9.0.18 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. Definition at line 65 of file IxAtmTypes.h.

A.9.0.19 enum IxAtmRxQueueId

Rx Queue Type for RX traffic.

IxAtmRxQueueId defines the queues involved for receiving data.

There are two queues to facilitate prioritisation handling and processing the 2 queues with different algorithms and constraints

e.g. : 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.

Definition at line 254 of file IxAtmTypes.h.

A.9.0.20 enum IxAtmServiceCategory

Enumerated type representing available ATM service categories. For more informatoin on these categories, see "Traffic Management Specification" v4.1, published by the ATM Forum - 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).

Definition at line 228 of file IxAtmTypes.h.

A.10 IXP425 Security (IxCryptoAcc) API

IXP425 Security (IxCryptoAcc) API. IXP425 Security (IxCryptoAcc) APIIXP425 Security (IxCryptoAcc) APIIXP425 Security (IxCryptoAcc) APIIXP425 Security component Public API.

A.10.0.1 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.10.0.2 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 64
  Max length (byte) of authentication key for SHA1 and MD5.
- #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
  DES key length in bytes.
- #define IX_CRYPTO_ACC_DES_BLOCK_64 8
  DES cipher block length in bytes.
- #define IX_CRYPTO_ACC_DES_IV_64 8
  DES initialization vector length in bytes.
- #define IX_CRYPTO_ACC_3DES_KEY_192 24
  3DES key length in bytes
- #define IX_CRYPTO_ACC_AES_KEY_128 16
  AES-128 key length in bytes.
- #define IX_CRYPTO_ACC_AES_KEY_192 24
  AES-192 key length in bytes.
- #define IX_CRYPTO_ACC_AES_KEY_256 32
  AES-256 key length in bytes.
- #define IX_CRYPTO_ACC_AES_BLOCK_128 16
  AES cipher block length in bytes.
- #define IX_CRYPTO_ACC_AES_CBC_IV_128 16
  AES initialization vector length in bytes for CBC mode.
- #define IX_CRYPTO_ACC_AES_CTR_IV_64 8
  AES initialization vector length in bytes for CTR mode.
- #define IX_CRYPTO_ACC_SHA1_KEY_160 20
  SHA1 key length in bytes.
- #define IX_CRYPTO_ACC_SHA1_DIGEST_160 20
  SHA1 message digest length in bytes.
- #define IX_CRYPTO_ACC_MD5_KEY_128 16
  MD5 key length in bytes.
- #define IX_CRYPTO_ACC_MD5_DIGEST_128 16
  MD5 message digest length in bytes.

A.10.0.3 Typedefs

- typedef void(* IxCryptoAccRegisterCompleteCallback)(UINT32 cryptoCtxId, IX_MBUF *pMbuf, IxCryptoAccStatus status)
  Cryptographic Context registration complete callback notification.
- typedef void(* IxCryptoAccHashKeyGenCompleteCallback)(UINT32 hashKeyId, IX_MBUF *pMbufHashKey, IxCryptoAccStatus status)
  Hash key generation complete callback notification.
- typedef void(* IxCryptoAccPerformCompleteCallback)(UINT32 cryptoCtxId, IX_MBUF *pSrcMbuf, IX_MBUF *pDestMbuf, IxCryptoAccStatus status)

Harware accelerator service request complete callback notification.

A.10.0.4 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 }
  Status Definitions.

A.10.0.5 Functions

- PUBLIC IxCryptoAccStatus ixCryptoAccInit (void)
  Initialise the Security Access component.
- PUBLIC IxCryptoAccStatus ixCryptoAccCtxRegister (IxCryptoAccCtx *pAccCtx, IX_MBUF *pMbufPrimaryChainVar, IX_MBUF *pMbufSecondaryChainVar, IxCryptoAccRegisterCompleteCallback registerCallbackFn, IxCryptoAccPerformCompleteCallback performCallbackFn, UINT32 *pCryptoCtxId)

Crypto context registration. Cryptographic Context ID (cryptoCtxId) for the registered crypto context obtained from this registration request will be used in ixCryptoAccAuthCryptPerform requests.

- PUBLIC IxCryptoAccStatus ixCryptoAccCtxUnregister (UINT32 cryptoCtxId)

  Unregister the crypto context from Cryptographic Context Database.

- 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 functionalities.

- PUBLIC void ixCryptoAccShow (void)

  API for printing statistics 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.

- PUBLIC IxCryptoAccStatus ixCryptoAccHashKeyGenerate (IxCryptoAccAuthAlgo hashAlgo, IX_MBUF *pMbufHashKey, IxCryptoAccHashKeyGenCompleteCallback hashKeyCallbackFn, UINT16 hashKeyStartOffset, UINT16 hashKeyLen, UINT16 hashKeyDestOffset, UINT32 *pHashKeyId)

  This function is used to generate authentication key needed in HMAC authentication if the authentication key is greater than 64 bytes. New authentication key of L bytes size will be generated in this function (L = 20 for SHA1, L = 16 for MD5). Please refer to RFC2104 for more details on the key size. The authentication key is padded (extended) so that its length (in bits) is congruent to 448, modulo 512 (please refer to RFC1321).

A.10.0.6 Detailed Description

IXP425 Security component Public API.

A.10.0.7 Define Documentation

A.10.0.8 #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.

Definition at line 102 of file IxCryptoAcc.h.

A.10.0.9 #define IX_CRYPTO_ACC_MAX_AUTH_KEY_LENGTH 64

Max length (byte) of authentication key for SHA1 and MD5.

Key size > 64 bytes need to be hashed to produce shorter key by calling API ixCryptoAccHashKeyGenerate(). (SHA1 : 20 bytes, MD5 : 16 bytes by default according to RFC2104. If key size <= 64 bytes, authentication key can be used for registration directly.

Definition at line 76 of file IxCryptoAcc.h.

A.10.0.10 Typedef Documentation

A.10.0.11 typedef void(* IxCryptoAccHashKeyGenCompleteCallback)(UINT32 hashKeyId, IX_MBUF *pMbufHashKey, IxCryptoAccStatus status)

Hash key generation complete callback notification.

This function is called to notify a client that the hash key has been generated. This function will return the status through the associated hashKeyId once the key is calculated by Network Processor Engine (NPE).

hashKeyId becomes invalid once the notification callback is called.

Parameters

- *hashKeyId* [in] This hashKeyId is provided when client sends in request to hash the key.
- **pMbufHashKey* [in] Pointer to the mbuf that contains original key and generated hash key. Client will need to copy the generated hash key from the mbuf into crypto context and used it as hash key for crypto registration request. This mbuf will be freed by client.
- *status* [in] Status (IxCryptoAccStatus) reporting to client.

Notes:

- IX_CRYPTO_ACC_STATUS_SUCCESS - hash key generation is successful.
- IX_CRYPTO_ACC_STATUS_FAIL - hash key generation failed.

  Definition at line 517 of file IxCryptoAcc.h.

A.10.0.12 typedef void(* IxCryptoAccPerformCompleteCallback)(UINT32 cryptoCtxId, IX_MBUF *pSrcMbuf, IX_MBUF *pDestMbuf, IxCryptoAccStatus status)

Harware accelerator service request complete callback notification.

This function is called to notify a client that the cryptographic transaction has been completed. The cryptoCtxId and status of completed operation are returned to the client through this callback function to indicate operation which crypto context has been completed. The CryptoCtxId is obtained via ixCryptoAccCtxRegister, and this ID is unique to a particular IPSec tunnel.

Parameters

- *cryptoCtxId* [in] This crypto context ID is provided when client sends in request via ixCryptoAccCtxRegister API to register crypto context. cryptoCtxId points to a struct consists of cryptographic paramaters required by the Network Processor Engine (NPE) in CCD database.

- **pSrcMbuf* [in] Pointer to the source mbuf which contains the data to be processed. It is also the output mbuf which contains the processed data if UseDifferentSrcAndDestMbufs is FALSE.
- **pDestMbuf* [in] Only used if UseDifferentSrcAndDestMbufs is TRUE. Pointer to the output mbuf which contains processed data.
- *status* [in] Status reporting to the client via IxCryptoAccStatus.

Notes:

- IX_CRYPTO_ACC_STATUS_SUCCESS - Operation is completed successfully.
- IX_CRYPTO_ACC_STATUS_FAIL - Operation failed.
- IX_CRYPTO_ACC_STATUS_AUTH_FAIL - Authentication is unsuccessful. Note that when authentication fails, the content of the destination mbuf (which is the same as the source mbuf if a normal in-place operation is performed) will be undetermined.

Definition at line 557 of file IxCryptoAcc.h.

A.10.0.13 typedef void(* IxCryptoAccRegisterCompleteCallback)(UINT32 cryptoCtxId, IX_MBUF *pMbuf, IxCryptoAccStatus status)

Cryptographic Context registration complete callback notification.

This function is called to notify a client that the Crypto Context has been registered. This function returns status through the associated cryptoCtxId once the initial values needed are calculated by Network Processor Engine (NPE) and stored in Cryptographic Context Database (CCD).

The CryptoCtxId is valid until ixCryptoAccCtxUnregister is invoked.

If the callback function returns the IX_CRYPTO_ACC_STATUS_WAIT status, it indicates that registration is not complete yet, but the mbuf pointer needs to be freed by client. Client needs to wait for the next completion indication. Registration complete successfully only if status IX_CRYPTO_ACC_STATUS_SUCCESS is received.

Parameters

- *cryptoCtxId* [in] This crypto context ID is provided when client sends in request via ixCryptoAccCtxRegister API to register crypto context. cryptoCtxId points to a struct consists of cryptographic paramaters required by the Network Processor Engine (NPE) in CCD database.
- **pMbuf* [in] Pointer to the mbuf (to be freed by client). The client should free any mbuf that is not NULL.
- *status* [in] Status (IxCryptoAccStatus) reporting to client.

Notes:

- IX_CRYPTO_ACC_STATUS_SUCCESS - registration is successful.
- IX_CRYPTO_ACC_STATUS_FAIL - registration failed.
- IX_CRYPTO_ACC_STATUS_WAIT - registration is not complete yet, wait for next completion indication. NPE is busy in calculating the initial variables needed for the registration.

Definition at line 486 of file IxCryptoAcc.h.

A.10.0.14 Enumeration Type Documentation

A.10.0.15 enum IxCryptoAccAuthAlgo

Authentication Algorithm Definitions.

Note: Only two authentication algorithms are supported, SHA1 and MD5.

Enumeration Values

- ***IX_CRYPTO_ACC_AUTH_NULL*** NULL authentication.
- ***IX_CRYPTO_ACC_AUTH_SHA1*** SHA1 algorithm.
- ***IX_CRYPTO_ACC_AUTH_MD5*** MD5 algorithm.
- ***IX_CRYPTO_ACC_AUTH_TYPE*** Maximum value for types of authentication algorithm.

Definition at line 283 of file IxCryptoAcc.h.

A.10.0.16 enum IxCryptoAccCipherAlgo

Cipher Algorithm Definitions.

Note: 3DES and AES will be supported if not violating import/export rules.

Enumeration Values

- ***IX_CRYPTO_ACC_CIPHER_NULL*** NULL encryption.
- ***IX_CRYPTO_ACC_CIPHER_DES*** DES algorithm.
- ***IX_CRYPTO_ACC_CIPHER_3DES*** Triple DES algorithm.
- ***IX_CRYPTO_ACC_CIPHER_AES*** AES algorithm.
- ***IX_CRYPTO_ACC_CIPHER_ALGO_TYPE*** Maximum value for types of cipher algorithm.

Definition at line 238 of file IxCryptoAcc.h.

A.10.0.17 enum IxCryptoAccCipherMode

Cipher Mode Definitions.

Note: CFB and OFB are not supported.

Enumeration Values

- ***IX_CRYPTO_ACC_MODE_NULL*** NULL cipher mode.
- ***IX_CRYPTO_ACC_MODE_ECB*** ECB mode of operation.
- ***IX_CRYPTO_ACC_MODE_CBC*** CBC mode of operation.
- ***IX_CRYPTO_ACC_MODE_CTR*** CTR mode of operation, only applicable to AES.
- ***IX_CRYPTO_ACC_MODE_TYPE*** Maximum value for types of operation.

Definition at line 259 of file IxCryptoAcc.h.

A.10.0.18 enum IxCryptoAccOperation

Cryptographic Operation Definitions.

Enumeration Values

- ***IX_CRYPTO_ACC_OP_ENCRYPT*** Encrypt operation.
- ***IX_CRYPTO_ACC_OP_DECRYPT*** Decrypt operation.
- ***IX_CRYPTO_ACC_OP_AUTH_CALC*** Authentication calculation operation.
- ***IX_CRYPTO_ACC_OP_AUTH_CHECK*** Authentication verification operation.
- ***IX_CRYPTO_ACC_OP_ENCRYPT_AUTH*** Encryption followed by authentication calculation operation.
- ***IX_CRYPTO_ACC_OP_AUTH_DECRYPT*** Authentication verification followed by decryption.
- ***IX_CRYPTO_ACC_OP_TYPE_OF_OPERATION*** Maximum value for types of operation.

  Definition at line 202 of file IxCryptoAcc.h.

A.10.0.19 enum IxCryptoAccStatus

Status Definitions.

Note: These status will be used by the APIs to return to the client.

Enumeration Values

- ***IX_CRYPTO_ACC_STATUS_SUCCESS*** Success status.
- ***IX_CRYPTO_ACC_STATUS_FAIL*** Fail status.
- ***IX_CRYPTO_ACC_STATUS_WAIT*** Wait status.
- ***IX_CRYPTO_ACC_STATUS_RETRY*** Retry status.
- ***IX_CRYPTO_ACC_STATUS_QUEUE_FULL*** Queue full.
- ***IX_CRYPTO_ACC_STATUS_OPERATION_NOT_SUPPORTED*** Invalid operation.
- ***IX_CRYPTO_ACC_STATUS_CIPHER_ALGO_NOT_SUPPORTED*** Invalid cipher algorithm.
- ***IX_CRYPTO_ACC_STATUS_CIPHER_MODE_NOT_SUPPORTED*** Invalid cipher mode of operation.
- ***IX_CRYPTO_ACC_STATUS_CIPHER_INVALID_KEY_LEN*** Invalid cipher key length.
- ***IX_CRYPTO_ACC_STATUS_CIPHER_INVALID_IV_LEN*** Invalid IV length.
- ***IX_CRYPTO_ACC_STATUS_CIPHER_INVALID_BLOCK_LEN*** Invalid cipher block length.
- ***IX_CRYPTO_ACC_STATUS_AUTH_ALGO_NOT_SUPPORTED*** Invalid authentication algorithm.

- ***IX_CRYPTO_ACC_STATUS_AUTH_INVALID_DIGEST_LEN*** Invalid message digest length.
- ***IX_CRYPTO_ACC_STATUS_AUTH_INVALID_KEY_LEN*** Invalid authentication key length.
- ***IX_CRYPTO_ACC_STATUS_AUTH_FAIL*** Authentication verification failed.
- ***IX_CRYPTO_ACC_STATUS_CRYPTO_CTX_NOT_VALID*** Invalid crypto context ID.
- ***IX_CRYPTO_ACC_STATUS_EXCEED_MAX_TUNNELS*** Exceed maximum number of crypto contexts allocation.

  Definition at line 303 of file IxCryptoAcc.h.

A.10.0.20 Function Documentation

A.10.0.21 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 functionalities.

This function is called for authentication and decryption/encryption functionalities service request. 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 within an ICV field.

Parameters

- *cryptoCtxId* [in] is the crypto context pointer to be supplied by the client. This cryptoCtxId is obtained via ixCryptoAccCtxRegister. The cryptoCtxId must be a valid Id.
- **pSrcMbuf* [in] is a pointer to mbuf which contains data to be processed. This mbuf structure is allocated by client. Result of this request will be stored in the same mbuf and overwritten the original data if UseDifferentSrcAndDestMbufs flag in IxCryptoAccCtx is set to FALSE (in-place operation). Otherwise, if UseDifferentSrcAndDestMbufs flag is set to TRUE, the result will be written into destination mbuf (non in-place operation) and the original data in this mbuf will remain unchanged. The same pointer is then returned to the client via registered IxCryptoAccPerformCompleteCallback callback function.
- **pDestMbuf* [in] only used if UseDifferentSrcAndDestMbufs is TRUE. This is the buffer where the result is written to. This mbuf structure is allocated by client. 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. The same pointer is then returned to the client via registered IxCryptoAccPerformCompleteCallback callback function.
- *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 bytes.

- *cryptStartOffset* [in] supplied by the client to indicate the start of the payload to be decrypted/ encrypted.
- *cryptDataLen* [in] supplied by the client to indicate the length of the payload to be decrypted/ encrypted in Bytes. The payload to be decrypted/encrypted must have the length that is multiple of cipher block length in size.
- *icvOffset* [in] supplied by the client to indicate the start of the ICV (Integrity Check Value) used for the authentication. This ICV field should not be splitted across multiple mbufs in a chained mbuf.
- **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.

Notes:

- Client shall handle IP mutable fields.
- Client shall pad the IP datagram to be a mutltiple of cipher block size, using ESP trailer for encryption (RFC2406, explicit padding)
- NPE shall pad the IP datagram to be a multiple of hashing block size, specified by the authentication algorithm (RFC2402, implicit padding)
- For authentication generation operation, client needs to clear the field which hold the authentication data (ICV) to zeroes. While for the authentication verification operation, client needs to supply the authentication data at the ICV field for NPE to verify the ICV value. If ICV is embedded in the payload, client DOES NOT need to move the ICV to the front / the back of the payload and clear the original ICV field. cryptoAcc access component will handle this.
- AES CTR couter block implementation is based on internet draft, draft-ietf-ipsec-ciph-aes-ctr-00.txt (July 2002), which can be found at IETF website, www.ietf.org

Reentrant : no

ISR Callable : yes

A.10.0.22 IxCryptoAccStatus 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 queues will be completed before the security hardware accelerator halt. Any new requests issued after this call will be rejected. All the Crypto Contexts will be unregistered in this function call.

Parameters

None

Returns

- IX_CRYPTO_ACC_STATUS_SUCCESS - Operation requested is successful with all the pending requests are completed and CCD is cleared.
- IX_CRYPTO_ACC_STATUS_FAIL - Cryptographic stop request failed for some unspecified internal reasons.
- Reentrant : no
- ISR Callable : no

A.10.0.23 IxCryptoAccStatus ixCryptoAccCtxRegister (IxCryptoAccCtx * *pAccCtx*, IX_MBUF * *pMbufPrimaryChainVar*, IX_MBUF * *pMbufSecondaryChainVar*, IxCryptoAccRegisterCompleteCallback *registerCallbackFn*, IxCryptoAccPerformCompleteCallback *performCallbackFn*, UINT32 * *pCryptoCtxId*)

Crypto context registration. Cryptographic Context ID (cryptoCtxId) for the registered crypto context obtained from this registration request will be used in ixCryptoAccAuthCryptPerform requests.

This function is used to register all the required information (eg. key, cipher algorithm, etc) for hardware accelerator services with IxCryptoAcc component. Those information will be stored in CCD database. All the information will be converted into a cryptographic parameters structure to be shared with NPE. The structure will be associated with a unique crypto context ID (cryptoCtxId). cryptoCtxId is passed back to client for future reference in callback function and also for hardware accelerator service requests. 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.

Note: Two scenarios of crypto context registration depending on authentication key size (in bytes) : 1. If (L <= key size <= 64), then call this API (ixCryptoAccCtxRegister) directly. L = 16 for MD5 and L = 20 for SHA1. 2. If (key size > 64), then authentication key needs to be hashed to become shorter key first by calling another API ixCryptoAccHashKeyGenerate (). Please follow steps below for this case:

1. Call ixCryptoAccHashKeyGenerate () to hash the authentication key
2. Wait for the callback from ixCryptoAccHashKeyGenerate (). Copy generated authentication key from mbuf into IxCryptoAccCtx as authentication key.
3. Call this API (ixCryptoAccCtxRegister)to register the context.

Parameters

- **pAccCtx* [in] is a pointer to hardware accelerator context. Information required in hardware accelerator context, such as key and algorithm configuration can be extracted from Security Association Database (SAD).
- **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 (mininum 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 (mininum 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 crypto context registration status to client when the registration is complete. 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 are 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 algorithm 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.10.0.24 IxCryptoAccStatus ixCryptoAccCtxUnregister (UINT32 *cryptoCtxId*)

Unregister the crypto context from Cryptographic Context Database.

This function is for freeing the particular crypto context (reference through CryptoCtxId) from the Cryptographic Context Database.

Parameters

cryptoCtxId [in] is pointer to crypto context 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 crypto context.
- IX_CRYPTO_ACC_STATUS_FAIL - Unregistration failed for some unspecified internal reasons, e.g. uninitialized.
- IX_CRYPTO_ACC_STATUS_CRYPTO_CTX_NOT_VALID - invalid crypto context.
- 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.10.0.25 IxCryptoAccStatus ixCryptoAccHashKeyGenerate (IxCryptoAccAuthAlgo *hashAlgo*, IX_MBUF * *pMbufHashKey*, IxCryptoAccHashKeyGenCompleteCallback *hashKeyCallbackFn*, UINT16 *hashKeyStartOffset*, UINT16 *hashKeyLen*, UINT16 *hashKeyDestOffset*, UINT32 * *pHashKeyId*)

This function is used to generate authentication key needed in HMAC authentication if the authentication key is greater than 64 bytes. New authentication key of L bytes size will be generated in this function (L = 20 for SHA1, L = 16 for MD5). Please refer to RFC2104 for more details on the key size. The authentication key is padded (extended) so that its length (in bits) is congruent to 448, modulo 512 (please refer to RFC1321).

Notes:

- This API will be called ONLY if the authentication key > 64 bytes. If key size is >=L and <= 64 bytes, authentication key is used directly in Crypto context for registration, no further hashing is needed.

- This function must be called prior the Crypto Context registration if the key size > 64. Result from this function will be used as authentication key in the Crypto Context registration.
- It is client's responsibility to ensure the mbuf is big enough to hold the result (generated authentication key). No checking on mbuf length will be done.

Parameters

- *hashAlgo* [in] is hashing algorithm to be used in generating the hash key. hashAlgo used to generate the key must be the same algorithm that will be used in crypto register and crypto perform services.
- **pMbufHashKey* [in] a pointer to an mbuf that contains the authentication key to be hashed. This mbuf structure is allocated by the client and the mbuf pointer must not be NULL. After the NPE complete the computation, the mbuf is returned separately through the client's registered callback. The authentication key generated will be stored in this mbuf, pointed by hashKeyDestOffset. This must NOT be a chained mbuf.
- *hashKeyCallbackFn* [in] callback function pointer to be called when the hash key generation operation is completed. This cannot be NULL.
- *hashKeyStartOffset* [in] offset to the mbuf mdata which contain the original hash key.
- *hashKeyLen* [in] key size
- *hashKeyDestOffset* [in] offset to the mbuf mdata to store the generated result (authentication key)
- **pHashKeyId:* [inout] Hash Key ID returned by access component to identify hash key generation.

Returns

- IX_CRYPTO_ACC_STATUS_SUCCESS - Hash key generation parameters are valid
- IX_CRYPTO_ACC_STATUS_FAIL - Hash key generation failed.
- IX_CRYPTO_ACC_STATUS_AUTH_ALGO_NOT_SUPPORTED - Invalid hash algorithm requested by the client.
- IX_CRYPTO_ACC_STATUS_AUTH_INVALID_KEY_LEN - Invalid hash key length (key size <= 64 bytes for MD5/SHA1). If this error status is received, it means that the authentication key (<= 64 bytes) can be used directly for crypto register function (ixCryptoAccCtxRegister), the key does not need to be hashed to become shorter key.
- IX_CRYPTO_ACC_STATUS_QUEUE_FULL - Queue full status returned and the authentication key generation request will be rejected.

Reentrant : no

ISR Callable : no

A.10.0.26 IxCryptoAccStatus ixCryptoAccInit (void)

Initialise the Security Access component.

This API will initialize all the internal resources of IxCryptoAcc access component. It will check whether the specified AES or DES coprocessor is present and issue a warning if not. This API must be called before any other API call on this component can be made. This API is called once only.

Parameters

None

Returns

- IX_CRYPTO_ACC_STATUS_SUCCESS - successfully initialize the component; a warning is issued if coprocessor is not present.
- IX_CRYPTO_ACC_STATUS_FAIL - Initialization failed for some unspecified internal reasons.

Reentrant : no

ISR Callable : no

A.10.0.27 void ixCryptoAccShow (void)

API for printing statistics 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.

Parameters

None

Returns

None

Reentrant : no

ISR Callable : no

A.10.0.28 void 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

Returns

None

Reentrant : no

ISR Callable : no

A.11 IXP425 DMA Types (IxDmaTypes)

IXP425 DMA Types (IxDmaTypes). IXP425 DMA Types (IxDmaTypes)IXP425 DMA Types (IxDmaTypes)IXP425 DMA Types (IxDmaTypes)The common set of types used in the DMA component.

A.11.0.1 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 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.11.0.2 Detailed Description

The common set of types used in the DMA component.

A.11.0.3 Enumeration Type Documentation

A.11.0.4 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.
- ***IX_DMA_ADDRESSING_MODE_INVALID*** Invalid Addressing Mode.

  Definition at line 95 of file IxDmaAcc.h.

A.11.0.5 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.

Definition at line 137 of file IxDmaAcc.h.

A.11.0.6 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.

  Definition at line 63 of file IxDmaAcc.h.

A.11.0.7 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
- ***IX_DMA_TRANSFER_MODE_INVALID*** Invalid transfer mode.

Definition at line 80 of file IxDmaAcc.h.

A.11.0.8 enum IxDmaTransferWidth

Dma transfer width definitions 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
- ***IX_DMA_TRANSFER_WIDTH_INVALID*** Invalid transfer width.

  Definition at line 110 of file IxDmaAcc.h.

A.12 IXP425 DMA Access Driver (IxDmaAcc) API

IXP425 DMA Access Driver (IxDmaAcc) API. IXP425 DMA Access Driver (IxDmaAcc) APIIXP425 DMA Access Driver (IxDmaAcc) APIIXP425 DMA Access Driver (IxDmaAcc) APIThe public API for the IXP425 IxDmaAcc component.

A.12.0.1 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.12.0.2 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.12.0.3 Functions

- PUBLIC IX_STATUS ixDmaAccInit (IxNpeDlNpeId npeId)

  Initialise the DMA Access component This function will initialise 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 IXP425 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.12.0.4 Detailed Description

The public API for the IXP425 IxDmaAcc component.

A.12.0.5 Define Documentation

A.12.0.6 #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.

Definition at line 167 of file IxDmaAcc.h.

A.12.0.7 Typedef Documentation

A.12.0.8 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

Definition at line 176 of file IxDmaAcc.h.

A.12.0.9 typedef UINT32 IxDmaAccRequestId

DMA Request Id type.

Definition at line 157 of file IxDmaAcc.h.

A.12.0.10 Function Documentation

A.12.0.11 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 IXP425 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 suport burst transfer width
- Fixed source address to fixed destinatiom address mode is not suported
- 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 IXP425 memory map address.
- *ixDmaDestinationAddr* Starting address of DMA destination. Must be a valid IXP425 memory map address.

- *ixDmaTransferLength* The size of DMA data transfer. The range must be from 1-64Kbyte
- *ixDmaTransferMode* The DMA transfer mode
- *ixDmaAddressingMode* The DMA addressing mode
- *ixTransferWidth* The DMA transfer width

Returns

- IX_DMA_SUCCESS Notification that the DMA request is succesful
- IX_DMA_FAIL IxDmaAcc not yet initialised or some internal error has occured
- IX_DMA_INVALID_TRANSFER_WIDTH Transfer width is nit 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.12.0.12 ixDmaAccInit (IxNpeDlNpeId *npeId*)

Initialise the DMA Access component This function will initialise the DMA Access component internals.

Parameters

npeId - NPE to use for Dma Transfer

Returns

- IX_SUCCESS succesfully initialised the component
- IX_FAIL Initialisation failed for some unspecified internal reason.

A.12.0.13 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.13 IXP425 Ethernet Access (IxEthAcc) API

IXP425 Ethernet Access (IxEthAcc) API. IXP425 Ethernet Access (IxEthAcc) APIIXP425 Ethernet Access (IxEthAcc) APIIXP425 Ethernet Access (IxEthAcc) APIethAcc is a library that does provides access to the internal IXP425 10/100Bt Ethernet MACs.

A.13.0.1 Data Structures

- 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.

A.13.0.2 Defines

- #define IX_ETH_ACC_NUMBER_OF_PORTS (2)
  Defines related to the number of NPE's and mapping between PortId and NPE.
- #define IX_IEEE803_MAC_ADDRESS_SIZE (6)
  Defines the size of the MAC address NPE.
- #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 highest MII address of any attached PHYs.
- #define ixEthAccMiiPhyScan(phyPresent) ixEthMiiPhyScan(phyPresent,IXP425_ETH_ACC_MII_MAX_ADDR)
  : deprecated API entry point. This definition ensures backward compatibility
- #define ixEthAccMiiPhyConfig(phyAddr, speed100, fullDuplex, autonegotiate) ixEthMiiPhyConfig(phyAddr,speed100,fullDuplex,autonegotiate)
  : deprecated API entry point. This definition ensures backward compatibility
- #define ixEthAccMiiPhyReset(phyAddr) ixEthMiiPhyReset(phyAddr)
  : deprecated API entry point. This definition ensures backward compatibility
- #define ixEthAccMiiLinkStatus(phyAddr, linkUp, speed100, fullDuplex, autoneg) ixEthMiiLinkStatus(phyAddr,linkUp,speed100,fullDuplex,autoneg)
  : deprecated API entry point. This definition ensures backward compatibility
- #define ixEthAccMiiShow(phyAddr) ixEthMiiPhyShow(phyAddr)
  : deprecated API entry point. This definition ensures backward compatibility

A.13.0.3 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 ixEthAccPortRxCallbackRegister .

A.13.0.4 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 IXP425 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 }

  Definition to provision the duplex mode of the MAC.

- enum IxEthAccTxSchedulerDiscipline { FIFO_NO_PRIORITY, FIFO_PRIORITY }

  Definition for the port transmit scheduling discipline Definition for the port transmit scheduling discipline.

A.13.0.5 Functions

- IxEthAccStatus ixEthAccInit (void)

  Initialize the Ethernet Access Service.

- void ixEthAccUnload (void)

  Unload 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 ixEthAccPortMulticastAddressJoinAll (IxEthAccPortId portId)
  Filter all frames with multicast dest.
- IxEthAccStatus ixEthAccPortMulticastAddressLeave (IxEthAccPortId portId, IxEthAccMacAddr *macAddr)
  Remove a multicast address from the MAC address table.
- IxEthAccStatus ixEthAccPortMulticastAddressLeaveAll (IxEthAccPortId portId)
  Clear the MAC address table.
- IxEthAccStatus ixEthAccPortUnicastAddressShow (IxEthAccPortId portId)
  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 ixEthAccPortTxFrameAppendPaddingEnable (IxEthAccPortId portId)
  Enable the appending of padding bytes to runt frames submitted to this port.
- IxEthAccStatus ixEthAccPortTxFrameAppendPaddingDisable (IxEthAccPortId portId)
  Disable the appending of padding bytes to the runt frames submitted to this port.
- 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.
- IxEthAccStatus ixEthAccMacInit (IxEthAccPortId portId)
  Initialize the ethernet MAC settings.
- 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 ixEthAccMiiReadRtn (UINT8 phyAddr, UINT8 phyReg, UINT16 *value)
  Read a 16 bit value from a PHY.
- IxEthAccStatus ixEthAccMiiWriteRtn (UINT8 phyAddr, UINT8 phyReg, UINT16 value)
  Write a 16 bit value to a PHY.
- IxEthAccStatus ixEthAccMiiStatsShow (UINT32 phyAddr)
  Display detailed information on a specified PHY.

A.13.0.6 Detailed Description

ethAcc is a library that does provides access to the internal IXP425 10/100Bt Ethernet MACs.

A.13.0.7 Define Documentation

A.13.0.8 #define IX_ETH_ACC_NUM_TX_PRIORITIES (8)

The number of transmit priorities.

Definition at line 121 of file IxEthAcc.h.

A.13.0.9 #define IX_ETH_ACC_NUMBER_OF_PORTS (2)

Defines related to the number of NPE's and mapping between PortId and NPE.

Definition at line 90 of file IxEthAcc.h.

A.13.0.10 #define IX_ETHACC_RX_MBUF_MIN_SIZE (2048)

This defines the required size of MBUF's submitted to the frame receive service.

Definition at line 163 of file IxEthAcc.h.

A.13.0.11 #define IX_IEEE803_MAC_ADDRESS_SIZE (6)

Defines the size of the MAC address NPE.

Definition at line 100 of file IxEthAcc.h.

A.13.0.12 #define ixEthAccMiiLinkStatus(phyAddr, linkUp, speed100, fullDuplex, autoneg) ixEthMiiLinkStatus(phyAddr,linkUp,speed100,fullDuplex,autoneg)

: deprecated API entry point. This definition ensures backward compatibility

See ixEthMiiLinkStatus

Note: this feature is board specific.

Definition at line 1651 of file IxEthAcc.h.

A.13.0.13 #define ixEthAccMiiPhyConfig(phyAddr, speed100, fullDuplex, autonegotiate) ixEthMiiPhyConfig(phyAddr,speed100,fullDuplex,autonegotiate)

: deprecated API entry point. This definition ensures backward compatibility

See ixEthMiiPhyConfig

Note: this feature is board specific

Definition at line 1621 of file IxEthAcc.h.

A.13.0.14 #define ixEthAccMiiPhyReset(phyAddr) ixEthMiiPhyReset(phyAddr)

: deprecated API entry point. This definition ensures backward compatibility

See ixEthMiiPhyReset

Note: this feature is board specific

Definition at line 1636 of file IxEthAcc.h.

A.13.0.15 #define ixEthAccMiiPhyScan(phyPresent) ixEthMiiPhyScan(phyPresent,IXP425_ETH_ACC_MII_MAX_ADDR)

: deprecated API entry point. This definition ensures backward compatibility

See ixEthMiiPhyScan

Note: this feature is board specific

Definition at line 1607 of file IxEthAcc.h.

A.13.0.16 #define ixEthAccMiiShow(phyAddr) ixEthMiiPhyShow(phyAddr)

deprecated API entry point. This definition ensures backward compatibility

See ixEthMiiPhyShow

Note: this feature is board specific

Definition at line 1668 of file IxEthAcc.h.

A.13.0.17 #define IXP425_ETH_ACC_MII_MAX_ADDR 32

This defines the highest MII address of any attached PHYs.

Max number for PHY address is 31, add on for range checking.

Definition at line 173 of file IxEthAcc.h.

A.13.0.18 Typedef Documentation

A.13.0.19 typedef void(* IxEthAccPortRxCallback)(UINT32 callbackTag, IX_MBUF *buffer, IxEthAccPortId portId)

Function prototype for Ethernet Frame Rx callback. Registered via *ixEthAccPortRxCallbackRegister* .

It is the responsibility of the user function to free any MBUF's which it receives.

Reentrant - yes , The user provided function should be reentrant.

ISR Callable - yes , The 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 IXP425 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 *ixEthAccPortRxCallbackRegister* . It allows the same callback to be used for multiple MACs.
- *mbuf* - Pointer to the Rx mbuf descriptor.
- *portId* - ID of the port which match the destination MAC address for this frame (The value is greater or equal to IX_ETH_DB_NUMBER_OF_PORTS if the MAC address is not found in the copy of the database shared with the NPE)

Returns

void

Warning: Any portID returned via the received callback is invalid if greater than 5. If your system defines more than 6 ports (0 to 5) in IxEthDBPortDefs.h you MUST check this value before using it in conjunction with the Ethernet Learning/Filtering Database. This is a constraint of an NPE Rx descriptor format limitation, which can report valid ports only between 0 and 5, and will use 6 for "reserved" and "7" for "not found", while the XScale Ethernet Database can use these numbers for legitimate ports.

Definition at line 439 of file IxEthAcc.h.

A.13.0.20 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 , The user provided function should be reentrant.

ISR Callable - yes , The 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 IXP425 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 *ixEthAccPortRxCallbackRegister* . It allows the same callback to be used for multiple MACs.
- *mbuf* - Pointer to the Tx mbuf descriptor.

Returns

void

Definition at line 351 of file IxEthAcc.h.

A.13.0.21 Enumeration Type Documentation

A.13.0.22 enum IxEthAccDuplexMode

Definition to provision the duplex mode of the MAC.

Enumeration Values

- ***IX_ETH_ACC_FULL_DUPLEX*** Full duplex operation of the MAC.
- ***IX_ETH_ACC_HALF_DUPLEX*** Half duplex operation of the MAC.

Definition at line 150 of file IxEthAcc.h.

A.13.0.23 enum IxEthAccPortId

This is an emum to define the IXP425 Mac Ethernet device.

Enumeration Values

- ***IX_ETH_PORT_1*** Ethernet Port 1.
- ***IX_ETH_PORT_2*** Ethernet port 2.

Definition at line 76 of file IxEthAcc.h.

A.13.0.24 enum IxEthAccStatus

This is an enum to define the Ethernet Access status.

Enumeration Values

- ***IX_ETH_ACC_SUCCESS*** return success
- ***IX_ETH_ACC_FAIL*** return fail
- ***IX_ETH_ACC_INVALID_PORT*** return invalid port
- ***IX_ETH_ACC_PORT_UNINITIALIZED*** return uninitialized
- ***IX_ETH_ACC_MAC_UNINITIALIZED*** return MAC uninitialized
- ***IX_ETH_ACC_INVALID_ARG*** return invalid arg
- ***IX_ETH_TX_Q_FULL*** return tx queue is full
- ***IX_ETH_ACC_NO_SUCH_ADDR*** return no such address

Definition at line 59 of file IxEthAcc.h.

A.13.0.25 enum IxEthAccTxPriority

enum to submit a frame with relative priority.

Enumeration Values

- ***IX_ETH_ACC_TX_PRIORITY_0*** Lowest Priorty submission.
- ***IX_ETH_ACC_TX_PRIORITY_1*** submission prority of 1 (0 is lowest)

- ***IX_ETH_ACC_TX_PRIORITY_2*** submission prority of 2 (0 is lowest)
- ***IX_ETH_ACC_TX_PRIORITY_3*** submission prority of 3 (0 is lowest)
- ***IX_ETH_ACC_TX_PRIORITY_4*** submission prority of 4 (0 is lowest)
- ***IX_ETH_ACC_TX_PRIORITY_5*** submission prority of 5 (0 is lowest)
- ***IX_ETH_ACC_TX_PRIORITY_6*** submission prority of 6 (0 is lowest)
- ***IX_ETH_ACC_TX_PRIORITY_7*** Highest priority submission.
- ***IX_ETH_ACC_TX_DEFAULT_PRIORITY*** By default send all pkts with lowest priority. Definition at line 129 of file IxEthAcc.h.

A.13.0.26 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 sub-mitted 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 prior-ity 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.

Enumeration Values

- ***FIFO_NO_PRIORITY*** frames submitted with no priority
- ***FIFO_PRIORITY*** higher prority frames submitted before lower priority

Definition at line 1275 of file IxEthAcc.h.

A.13.0.27 Function Documentation

A.13.0.28 ixEthAccInit (void)

Initialize the Ethernet Access Service.

Initialize the IXP425 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.13.0.29 ixEthAccMacInit (IxEthAccPortId *portId*)

Initialize the ethernet MAC settings.

* - Reentrant - no

ISR Callable - no

Parameters

portId IxEthAccPortId

Returns

IxEthAccStatus

- IX_ETH_ACC_SUCCESS
- IX_ETH_ACC_INVALID_PORT : portId is invalid.

A.13.0.30 ixEthAccMibIIStatsClear (IxEthAccPortId *portId*)

Clear the statistics maintained for a port. Clear the statistics maintained for a port.

Reentrant - yes

ISR Callable - no

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.13.0.31 ixEthAccMibIIStatsGet (IxEthAccPortId *portId*, IxEthEthObjStats * *retStats*)

Return the statistics maintained for a port. Return the statistics maintained for a port.

Reentrant - yes

ISR Callable - no

Parameters

- *portId IxEthAccPortId*
- *retStats IxEthEthObjStats*

Note: The user is responsible for cache coheriency of the retStat buffer. The data is actually populated via the NPE's. 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.13.0.32 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

Parameters

- *portId IxEthAccPortId*
- *retStats IxEthEthObjStats*

Note: The user is responsible for cache coheriency of the retStats buffer. The data is actually populated via the NPE's. 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.13.0.33 ixEthAccMiiReadRtn (UINT8 *phyAddr*, UINT8 *phyReg*, UINT16 * *value*)

Read a 16 bit value from a PHY.

Read a 16-bit word from a register of a MII-compliant PHY. Reading is performed through the MII management interface. This function returns when the read has successfully completed, or when a timeout has elapsed.

Reentrant - no

ISR Callable - no

Precondition

The MAC on Ethernet Port 2 (NPE C) must be initialised, and generating the MDIO clock.

Parameters

- *phyAddr:* the address of the Ethernet PHY (0-31)
- *phyReg:* the number of the MII register to read (0-31)
- *value:* the value read from the register

Returns

IxEthAccStatus

- IX_ETH_ACC_SUCCESS
- IX_ETH_ACC_FAIL : failed to read register.

A.13.0.34 ixEthAccMiiStatsShow (UINT32 *phyAddr*)

Display detailed information on a specified PHY.

Display the current values of the first eigth MII registers for a PHY,

Reentrant - no

ISR Callable - no

Precondition

The MAC on Ethernet Port 2 (NPE C) must be initialised, 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.13.0.35 ixEthAccMiiWriteRtn (UINT8 *phyAddr*, UINT8 *phyReg*, UINT16 *value*)

Write a 16 bit value to a PHY.

Write a 16-bit word from a register of a MII-compliant PHY. Writing is performed through the MII management interface. This function returns when the write has successfully completed, or when a timeout has elapsed.

Reentrant - no

ISR Callable - no

Precondition

The MAC on Ethernet Port 2 (NPE C) must be initialised, and generating the MDIO clock.

Parameters

- *phyAddr:* the address of the Ethernet PHY (0-31)
- *phyReg:* the number of the MII register to write (0-31)
- *value:* the value to write to the register

Returns

IxEthAccStatus

- IX_ETH_ACC_SUCCESS
- IX_ETH_ACC_FAIL : failed to write register.

A.13.0.36 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.13.0.37 ixEthAccPortDuplexModeGet (IxEthAccPortId *portId*, IxEthAccDuplexMode * *mode*)

Get the duplex mode for the MAC.

return the duplex configuration of the IXP425 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.13.0.38 ixEthAccPortDuplexModeSet (IxEthAccPortId *portId*, IxEthAccDuplexMode *mode*)

Set the duplex mode for the MAC.
Configure the IXP425 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.13.0.39 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 The rx and Tx Done callbacks registration via *ixEthAccPortTxDoneCallbackRegister* amd *ixEthAccPortRxCallbackRegister* has to be done before enabling the traffic.

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.13.0.40 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.13.0.41 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. Verify that the Ethernet port is present before initializing

Reentrant - no

ISR Callable - no

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: if the ethernet port is not present, a warning will be issued.
- IX_ETH_ACC_FAIL : The NPE processor has failed to initialize.
- IX_ETH_ACC_INVALID_PORT : portId is invalid.

A.13.0.42 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 NPE's. This is a result of the hardware comparison algorithm used in the destination mac address logic within the Ethernet MAC.

See Also: IXP425 hardware development manual.

Other functions modify the MAC filtering include:

- *ixEthAccPortMulticastAddressJoinAll()* - all multicast frames are forwarded to the application

- *ixEthAccPortMulticastAddressLeaveAll()* - rollback the effects of *ixEthAccPortMulticastAddressJoinAll()*
- *ixEthAccPortMulticastAddressLeave()* - unprovision a new filtering address
- *ixEthAccPortMulticastAddressJoin()* - provision a new filtering address
- *ixEthAccPortPromiscuousModeSet()* - all frames are forwarded to the application regardless of the multicast address provisioned
- *ixEthAccPortPromiscuousModeClear()* - frames are forwarded to the application following the multicast address provisioned

In all cases, unicast and broadcast addresses are forwarded to the application.

Reentrant - no

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.13.0.43 ixEthAccPortMulticastAddressJoinAll (IxEthAccPortId *portId*)

Filter all frames with multicast dest.

This function clears the MAC address table, and then sets the MAC to forward ALL multicast frames to the NPE. Specifically, it forwards all frames whose destination address has the LSB of the highest byte set (01:00:00:00:00:00). This bit is commonly referred to as the "multicast bit". Broadcast frames will still be forwarded.

Other functions modify the MAC filtering include:

- *ixEthAccPortMulticastAddressJoinAll()* - all multicast frames are forwarded to the application
- *ixEthAccPortMulticastAddressLeaveAll()* - rollback the effects of *ixEthAccPortMulticastAddressJoinAll()*
- *ixEthAccPortMulticastAddressLeave()* - unprovision a new filtering address
- *ixEthAccPortMulticastAddressJoin()* - provision a new filtering address
- *ixEthAccPortPromiscuousModeSet()* - all frames are forwarded to the application regardless of the multicast address provisioned
- *ixEthAccPortPromiscuousModeClear()* - frames are forwarded to the application following the multicast address provisioned

In all cases, unicast and broadcast addresses are forwarded to the application.

Reentrant - no

ISR Callable - no

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.13.0.44 ixEthAccPortMulticastAddressLeave (IxEthAccPortId *portId*, IxEthAccMacAddr * *macAddr*)

Remove a multicast address from the MAC address table.

Remove a multicast address from the MAC address table.

Other functions modify the MAC filtering include:

- *ixEthAccPortMulticastAddressJoinAll()* - all multicast frames are forwarded to the application
- *ixEthAccPortMulticastAddressLeaveAll()* - rollback the effects of *ixEthAccPortMulticastAddressJoinAll()*
- *ixEthAccPortMulticastAddressLeave()* - unprovision a new filtering address
- *ixEthAccPortMulticastAddressJoin()* - provision a new filtering address
- *ixEthAccPortPromiscuousModeSet()* - all frames are forwarded to the application regardless of the multicast address provisioned
- *ixEthAccPortPromiscuousModeClear()* - frames are forwarded to the application following the multicast address provisioned

In all cases, unicast and broadcast addresses are forwarded to the application.

Reentrant - no

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.13.0.45 ixEthAccPortMulticastAddressLeaveAll (IxEthAccPortId *portId*)

Clear the MAC address table.

This function clears the MAC address table, and then sets the MAC as configured by the promiscuous mode current settings.

Other functions modify the MAC filtering include:

- *ixEthAccPortMulticastAddressJoinAll()* - all multicast frames are forwarded to the application
- *ixEthAccPortMulticastAddressLeaveAll()* - rollback the effects of *ixEthAccPortMulticastAddressJoinAll()*
- *ixEthAccPortMulticastAddressLeave()* - unprovision a new filtering address
- *ixEthAccPortMulticastAddressJoin()* - provision a new filtering address
- *ixEthAccPortPromiscuousModeSet()* - all frames are forwarded to the application regardless of the multicast address provisioned
- *ixEthAccPortPromiscuousModeClear()* - frames are forwarded to the application following the multicast address provisioned

In all cases, unicast and broadcast addresses are forwarded to the application.

Reentrant - no

ISR Callable - no

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.13.0.46 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.13.0.47 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*
  See also: *ixEthAccPortPromiscuousModeSet*

Other functions modify the MAC filtering include:

- *ixEthAccPortMulticastAddressJoinAll()* - all multicast frames are forwarded to the application
- *ixEthAccPortMulticastAddressLeaveAll()* - rollback the effects of *ixEthAccPortMulticastAddressJoinAll()*
- *ixEthAccPortMulticastAddressLeave()* - unprovision a new filtering address
- *ixEthAccPortMulticastAddressJoin()* - provision a new filtering address
- *ixEthAccPortPromiscuousModeSet()* - all frames are forwarded to the application regardless of the multicast address provisioned
- *ixEthAccPortPromiscuousModeClear()* - frames are forwarded to the application following the multicast address provisioned

In all cases, unicast and broadcast addresses are forwarded to the application.

Reentrant - yes

ISR Callable - no

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.13.0.48 ixEthAccPortPromiscuousModeSet (IxEthAccPortId *portId*)

Put the MAC device in promiscuous mode.

If the device is in promiscuous mode then all all received frames shall be forwared to the NPE for processing.

See also: *ixEthAccPortPromiscuousModeClear*

Other functions modify the MAC filtering include:

- *ixEthAccPortMulticastAddressJoinAll()* - all multicast frames are forwarded to the application
- *ixEthAccPortMulticastAddressLeaveAll()* - rollback the effects of *ixEthAccPortMulticastAddressJoinAll()*
- *ixEthAccPortMulticastAddressLeave()* - unprovision a new filtering address
- *ixEthAccPortMulticastAddressJoin()* - provision a new filtering address
- *ixEthAccPortPromiscuousModeSet()* - all frames are forwarded to the application regardless of the multicast address provisioned
- *ixEthAccPortPromiscuousModeClear()* - frames are forwarded to the application following the multicast address provisioned

In all cases, unicast and broadcast addresses are forwarded to the application.

Reentrant - yes

ISR Callable - no

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.13.0.49 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 MBUF's supplied via this callback. In addition the registered callback must free up MBUF's from the recieve 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

Parameters

- *portId* - Register callback for a particular MAC device.
- *rxCallbackFn* - *IxEthAccRxCallbackFn* - Function to be called when Ethernet frames are availble.
- *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 : An argument other than portId is invalid.

A.13.0.50 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 behaviour 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.13.0.51 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 recieved 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.13.0.52 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

Notes:

- 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 sufficent number of buffers prior to enabling the NPE Ethernet receive path.

Parameters

- *portId* - Provide buffers only to specific Rx MAC.
- *buffer* - 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

Warning: This function will only check the parameters if the NDEBUG flag is not defined. Turning on the argument checking (disabled by default) will result in a lower EthAcc performance as this function is part of the data path.

A.13.0.53 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 : An argument other than portId is invalid.

A.13.0.54 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 behaviour while the port is enabled.

Tx FCS append is not disabled while Tx padding is enabled.

See also:

ixEthAccPortTxFrameAppendPaddingEnable

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.13.0.55 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 behaviour 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.13.0.56 ixEthAccPortTxFrameAppendPaddingDisable (IxEthAccPortId *portId*)

Disable the appending of padding bytes to the runt frames submitted to this port.

Disable the appending of padding bytes to runt frames submitted to this port. This is not the default behavior of the access component.

Warning: Do not change this behaviour 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.13.0.57 ixEthAccPortTxFrameAppendPaddingEnable (IxEthAccPortId *portId*)

Enable the appending of padding bytes to runt frames submitted to this port.

Enable the appending of up to 60 null-bytes to runt frames submitted to this port. This is the default behavior of the access component.

Warning: Do not change this behaviour while the port is enabled.

Note: When Tx padding is enabled, Tx FCS generation is turned on

See also:
ixEthAccPortTxFrameAppendFCSDusable

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.13.0.58 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.

When the frame is transmitted, the buffer shall be returned thru the callback *IxEthAccPortTxDoneCallback* .

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 when submitting faster than the network line rate. In case of over=Submitting, the order of the frames on the network may be modified.

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.
- *buffer* - 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.13.0.59 ixEthAccPortUnicastAddressShow (IxEthAccPortId *portId*)

Display unicast address has been configured using *ixEthAccUnicastMacAddressSet* .

Display unicast address has been configured using *ixEthAccUnicastMacAddressSet* . Display also the MAC filter used

Other functions modify the MAC filtering include:

- *ixEthAccPortMulticastAddressJoinAll()* - all multicast frames are forwarded to the application
- *ixEthAccPortMulticastAddressLeaveAll()* - rollback the effects of *ixEthAccPortMulticastAddressJoinAll()*
- *ixEthAccPortMulticastAddressLeave()* - unprovision a new filtering address
- *ixEthAccPortMulticastAddressJoin()* - provision a new filtering address

- *ixEthAccPortPromiscuousModeSet()* - all frames are forwarded to the application regardless of the multicast address provisioned
- *ixEthAccPortPromiscuousModeClear()* - frames are forwarded to the application following the multicast address provisioned

In all cases, unicast and broadcast addresses are forwarded to the application.

Reentrant - yes

ISR Callable - no

Parameters

portId - Ethernet port id.

Returns

void

A.13.0.60 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

- IX_ETH_ACC_SUCCESS
- IX_ETH_ACC_INVALID_PORT : portId is invalid.
- IX_ETH_ACC_MAC_UNINITIALIZED : port MAC address is not initialized.
- IX_ETH_ACC_FAIL : macAddr is invalid.

A.13.0.61 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.13.0.62 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

Precondition

Parameters

portId IxEthAccPortId

Returns

void

A.13.0.63 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.13.0.64 ixEthAccUnload (void)

Unload the Ethernet Access Service.

* - Reentrant - no

ISR Callable - no

Returns

void

A.14 IXP425 Ethernet Access Fast Path (IxEthAccFpathDep) API

IXP425 Ethernet Access Fast Path (IxEthAccFpathDep) API. IXP425 Ethernet Access Fast Path (IxEthAccFpathDep) APIIXP425 Ethernet Access Fast Path (IxEthAccFpathDep) APIIXP425 Ethernet Access Fast Path (IxEthAccFpathDep) APIThe dependencies common to the IXP425 Ethernet Access (IxEthAcc) API component and the IXP425 Fast Path Access (IxFpathAcc) API component.

A.14.0.1 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.14.0.2 Detailed Description

The dependencies common to the IXP425 Ethernet Access (IxEthAcc) API component and the IXP425 Fast Path Access (IxFpathAcc) API component.

A.14.0.3 Define Documentation

A.14.0.4 #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 behaviour 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.

Definition at line 86 of file IxEthAccFpathDep.h.

A.15 IXP425 Ethernet Database (IxEthDB) API

IXP425 Ethernet Database (IxEthDB) API. IXP425 Ethernet Database (IxEthDB) APIIXP425 Ethernet Database (IxEthDB) APIIXP425 Ethernet Database (IxEthDB) APIethDB is a library that does provides a MAC address database learning/filtering capability

A.15.0.1 Data Structures

- struct IxEthDBMacAddr
  The IEEE 802.3 Ethernet MAC address structure.

A.15.0.2 Defines

- #define **INLINE** __inline__
- #define **IX_ETH_DB_PRIVATE** 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.15.0.3 Typedefs

- typedef UINT32 IxEthDBPortId
  Definition of an IXP425 port.
- typedef UINT32 IxEthDBPortMap
  Port dependency map definition.

A.15.0.4 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.15.0.5 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 the port MAC address
- IxEthDBStatus ixEthDBFilteringStaticEntryProvision (IxEthDBPortId portID, IxEthDBMacAddr *macAddr)
  Populate the Ethernet learning/filtering database with a static MAC address.
- IxEthDBStatus ixEthDBFilteringDynamicEntryProvision (IxEthDBPortId portID, IxEthDBMacAddr *macAddr)
  Populate the Ethernet learning/filtering database with a dynamic MAC address.
- IxEthDBStatus ixEthDBFilteringEntryDelete (IxEthDBMacAddr *macAddr)
  Remove a MAC address entry from the Ethernet learning/filtering database.
- IxEthDBStatus ixEthDBFilteringPortSearch (IxEthDBPortId portID, IxEthDBMacAddr *macAddr)
  Search the Ethernet learning/filtering database for the given MAC address and port ID.
- IxEthDBStatus ixEthDBFilteringDatabaseSearch (IxEthDBPortId *portID, IxEthDBMacAddr *macAddr)
  Search the Ethernet learning/filtering database for a MAC address and return the port ID.

- IxEthDBStatus ixEthDBFilteringPortUpdatingSearch (IxEthDBPortId *portID, IxEthDBMacAddr *macAddr)
  Search the filtering database for a MAC address, return the port ID and reset the record age.
- IxEthDBStatus ixEthDBPortAgingDisable (IxEthDBPortId port)
  Disable the aging function for a specific port.
- IxEthDBStatus ixEthDBPortAgingEnable (IxEthDBPortId portID)
  Enable the aging function for a specific port.
- void ixEthDBDatabaseMaintenance (void)
  Performs a maintenance operation on the Ethernet learning/filtering database.
- IxEthDBStatus ixEthDBFilteringDatabaseShow (IxEthDBPortId portID)
  This function displays the Mac Ethernet MAC address filtering tables.
- void ixEthDBFilteringDatabaseShowAll (void)
  Displays the MAC address recorded in the filtering database for all registered ports (see IxEthDBPortDefs.h), grouped by port ID.

A.15.0.6 Detailed Description

ethDB is a library that does provides a MAC address database learning/filtering capability

A.15.0.7 Define Documentation

A.15.0.8 #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.

Definition at line 387 of file IxEthDB.h.

A.15.0.9 #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.

Definition at line 376 of file IxEthDB.h.

A.15.0.10 Typedef Documentation

A.15.0.11 typedef UINT32 IxEthDBPortId

Definition of an IXP425 port.

Definition at line 113 of file IxEthDB.h.

A.15.0.12 typedef UINT32 IxEthDBPortMap

Port dependency map definition.

Definition at line 120 of file IxEthDB.h.

A.15.0.13 Enumeration Type Documentation

A.15.0.14 enum IxEthDBStatus

Ethernet database status.

Enumeration Values

- ***IX_ETH_DB_SUCCESS*** Return success.
- ***IX_ETH_DB_FAIL*** Return fail.
- ***IX_ETH_DB_INVALID_PORT*** invalid port
- ***IX_ETH_DB_PORT_UNINITIALIZED*** port uninitialized
- ***IX_ETH_DB_MAC_UNINITIALIZED*** MAC uninitialized.
- ***IX_ETH_DB_INVALID_ARG*** invalid arg
- ***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.
  Definition at line 81 of file IxEthDB.h.

A.15.0.15 Function Documentation

A.15.0.16 void ixEthDBDatabaseMaintenance (void)

Performs a maintenance operation on the Ethernet learning/filtering database.

In order to perform a database maintenance this function must be called every seconds. It should be called regardless of whether learning is enabled or not.

Returns

void

- Reentrant - no
- ISR Callable - no

A.15.0.17 IxEthDBStatus ixEthDBFilteringDatabaseSearch (IxEthDBPortId * *portID*, IxEthDBMacAddr * *macAddr*)

Search the Ethernet learning/filtering database for a MAC address and return the port ID.

Searches the database for a MAC address. The function returns the portID for the MAC address record, if found. If no match is found the function returns IX_ETH_DB_NO_SUCH_ADDR. The portID is only valid if the function finds a match.

Reentrant - yes

ISR Callable - no

Parameters

- *portID* port ID the address belongs to (populated only on a successful search)
- *macAddr* MAC address to search for

Returns

- *IX_ETH_DB_SUCCESS* the record exists in the database
- *IX_ETH_DB_NO_SUCH_ADDR* the record was not found in the database

A.15.0.18 IxEthDBStatus ixEthDBFilteringDatabaseShow (IxEthDBPortId *portID*)

This function displays the Mac Ethernet MAC address filtering tables.

It displays the MAC address, port ID, entry type (dynamic/static),and age for the given port ID.

Reentrant - yes

ISR Callable - no

Parameters

portID port ID to display the MAC address entries

Returns

- *IX_ETH_DB_SUCCESS* operation completed successfully
- *IX_ETH_DB_INVALID_PORT* portID is invalid
- *IX_ETH_DB_PORT_UNINITIALIZED* port ID is not initialized

A.15.0.19 void ixEthDBFilteringDatabaseShowAll (void)

Displays the MAC address recorded in the filtering database for all registered ports (see IxEthDBPortDefs.h), grouped by port ID.

Returns

void

Reentrant - yes

ISR Callable - no

A.15.0.20 IxEthDBStatus ixEthDBFilteringDynamicEntryProvision (IxEthDBPortId *portID*, IxEthDBMacAddr * *macAddr*)

Populate the Ethernet learning/filtering database with a dynamic MAC address.

Populates the Ethernet learning/filtering database with a dynamic MAC address. This entry will be subject to normal aging function, if aging is enabled on its port. If there is an entry (static or dynamic) with the same MAC address on any port this entry will take precedence. Any other entry with the same MAC address will be removed.

Reentrant - yes

ISR Callable - no

Parameters

- *portID* port ID to add the dynamic address to
- *macAddr* static MAC address to add

Returns

- *IX_ETH_DB_SUCCESS* the add was successful
- *IX_ETH_DB_FAIL* failed to populate the database entry
- *IX_ETH_DB_INVALID_PORT* portID is invalid
- *IX_ETH_DB_PORT_UNINITIALIZED* port is not initialized

A.15.0.21 IxEthDBStatus ixEthDBFilteringEntryDelete (IxEthDBMacAddr * *macAddr*)

Remove a MAC address entry from the Ethernet learning/filtering database.

* - Reentrant - yes

ISR Callable - no

Parameters

macAddr MAC address to remove

Returns

- *IX_ETH_DB_SUCCESS* the removal was successful
- *IX_ETH_DB_NO_SUCH_ADDR* failed to remove the address (not in the database)

A.15.0.22 IxEthDBStatus ixEthDBFilteringPortSearch (IxEthDBPortId *portID*, IxEthDBMacAddr * *macAddr*)

Search the Ethernet learning/filtering database for the given MAC address and port ID.

This functions searches the database for a specific port ID and MAC address. Both the port ID and the MAC address have to match in order for the record to be reported as found.

Reentrant - yes

ISR Callable - no

Parameters

- *portID* port ID to search for

- *macAddr* MAC address to search for

Returns

- *IX_ETH_DB_SUCCESS* the record exists in the database
- *IX_ETH_DB_NO_SUCH_ADDR* the record was not found in the database
- *IX_ETH_DB_INVALID_PORT* portID is invalid
- *IX_ETH_DB_PORT_UNINITIALIZED* port ID is not initialized

A.15.0.23 IxEthDBStatus ixEthDBFilteringPortUpdatingSearch (IxEthDBPortId * *portID*, IxEthDBMacAddr * *macAddr*)

Search the filtering database for a MAC address, return the port ID and reset the record age.

Searches the database for a MAC address. The function returns the portID for the MAC address record and resets the entry age to 0, if found. If no match is found the function returns IX_ETH_DB_NO_SUCH_ADDR. The portID is only valid if the function finds a match.

Reentrant - yes

ISR Callable - no

Returns

- *IX_ETH_DB_SUCCESS* the MAC address was found
- *IX_ETH_DB_NO_SUCH_ADDR* the MAC address was not found

A.15.0.24 IxEthDBStatus ixEthDBFilteringStaticEntryProvision (IxEthDBPortId *portID*, IxEthDBMacAddr * *macAddr*)

Populate the Ethernet learning/filtering database with a static MAC address.

Populates the Ethernet learning/filtering database with a static MAC address. The entry will not be subject to aging. If there is an entry (static or dynamic) with the corresponding MAC address on any port this entry will take precedence. Any other entry with the same MAC address will be removed.

Reentrant - yes

ISR Callable - no

Parameters

- *portID* port ID to add the static address to
- *macAddr* static MAC address to add

Returns

- *IX_ETH_DB_SUCCESS* the add was successful
- *IX_ETH_DB_FAIL* failed to populate the database entry
- *IX_ETH_DB_INVALID_PORT* portID is invalid
- *IX_ETH_DB_PORT_UNINITIALIZED* port is not initialized

A.15.0.25 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.15.0.26 IxEthDBStatus ixEthDBPortAddressSet (IxEthDBPortId *portID*, IxEthDBMacAddr * *macAddr*)

set the 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.

Parameters

- *portID* ID of the port whose MAC address is set
- *macAddr* port MAC address

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.15.0.27 IxEthDBStatus ixEthDBPortAgingDisable (IxEthDBPortId *port*)

Disable the aging function for a specific port.

* - Reentrant - yes

ISR Callable - no

Parameters

portID port ID to disable aging on

Returns

- *IX_ETH_DB_SUCCESS* aging disabled successfully
- *IX_ETH_DB_INVALID_PORT* portID is invalid
- *IX_ETH_DB_PORT_UNINITIALIZED* port ID is not initialized

A.15.0.28 IxEthDBStatus ixEthDBPortAgingEnable (IxEthDBPortId *portID*)

Enable the aging function for a specific port.

Enables the aging of dynamic MAC address entries stored in the learning/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 port ID to enable aging on

Returns

- *IX_ETH_DB_SUCCESS* aging enabled successfully
- *IX_ETH_DB_INVALID_PORT* portID is invalid
- *IX_ETH_DB_PORT_UNINITIALIZED* port ID is not initialized

A.15.0.29 IxEthDBStatus ixEthDBPortDisable (IxEthDBPortId *portID*)

disable processing on a port

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).

Note: After Ethernet NPEs are disabled they are stopped therefore when re-enabled they need to be reset, downloaded with microcode and started. For learning to restart working the user needs to call again ixEthAccPortUnicastMacAddressSet or ixEthDBUnicastAddressSet with the respective port MAC address. Residual MAC addresses learnt before the port was disable will take up to IX_ETH_DB_MAINTENANCE_TIME to be downloaded in the NPE MAC tree after learning is restarted on the port. The dynamic addresses do not dissapear after the disable-enable sequence. They also do not age during the time the port is disabled.

Parameters

portID ID of the port to disable processing on

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.15.0.30 IxEthDBStatus ixEthDBPortEnable (IxEthDBPortId *portID*)

enable a port

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).

Parameters

portID ID of the port to enable processing on

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.15.0.31 void ixEthDBPortInit (IxEthDBPortId *portID*)

Initializes a port.

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).

Parameters

portID ID of the port to be initialized

Returns

void

See also:

IxEthDBPortDefs.h for port definitions

A.16 IXP425 Ethernet Database Port Definitions (IxEthDBPortDefs)

IXP425 Ethernet Database Port Definitions (IxEthDBPortDefs)IXP425 Ethernet Database Port Definitions (IxEthDBPortDefs)IXP425 Ethernet Database Port Definitions (IxEthDBPortDefs)IXP425 Ethernet Database Port Definitions (IxEthDBPortDefs)IXP425 Ethernet Port Definitions for private MAC learning API.

A.16.0.1 Data Structures

- struct IxEthDBPortDefinition
  Port Definition - a structure contains the Port type and capabilities.

A.16.0.2 Defines

- #define IX_ETH_DB_NUMBER_OF_PORTS (sizeof (ixEthDBPortDefinitions) / sizeof (ixEthDBPortDefinitions[0]))
  number of supported ports
- #define IX_ETH_DB_PORTS_ASSERTION { switch(0) { case 0 : ; case 1 : ; case IX_ETH_DB_NUMBER_OF_PORTS : ; }}
  catch invalid port definitions (<2) with a compile-time assertion resulting in a duplicate case error.
- #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.16.0.3 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.16.0.4 Detailed Description

IXP425 Ethernet Port Definitions for private MAC learning API.

A.16.0.5 Define Documentation

A.16.0.6 #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

Definition at line 113 of file IxEthDBPortDefs.h.

A.16.0.7 #define IX_ETH_DB_CHECK_PORT(portID)

Value

```
{ \
    if ((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

Definition at line 119 of file IxEthDBPortDefs.h.

A.16.0.8 Enumeration Type Documentation

A.16.0.9 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).

Enumeration Values

- ***NO_CAPABILITIES*** no aging capabilities
- ***ENTRY_AGING*** aging capabilities present
  Definition at line 65 of file IxEthDBPortDefs.h.

A.16.0.10 enum IxEthDBPortType

Port types - currently only Ethernet NPEs are recognized as specific types.

Enumeration Values

- ***ETH_GENERIC*** generic ethernet port - not supported
- ***ETH_NPE*** specific Ethernet NPE
  Definition at line 53 of file IxEthDBPortDefs.h.

A.17 IXP425 Ethernet PHY Access (IxEthMii) API

IXP425 Ethernet Phy Access (IxEthMii) API. IXP425 Ethernet Phy Access (IxEthMii) APIIXP425 Ethernet Phy Access (IxEthMii) APIIXP425 Ethernet Phy Access (IxEthMii) APIethMii is a library that does provides access to the Ethernet PHYs

A.17.0.1 Functions

- IX_STATUS ixEthMiiPhyScan (BOOL phyPresent[], UINT32 maxPhyCount)
  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.

- IX_STATUS ixEthMiiPhyConfig (UINT32 phyAddr, BOOL speed100, BOOL fullDuplex, BOOL autonegotiate)

  Configure a PHY Configure a PHY's speed, duplex and autonegotiation status.
- IX_STATUS ixEthMiiPhyLoopbackEnable (UINT32 phyAddr)

  Enable PHY Loopback in a specific Eth MII port.
- IX_STATUS ixEthMiiPhyLoopbackDisable (UINT32 phyAddr)

  Disable PHY Loopback in a specific Eth MII port.
- IX_STATUS ixEthMiiPhyReset (UINT32 phyAddr)

  Reset a PHY Reset a PHY.
- IX_STATUS ixEthMiiLinkStatus (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.
- IX_STATUS ixEthMiiPhyShow (UINT32 phyAddr)

  Display information on a specified PHY Display link status, speed, duplex and Auto Negotiation status.

A.17.0.2 Detailed Description

ethMii is a library that does provides access to the Ethernet PHYs

A.17.0.3 Function Documentation

A.17.0.4 ixEthMiiLinkStatus (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 initialised, 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

IX_STATUS

- IX_SUCCESS
- IX_FAIL : invalid arguments.

A.17.0.5 ixEthMiiPhyConfig (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 initialised, 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

IX_STATUS

- IX_SUCCESS
- IX_FAIL : invalid arguments.

A.17.0.6 ixEthMiiPhyLoopbackDisable (UINT32 *phyAddr*)

Disable PHY Loopback in a specific Eth MII port.

* - Reentrant - no

ISR Callable - no

Parameters

phyAddr[in] - the address of the Ethernet PHY (0-31)

Returns

IX_STATUS

- IX_SUCCESS
- IX_FAIL : invalid arguments.

A.17.0.7 ixEthMiiPhyLoopbackEnable (UINT32 *phyAddr*)

Enable PHY Loopback in a specific Eth MII port.

* - Reentrant - no

ISR Callable - no

Parameters

phyAddr[in] - the address of the Ethernet PHY (0-31)

Returns

IX_STATUS

- IX_SUCCESS
- IX_FAIL : invalid arguments.

A.17.0.8 ixEthMiiPhyReset (UINT32 *phyAddr*)

Reset a PHY Reset a PHY.

* - Reentrant - no

ISR Callable - no

Precondition

The MAC on Ethernet Port 2 (NPE C) must be initialised, and generating the MDIO clock.

Parameters

phyAddr: the address of the Ethernet PHY (0-31)

Returns

IX_STATUS

- IX_SUCCESS
- IX_FAIL : invalid arguments.

A.17.0.9 ixEthMiiPhyScan (BOOL *phyPresent*[], UINT32 *maxPhyCount*)

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 initialised, and generating the MDIO clock.

Parameters

- *phyPresent* : boolean array of IXP425_ETH_ACC_MII_MAX_ADDR entries
- *maxPhyCount* : number of PHYs to search for (the scan will stop when the indicated number of PHYs is found).

Returns

IX_STATUS

- IX_ETH_ACC_SUCCESS
- IX_ETH_ACC_FAIL : invalid arguments.

A.17.0.10 ixEthMiiPhyShow (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 initialised, and generating the MDIO clock.

Parameters

phyAddr: the address of the Ethernet PHY (0-31)

Returns

IX_STATUS

- IX_SUCCESS
- IX_FAIL : invalid arguments.

A.18 IXP425 Ethernet NPE (IxEthNpe) API

IXP425 Ethernet NPE (IxEthNpe) API. IXP425 Ethernet NPE (IxEthNpe) APIIXP425 Ethernet NPE (IxEthNpe) APIIXP425 Ethernet NPE (IxEthNpe) APIContains the API for Ethernet NPE.

A.18.0.1 Defines

- #define IX_ETHNPE_X2P_NPE_HALT 0x00

  Request from the XScale client for the NPE to immediately halt all execution and flush any mbufs in its possession.
- #define IX_ETHNPE_X2P_NPE_PORT_DISABLE 0x40

  Request from the XScale client for the NPE to immediately flush any mbufs in its possession.
- #define IX_ETHNPE_X2P_ELT_SETPORTADDRESS 0x10

Indication from the XScale client that the attached Ethernet port's MAC address is equal to the speci-fied 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 XScale 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 XScale 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 XScale 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 XScale client that the universal FastPath Ethernet type should be set to the specified value.

- #define IX_ETHNPE_X2P_FP_WRITETEMPMDF 0x21

  Indication from the XScale 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 XScale 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 XScale 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 XScale 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 IX_ETHNPE_P2X_NPE_PORT_DISABLE 0x40
  Response to a IX_ETHNPE_X2P_NPE_PORT_DISABLE.
- #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
  QMgr Queue Fast Path shutdown field left boundary.
- #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 0x80000000
  Macro to mask the NPE ID field of the RxEnet Queue Manager Entry.
- #define IX_ETHNPE_QM_Q_RXENET_PORTID_MASK 0x70000000
  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 0x80000000
  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)
  Extraction macro for Fast Path Bit field of TxEnet Queue Manager Entry.
- #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.18.0.2 Detailed Description

Contains the API for Ethernet NPE.

A.18.0.3 Define Documentation

A.18.0.4 #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.

Definition at line 186 of file IxEthNpe.h.

A.18.0.5 #define IX_ETHNPE_P2X_ELT_BALANCEREQUEST 0x12

Request from the NPE for the XScale 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).

Definition at line 199 of file IxEthNpe.h.

A.18.0.6 #define IX_ETHNPE_P2X_NPE_PORT_DISABLE 0x40

Response to a IX_ETHNPE_X2P_NPE_PORT_DISABLE.

Indication from the NPE that mbufs held are released (msg word1 is null) or there is still pending traffic (msg word1 is not null)

Definition at line 267 of file IxEthNpe.h.

A.18.0.7 #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.

Definition at line 394 of file IxEthNpe.h.

A.18.0.8 #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

Definition at line 522 of file IxEthNpe.h.

A.18.0.9 #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.

Definition at line 507 of file IxEthNpe.h.

A.18.0.10 #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

Definition at line 653 of file IxEthNpe.h.

A.18.0.11 #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.

Definition at line 427 of file IxEthNpe.h.

A.18.0.12 #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

Definition at line 558 of file IxEthNpe.h.

A.18.0.13 #define IX_ETHNPE_QM_Q_RXENET_ENTRY(addr, id, prt)

Value

```
((id)  << 30 | \
                                                    (prt) << 28 | (addr))
```

Queue entry construction macros for RxEnet Queue Manager.

Definition at line 675 of file IxEthNpe.h.

A.18.0.14 #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.

Definition at line 533 of file IxEthNpe.h.

A.18.0.15 #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 XScale client) to any of the physical ports. 6: It is reserved 7: Indication that the NPE did not find the associated frame's destination MAC address within its internal filtering database.

Definition at line 547 of file IxEthNpe.h.

A.18.0.16 #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.

Definition at line 455 of file IxEthNpe.h.

A.18.0.17 #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

Definition at line 599 of file IxEthNpe.h.

A.18.0.18 #define IX_ETHNPE_QM_Q_TXENET_ENTRY(addr, fp, pri)

Value

```
((fp)  << 30 | \
                                                    (pri) << 28 | (addr))
```

Queue entry construction macros for TxEnet Queue Manager.

Definition at line 683 of file IxEthNpe.h.

A.18.0.19 #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.

Definition at line 436 of file IxEthNpe.h.

A.18.0.20 #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)
```

Extraction macro for Fast Path Bit field of TxEnet Queue Manager Entry.Macro to set to use either SlowPath/FastPath TxEnet Queue Manager Entry

"1": if the mbuf address points to a FastPath descriptor and "0": if the mbuf address points to a SlowPath mbuf descriptor.

This bit is always cleared upon return from the Ethernet NPE (to either the TxEnetDone or FreeFP queue).

Definition at line 574 of file IxEthNpe.h.

A.18.0.21 #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.

Definition at line 465 of file IxEthNpe.h.

A.18.0.22 #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

Definition at line 610 of file IxEthNpe.h.

A.18.0.23 #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.

Definition at line 475 of file IxEthNpe.h.

A.18.0.24 #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.

Definition at line 620 of file IxEthNpe.h.

A.18.0.25 #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.

Definition at line 445 of file IxEthNpe.h.

A.18.0.26 #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).

Definition at line 587 of file IxEthNpe.h.

A.18.0.27 #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.

Definition at line 497 of file IxEthNpe.h.

A.18.0.28 #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

Definition at line 643 of file IxEthNpe.h.

A.18.0.29 #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 en-tries originating from the Eth1 NPE.

Definition at line 632 of file IxEthNpe.h.

A.18.0.30 #define IX_ETHNPE_X2P_ELT_ACCESSRELEASE 0x12

Indication from the XScale 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.

Definition at line 109 of file IxEthNpe.h.

A.18.0.31 #define IX_ETHNPE_X2P_FP_SETETHERNETTYPE 0x20

Indication from the XScale 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.

Definition at line 129 of file IxEthNpe.h.

A.18.0.32 #define IX_ETHNPE_X2P_NPE_HALT 0x00

Request from the XScale 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.

Definition at line 64 of file IxEthNpe.h.

A.18.0.33 #define IX_ETHNPE_X2P_NPE_PORT_DISABLE 0x40

Request from the XScale client for the NPE to immediately 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.

Definition at line 78 of file IxEthNpe.h.

A.18.0.34 #define IX_ETHNPE_X2P_STATS_RESET 0x31

Request from the XScale 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.

Definition at line 158 of file IxEthNpe.h.

A.19 IXP425 Feature Control (IxFeatureCtrl) API

IXP425 Feature Control (IxFeatureCtrl) API. IXP425 Feature Control (IxFeatureCtrl) APIIXP425 Feature Control (IxFeatureCtrl) APIIXP425 Feature Control (IxFeatureCtrl) APIThe Public API for the IXP425 Feature Control.

A.19.0.1 Modules

- Software Configuration for Access Component
  This section describes software configuration in access component. The configuration can be changed at run-time. ixFeatureCtrlSwConfigurationCheck( ) will be used across applicable access component to check the configuration. ixFeatureCtrlSwConfigurationWrite( ) is used to write the software configuration.

A.19.0.2 Defines

- #define IX_FEATURE_CTRL_COMPONENT_DISABLED 0
  Hardware Component is disabled/unavailable. Return status by ixFeatureCtrlComponentCheck().
- #define IX_FEATURE_CTRL_COMPONENT_ENABLED 1
  Hardware Component is available. Return status by ixFeatureCtrlComponentCheck().
- #define IX_FEATURE_CTRL_SILICON_TYPE_A0 0
  This is the value of A0 Silicon in product ID.
- #define IX_FEATURE_CTRL_SILICON_TYPE_B0 1
  This is the value of B0 Silicon in product ID.
- #define IX_FEATURE_CTRL_SILICON_STEPPING_MASK 0xF
  This is the mask of silicon stepping in product ID.
- #define IX_FEATURE_CTRL_XSCALE_FREQ_533 ((0x1C)<<4)
  This is the value of 533MHz XScale Core in product ID.
- #define IX_FEATURE_CTRL_XSCALE_FREQ_400 ((0x1D)<<4)
  This is the value of 400MHz XScale Core in product ID.
- #define IX_FEATURE_CTRL_XSCALE_FREQ_266 ((0x1F)<<4)
  This is the value of 266MHz XScale Core in product ID.

- #define IX_FEATURE_CTRL_XSCALE_FREQ_MASK ((0xFF)<<4)
  This is the mask of XScale Core in product ID.
- #define IX_FEATURECTRL_REG_LOC_RCOMP 0
  The bit location for RComp Circuitry.
- #define IX_FEATURECTRL_REG_LOC_USB 1
  The bit location for USB Controller.
- #define IX_FEATURECTRL_REG_LOC_HASH 2
  The bit location for Hashing Coprocessor.
- #define IX_FEATURECTRL_REG_LOC_AES 3
  The bit location for AES Coprocessor.
- #define IX_FEATURECTRL_REG_LOC_DES 4
  The bit location for DES Coprocessor.
- #define IX_FEATURECTRL_REG_LOC_HDLC 5
  The bit location for HDLC Coprocessor.
- #define IX_FEATURECTRL_REG_LOC_AAL 6
  The bit location for AAL Coprocessor.
- #define IX_FEATURECTRL_REG_LOC_HSS 7
  The bit location for HSS Coprocessor.
- #define IX_FEATURECTRL_REG_LOC_UTOPIA 8
  The bit location for UTOPIA Coprocessor.
- #define IX_FEATURECTRL_REG_LOC_ETH0 9
  The bit location for Ethernet 0 Coprocessor.
- #define IX_FEATURECTRL_REG_LOC_ETH1 10
  The bit location for Ethernet 1 Coprocessor.
- #define IX_FEATURECTRL_REG_LOC_NPEA 11
  The bit location for NPE A.
- #define IX_FEATURECTRL_REG_LOC_NPEB 12
  The bit location for NPE B.
- #define IX_FEATURECTRL_REG_LOC_NPEC 13
  The bit location for NPE C.
- #define IX_FEATURECTRL_REG_LOC_PCI 14
  The bit location for PCI Controller.
- #define IX_FEATURECTRL_REG_LOC_UTOPIA_PHY_LIMIT 16
  The bit location for Utopia PHY Limit Status.
- #define IX_FEATURECTRL_RCOMP (1<<IX_FEATURECTRL_REG_LOC_RCOMP)
  The Component Name for RCOMP Circuitry. The name will be used for ixFeatureCtrlComponentCheck().
- #define IX_FEATURECTRL_USB (1<<IX_FEATURECTRL_REG_LOC_USB)
  The Component Name for USB Controller. The name will be used for ixFeatureCtrlComponentCheck().

- #define IX_FEATURECTRL_HASH (1<<IX_FEATURECTRL_REG_LOC_HASH)
  The Component Name for Hashing Coprocessor. The name will be used for ixFeatureCtrlComponentCheck().
- #define IX_FEATURECTRL_AES (1<<IX_FEATURECTRL_REG_LOC_AES)
  The Component Name for AES Coprocessor. The name will be used for ixFeatureCtrlComponentCheck().
- #define IX_FEATURECTRL_DES (1<<IX_FEATURECTRL_REG_LOC_DES)
  The Component Name for DES Coprocessor. The name will be used for ixFeatureCtrlComponentCheck().
- #define IX_FEATURECTRL_HDLC (1<<IX_FEATURECTRL_REG_LOC_HDLC)
  The Component Name for HDLC Coprocessor. The name will be used for ixFeatureCtrlComponentCheck().
- #define IX_FEATURECTRL_AAL (1<<IX_FEATURECTRL_REG_LOC_AAL)
  The Component Name for AAL Coprocessor. The name will be used for ixFeatureCtrlComponentCheck().
- #define IX_FEATURECTRL_HSS (1<<IX_FEATURECTRL_REG_LOC_HSS)
  The Component Name for HSS Coprocessor. The name will be used for ixFeatureCtrlComponentCheck().
- #define IX_FEATURECTRL_UTOPIA (1<<IX_FEATURECTRL_REG_LOC_UTOPIA)
  The Component Name for Utopia Coprocessor. The name will be used for ixFeatureCtrlComponentCheck().
- #define IX_FEATURECTRL_ETH0 (1<<IX_FEATURECTRL_REG_LOC_ETH0)
  The Component Name for Ethernet 0 Coprocessor. The name will be used for ixFeatureCtrlComponentCheck().
- #define IX_FEATURECTRL_ETH1 (1<<IX_FEATURECTRL_REG_LOC_ETH1)
  The Component Name for Ethernet 1 Coprocessor. The name will be used for ixFeatureCtrlComponentCheck().
- #define IX_FEATURECTRL_NPEA (1<<IX_FEATURECTRL_REG_LOC_NPEA)
  The Component Name for NPE A. The name will be used for ixFeatureCtrlComponentCheck().
- #define IX_FEATURECTRL_NPEB (1<<IX_FEATURECTRL_REG_LOC_NPEB)
  The Component Name for NPE B. The name will be used for ixFeatureCtrlComponentCheck().
- #define IX_FEATURECTRL_NPEC (1<<IX_FEATURECTRL_REG_LOC_NPEC)
  The Component Name for NPE C. The name will be used for ixFeatureCtrlComponentCheck().
- #define IX_FEATURECTRL_PCI (1<<IX_FEATURECTRL_REG_LOC_PCI)
  The Component Name for PCI Controller. The name will be used for ixFeatureCtrlComponentCheck().
- #define IX_FEATURECTRL_REG_UTOPIA_32PHY 0x0
  Maximum UTOPIA PHY available to IXP425 is 32.
- #define IX_FEATURECTRL_REG_UTOPIA_16PHY 0x1
  Maximum UTOPIA PHY available to IXP425 is 16.
- #define IX_FEATURECTRL_REG_UTOPIA_8PHY 0x2
  Maximum UTOPIA PHY available to IXP425 is 8.

- #define IX_FEATURECTRL_REG_UTOPIA_4PHY 0x3
  Maximum UTOPIA PHY available to IXP425 is 4.

A.19.0.3 Typedefs

- typedef UINT32 IxFeatureCtrlReg
  Feature Control Register that contains hardware components' availability information.
- typedef UINT32 IxFeatureCtrlProductId
  Product ID of Silicon that contains Silicon Stepping and Maximum XScale Core Frequency information.
- typedef UINT32 IxFeatureCtrlComponentType
  The component type used for ixFeatureCtrlComponentCheck().

A.19.0.4 Functions

- IxFeatureCtrlReg ixFeatureCtrlRead (void)
  This function reads out the CURRENT value of Feature Control Register. The current value may not be the same as that of the hardware component availability.
- IxFeatureCtrlReg ixFeatureCtrlHwCapabilityRead (void)
  This function reads out the hardware capability of a silicon type as defined in feature control register.This value is different from that returned by ixFeatureCtrlRead() because this function returns the actual hardware component availability.
- void ixFeatureCtrlWrite (IxFeatureCtrlReg expUnitReg)
  This function write the value stored in IxFeatureCtrlReg expUnitReg to the Feature Control Register.
- IX_STATUS ixFeatureCtrlComponentCheck (IxFeatureCtrlComponentType componentType)
  This function will check the availability of hardware component specified as componentType value.
- IxFeatureCtrlProductId ixFeatureCtrlProductIdRead (void)
  This function will return IXP425 product ID i.e. CP15, Register 0.
- IX_STATUS ixFeatureCtrlSwConfigurationCheck (IxFeatureCtrlSwConfig swConfigType)
  This function checks whether the specified software configuration is enabled or disabled.
- void ixFeatureCtrlSwConfigurationWrite (IxFeatureCtrlSwConfig swConfigType, BOOL enabled)
  This function enable/disable the specified software configuration.

A.19.0.5 Detailed Description

The Public API for the IXP425 Feature Control.

A.19.0.6 Define Documentation

A.19.0.7 #define IX_FEATURE_CTRL_COMPONENT_DISABLED 0

Hardware Component is disabled/unavailable. Return status by ixFeatureCtrlComponentCheck().

Definition at line 78 of file IxFeatureCtrl.h.

A.19.0.8 #define IX_FEATURE_CTRL_COMPONENT_ENABLED 1

Hardware Component is available. Return status by ixFeatureCtrlComponentCheck().

Definition at line 88 of file IxFeatureCtrl.h.

A.19.0.9 #define IX_FEATURE_CTRL_SILICON_STEPPING_MASK 0xF

This is the mask of silicon stepping in product ID.

Definition at line 142 of file IxFeatureCtrl.h.

A.19.0.10 #define IX_FEATURE_CTRL_SILICON_TYPE_A0 0

This is the value of A0 Silicon in product ID.

Definition at line 124 of file IxFeatureCtrl.h.

A.19.0.11 #define IX_FEATURE_CTRL_SILICON_TYPE_B0 1

This is the value of B0 Silicon in product ID.

Definition at line 133 of file IxFeatureCtrl.h.

A.19.0.12 #define IX_FEATURE_CTRL_XSCALE_FREQ_266 ((0x1F)<<4)

This is the value of 266MHz XScale Core in product ID.

Definition at line 169 of file IxFeatureCtrl.h.

A.19.0.13 #define IX_FEATURE_CTRL_XSCALE_FREQ_400 ((0x1D)<<4)

This is the value of 400MHz XScale Core in product ID.

Definition at line 160 of file IxFeatureCtrl.h.

A.19.0.14 #define IX_FEATURE_CTRL_XSCALE_FREQ_533 ((0x1C)<<4)

This is the value of 533MHz XScale Core in product ID.

Definition at line 151 of file IxFeatureCtrl.h.

A.19.0.15 #define IX_FEATURE_CTRL_XSCALE_FREQ_MASK ((0xFF)<<4)

This is the mask of XScale Core in product ID.

Definition at line 178 of file IxFeatureCtrl.h.

A.19.0.16 #define IX_FEATURECTRL_AAL (1<<IX_FEATURECTRL_REG_LOC_AAL)

The Component Name for AAL Coprocessor. The name will be used for ixFeatureCtrlComponentCheck().

Definition at line 437 of file IxFeatureCtrl.h.

A.19.0.17 #define IX_FEATURECTRL_AES (1<<IX_FEATURECTRL_REG_LOC_AES)

The Component Name for AES Coprocessor. The name will be used for ixFeatureCtrlComponentCheck().

Definition at line 407 of file IxFeatureCtrl.h.

A.19.0.18 #define IX_FEATURECTRL_DES (1<<IX_FEATURECTRL_REG_LOC_DES)

The Component Name for DES Coprocessor. The name will be used for ixFeatureCtrlComponentCheck().

Definition at line 417 of file IxFeatureCtrl.h.

A.19.0.19 #define IX_FEATURECTRL_ETH0 (1<<IX_FEATURECTRL_REG_LOC_ETH0)

The Component Name for Ethernet 0 Coprocessor. The name will be used for ixFeatureCtrlComponentCheck().

Definition at line 467 of file IxFeatureCtrl.h.

A.19.0.20 #define IX_FEATURECTRL_ETH1 (1<<IX_FEATURECTRL_REG_LOC_ETH1)

The Component Name for Ethernet 1 Coprocessor. The name will be used for ixFeatureCtrlComponentCheck().

Definition at line 477 of file IxFeatureCtrl.h.

A.19.0.21 #define IX_FEATURECTRL_HASH (1<<IX_FEATURECTRL_REG_LOC_HASH)

The Component Name for Hashing Coprocessor. The name will be used for ixFeatureCtrlComponentCheck().

Definition at line 397 of file IxFeatureCtrl.h.

A.19.0.22 #define IX_FEATURECTRL_HDLC (1<<IX_FEATURECTRL_REG_LOC_HDLC)

The Component Name for HDLC Coprocessor. The name will be used for ixFeatureCtrlComponentCheck().

Definition at line 427 of file IxFeatureCtrl.h.

A.19.0.23 #define IX_FEATURECTRL_HSS (1<<IX_FEATURECTRL_REG_LOC_HSS)

The Component Name for HSS Coprocessor. The name will be used for ixFeatureCtrlComponentCheck().

Definition at line 447 of file IxFeatureCtrl.h.

A.19.0.24 #define IX_FEATURECTRL_NPEA (1<<IX_FEATURECTRL_REG_LOC_NPEA)

The Component Name for NPE A. The name will be used for ixFeatureCtrlComponentCheck().

Definition at line 487 of file IxFeatureCtrl.h.

A.19.0.25 #define IX_FEATURECTRL_NPEB (1<<IX_FEATURECTRL_REG_LOC_NPEB)

The Component Name for NPE B. The name will be used for ixFeatureCtrlComponentCheck().

Definition at line 497 of file IxFeatureCtrl.h.

A.19.0.26 #define IX_FEATURECTRL_NPEC (1<<IX_FEATURECTRL_REG_LOC_NPEC)

The Component Name for NPE C. The name will be used for ixFeatureCtrlComponentCheck().

Definition at line 507 of file IxFeatureCtrl.h.

A.19.0.27 #define IX_FEATURECTRL_PCI (1<<IX_FEATURECTRL_REG_LOC_PCI)

The Component Name for PCI Controller. The name will be used for ixFeatureCtrlComponentCheck().

Definition at line 517 of file IxFeatureCtrl.h.

A.19.0.28 #define IX_FEATURECTRL_RCOMP (1<<IX_FEATURECTRL_REG_LOC_RCOMP)

The Component Name for RCOMP Circuitry. The name will be used for ixFeatureCtrlComponentCheck().

Definition at line 377 of file IxFeatureCtrl.h.

A.19.0.29 #define IX_FEATURECTRL_REG_LOC_AAL 6

The bit location for AAL Coprocessor.

Definition at line 281 of file IxFeatureCtrl.h.

A.19.0.30 #define IX_FEATURECTRL_REG_LOC_AES 3

The bit location for AES Coprocessor.

Definition at line 254 of file IxFeatureCtrl.h.

A.19.0.31 #define IX_FEATURECTRL_REG_LOC_DES 4

The bit location for DES Coprocessor.

Definition at line 263 of file IxFeatureCtrl.h.

A.19.0.32 #define IX_FEATURECTRL_REG_LOC_ETH0 9

The bit location for Ethernet 0 Coprocessor.

Definition at line 308 of file IxFeatureCtrl.h.

A.19.0.33 #define IX_FEATURECTRL_REG_LOC_ETH1 10

The bit location for Ethernet 1 Coprocessor.

Definition at line 317 of file IxFeatureCtrl.h.

A.19.0.34 #define IX_FEATURECTRL_REG_LOC_HASH 2

The bit location for Hashing Coprocessor.

Definition at line 245 of file IxFeatureCtrl.h.

A.19.0.35 #define IX_FEATURECTRL_REG_LOC_HDLC 5

The bit location for HDLC Coprocessor.

Definition at line 272 of file IxFeatureCtrl.h.

A.19.0.36 #define IX_FEATURECTRL_REG_LOC_HSS 7

The bit location for HSS Coprocessor.

Definition at line 290 of file IxFeatureCtrl.h.

A.19.0.37 #define IX_FEATURECTRL_REG_LOC_NPEA 11

The bit location for NPE A.

Definition at line 326 of file IxFeatureCtrl.h.

A.19.0.38 #define IX_FEATURECTRL_REG_LOC_NPEB 12

The bit location for NPE B.

Definition at line 335 of file IxFeatureCtrl.h.

A.19.0.39 #define IX_FEATURECTRL_REG_LOC_NPEC 13

The bit location for NPE C.

Definition at line 344 of file IxFeatureCtrl.h.

A.19.0.40 #define IX_FEATURECTRL_REG_LOC_PCI 14

The bit location for PCI Controller.

Definition at line 353 of file IxFeatureCtrl.h.

A.19.0.41 #define IX_FEATURECTRL_REG_LOC_RCOMP 0

The bit location for RComp Circuitry.

Definition at line 227 of file IxFeatureCtrl.h.

A.19.0.42 #define IX_FEATURECTRL_REG_LOC_USB 1

The bit location for USB Controller.

Definition at line 236 of file IxFeatureCtrl.h.

A.19.0.43 #define IX_FEATURECTRL_REG_LOC_UTOPIA 8

The bit location for UTOPIA Coprocessor.

Definition at line 299 of file IxFeatureCtrl.h.

A.19.0.44 #define IX_FEATURECTRL_REG_LOC_UTOPIA_PHY_LIMIT 16

The bit location for Utopia PHY Limit Status.

Definition at line 362 of file IxFeatureCtrl.h.

A.19.0.45 #define IX_FEATURECTRL_REG_UTOPIA_16PHY 0x1

Maximum UTOPIA PHY available to IXP425 is 16.

Definition at line 537 of file IxFeatureCtrl.h.

A.19.0.46 #define IX_FEATURECTRL_REG_UTOPIA_32PHY 0x0

Maximum UTOPIA PHY available to IXP425 is 32.

Definition at line 527 of file IxFeatureCtrl.h.

A.19.0.47 #define IX_FEATURECTRL_REG_UTOPIA_4PHY 0x3

Maximum UTOPIA PHY available to IXP425 is 4.

Definition at line 557 of file IxFeatureCtrl.h.

A.19.0.48 #define IX_FEATURECTRL_REG_UTOPIA_8PHY 0x2

Maximum UTOPIA PHY available to IXP425 is 8.

Definition at line 547 of file IxFeatureCtrl.h.

A.19.0.49 #define IX_FEATURECTRL_USB (1<<IX_FEATURECTRL_REG_LOC_USB)

The Component Name for USB Controller. The name will be used for ixFeatureCtrlComponentCheck().

Definition at line 387 of file IxFeatureCtrl.h.

A.19.0.50 #define IX_FEATURECTRL_UTOPIA (1<<IX_FEATURECTRL_REG_LOC_UTOPIA)

The Component Name for Utopia Coprocessor. The name will be used for ixFeatureCtrlComponentCheck().

Definition at line 457 of file IxFeatureCtrl.h.

A.19.0.51 Typedef Documentation

A.19.0.52 IxFeatureCtrlComponentType

The component type used for ixFeatureCtrlComponentCheck().

Definition at line 647 of file IxFeatureCtrl.h.

A.19.0.53 IxFeatureCtrlProductId

Product ID of Silicon that contains Silicon Stepping and Maximum XScale Core Frequency information.

Definition at line 637 of file IxFeatureCtrl.h.

A.19.0.54 IxFeatureCtrlReg

Feature Control Register that contains hardware components' availability information.

Definition at line 627 of file IxFeatureCtrl.h.

A.19.0.55 Function Documentation

A.19.0.56 IX_STATUS ixFeatureCtrlComponentCheck (IxFeatureCtrlComponentType *componentType*)

This function will check the availability of hardware component specified as componentType value.

Usage Example:

- if(IX_FEATURE_CTRL_COMPONENT_DISABLED != ixFeatureCtrlComponentCheck(IX_FEATURECTRL_ETH0))
- if(IX_FEATURE_CTRL_COMPONENT_ENABLED == ixFeatureCtrlComponentCheck(IX_FEATURECTRL_PCI))

This function is typically called during component initialization time.

Parameters

IxFeatureCtrlComponentType (in)componentType - the type of a component as defined above as IX_FEATURECTRL_XXX (Exp: IX_FEATURECTRL_PCI, IX_FEATURECTRL_ETH0)

Returns

- IX_FEATURE_CTRL_COMPONENT_ENABLED if component is available
- IX_FEATURE_CTRL_COMPONENT_DISABLED if component is unavailable

A.19.0.57 IxFeatureCtrlReg ixFeatureCtrlHwCapabilityRead (void)

This function reads out the hardware capability of a silicon type as defined in feature control register.This value is different from that returned by ixFeatureCtrlRead() because this function returns the actual hardware component availability.

The bit location of each hardware component is defined above. A value of '1' in bit means the hardware component is not available. A value of '0' means the hardware component is available.

Returns

IxFeatureCtrlReg - the hardware capability of IXP425.

Warning: This function must not be called when IXP425 is running as the result is undefined.

A.19.0.58 IxFeatureCtrlProductId ixFeatureCtrlProductIdRead (void)

This function will return IXP425 product ID i.e. CP15, Register 0.

Returns

IxFeatureCtrlProductId - the value of product ID.

A.19.0.59 IxFeatureCtrlReg ixFeatureCtrlRead (void)

This function reads out the CURRENT value of Feature Control Register. The current value may not be the same as that of the hardware component availability.

The bit location of each hardware component is defined above. A value of '1' in bit means the hardware component is not available. A value of '0' means the hardware component is available.

Returns

IxFeatureCtrlReg - the current value of IXP425 Feature Control Register

A.19.0.60 IX_STATUS ixFeatureCtrlSwConfigurationCheck (IxFeatureCtrlSwConfig *swConfigType*)

This function checks whether the specified software configuration is enabled or disabled.

Usage Example:

- if(IX_FEATURE_CTRL_SWCONFIG_DISABLED != ixFeatureCtrlSwConfigurationCheck(IX_FEATURECTRL_ETH_LEARNING))
- if(IX_FEATURE_CTRL_SWCONFIG_ENABLED == ixFeatureCtrlSwConfigurationCheck(IX_FEATURECTRL_ETH_LEARNING))

This function is typically called during access component initialization time.

Parameters

IxFeatureCtrlSwConfig (in)swConfigType - the type of a software configuration defined in IxFeatureCtrlSwConfig enumeration.

Returns

- IX_FEATURE_CTRL_SWCONFIG_ENABLED if software configuration is enabled.
- IX_FEATURE_CTRL_SWCONFIG_DISABLED if software configuration is disabled.

A.19.0.61 void ixFeatureCtrlSwConfigurationWrite (IxFeatureCtrlSwConfig *swConfigType*, BOOL *enabled*)

This function enable/disable the specified software configuration.

Usage Example:

- ixFeatureCtrlSwConfigurationWrite(IX_FEATURECTRL_ETH_LEARNING, TRUE) is used to enable Ethernet Learning Feature
- ixFeatureCtrlSwConfigurationWrite(IX_FEATURECTRL_ETH_LEARNING, FALSE) is used to disable Ethernet Learning Feature

Parameters

- *IxFeatureCtrlSwConfig* (in)swConfigType - the type of a software configuration defined in IxFeatureCtrlSwConfig enumeration.
- *BOOL* (in)enabled - To enable(TRUE) / disable (FALSE) the specified software configuration.

Returns

none

A.19.0.62 void ixFeatureCtrlWrite (IxFeatureCtrlReg *expUnitReg*)

This function write the value stored in IxFeatureCtrlReg expUnitReg to the Feature Control Register.

The bit location of each hardware component is defined above. The write is only effective on available hardware components. Writing '1' in a bit will software disable the respective hardware component. A '0' will mean that the hardware component will remain to be operable.

Parameters

IxFeatureCtrlReg (in)expUnitReg - The value to be written to feature control register.

Returns

None

A.19.1 Software Configuration for Access Component

Software Configuration for Access ComponentSoftware Configuration for Access ComponentSoftware Configuration for Access ComponentSoftware Configuration for Access ComponentThis section describes software configuration in access component. The configuration can be changed at run-time. ixFeatureCtrlSwConfigurationCheck( ) will be used across applicable access component to check the configuration. ixFeatureCtrlSwConfigurationWrite( ) is used to write the software configuration.

A.19.1.1 Defines

- #define IX_FEATURE_CTRL_SWCONFIG_DISABLED 0
  Software configuration is disabled.
- #define IX_FEATURE_CTRL_SWCONFIG_ENABLED 1
  Software configuration is enabled.

A.19.1.2 Enumerations

- enum IxFeatureCtrlSwConfig { IX_FEATURECTRL_ETH_LEARNING, IX_FEATURECTRL_SWCONFIG_MAX }
  Enumeration for software configuration in access components.

A.19.1.3 Detailed Description

This section describes software configuration in access component. The configuration can be changed at run-time. ixFeatureCtrlSwConfigurationCheck( ) will be used across applicable access component to check the configuration. ixFeatureCtrlSwConfigurationWrite( ) is used to write the software configuration.

Note: All software configurations are default to be enabled.

A.19.1.4 Define Documentation

A.19.1.5 #define IX_FEATURE_CTRL_SWCONFIG_DISABLED 0

Software configuration is disabled.

Definition at line 581 of file IxFeatureCtrl.h.

A.19.1.6 #define IX_FEATURE_CTRL_SWCONFIG_ENABLED 1

Software configuration is enabled.

Definition at line 591 of file IxFeatureCtrl.h.

A.19.1.7 Enumeration Type Documentation

A.19.1.8 enum IxFeatureCtrlSwConfig

Enumeration for software configuration in access components.

Enumeration Values

- ***IX_FEATURECTRL_ETH_LEARNING*** EthDB Learning Feature.
- ***IX_FEATURECTRL_SWCONFIG_MAX*** Maximum boudary for IxFeatureCtrlSwConfig.
  Definition at line 604 of file IxFeatureCtrl.h.

A.20 IXP425 Fast Path Access (IxFpathAcc) API

IXP425 Fast Path Access (IxFpathAcc) API. IXP425 Fast Path Access (IxFpathAcc) APIIXP425 Fast Path Access (IxFpathAcc) APIIXP425 Fast Path Access (IxFpathAcc) APIThe public API for the IXP425 IxFpathAcc Component.

A.20.0.1 Modules

- IXP425 Fast Path Classifier Opcode Definitions
  The definitions for the IXP425 IxFpathAcc Classifier Opcodes.
- IXP425 Fast Path Modifier Template
  The definitions for the IXP425 IxFpathAcc Modifier.

A.20.0.2 Defines

- #define IX_FPATH_POOL_FULL 2
  Buffer pool is full.

A.20.0.3 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.20.0.4 Enumerations

- enum IxFpathEthernetPort { IX_FPATHACC_ETHERNET_PORT0 = 0, IX_FPATHACC_ETHERNET_MAX_PORTS }
  The Ethernet port identifiers.

A.20.0.5 Functions

- PUBLIC IX_STATUS ixFpathAccInit (void)
  Initialise 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.20.0.6 Detailed Description

The public API for the IXP425 IxFpathAcc Component.

A.20.0.7 Define Documentation

A.20.0.8 #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.

Definition at line 178 of file IxFpathAcc.h.

A.20.0.9 Typedef Documentation

A.20.0.10 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 initialisation. 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.

Returns

None

See also:

ixFpathAccBufferPoolUnderRunCallbackSet
ixFpathAccBufferSupply

Definition at line 235 of file IxFpathAcc.h.

A.20.0.11 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.

Returns

None

Definition at line 209 of file IxFpathAcc.h.

A.20.0.12 Enumeration Type Documentation

A.20.0.13 enum IxFpathEthernetPort

The Ethernet port identifiers.

Enumeration Values

- ***IX_FPATHACC_ETHERNET_PORT0*** Ethernet port 0 - attached to NPE-B >.
- ***IX_FPATHACC_ETHERNET_MAX_PORTS*** Maximum value of ethernet port range.

Definition at line 153 of file IxFpathAcc.h.

A.20.0.14 Function Documentation

A.20.0.15 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 curently 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 temaplte is set for a VC or internal error.

A.20.0.16 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 superceed. Hence only one notofication callback is supported.

Notes:

- 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 funciton when invoked. this value is ignored if the callback is NULL.

Returns

- IX_SUCCESS notication callback has been set.
- IX_FAIL IxFpathAcc not yet initialised or some internal error has occurred.

A.20.0.17 ixFpathAccBufferSupply (IX_MBUF * *mBuf*, int * *unpopulatedEntryCnt*)

Populate the Fast Path buffer pool.

This function supplys 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 sufficent 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.20.0.18 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 Ehternet frame header from scratch.

Hence, this function must be called before any fastpath templates are supplied to the service for VCs which build Ehternet 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 Ehternet 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.20.0.19 ixFpathAccInit (void)

Initialise the Fpath Access component.

This function will initialise 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 initialised the component
- IX_FAIL Initialisation failed for some unspecified internal reason.

A.20.0.20 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.20.0.21 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.20.0.22 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.20.0.23 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 classfier 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.20.0.24 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.

Note: the callback function is execute in the context of the IxNpeMh message notification.

See also:

ixFpathAccTemplatePairSet

Note: 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.

Parameters

- *macAddrCallback* new Mac address callback. This parameter can be set to NULL to clear a previously submitted callback.
- *npeVcId* VC indentifier 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 invlaid parameter or some unspecified internal error has occurred. The state of the Npe(s) is indeterminate.

A.20.1 IXP425 Fast Path Classifier Opcode Definitions

IXP425 Fast Path Classifier Opcode Definitions. IXP425 Fast Path Classifier Opcode DefinitionsIXP425 Fast Path Classifier Opcode DefinitionsIXP425 Fast Path Classifier Opcode DefinitionsThe definitions for the IXP425 IxFpathAcc Classifier Opcodes.

A.20.1.1 Defines

- #define IX_FPATHACC_CLS_OPCODE_END 0x00
  End.
- #define IX_FPATHACC_CLS_OPCODE_UPPER_LIMIT 0x01
  UpperLimit.
- #define IX_FPATHACC_CLS_OPCODE_LOWER_LIMIT 0x02
  LowerLimit.
- #define IX_FPATHACC_CLS_OPCODE_MATCH_BYTES 0x03
  Match.
- #define IX_FPATHACC_CLS_OPCODE_CHECKSUM 0x04
  Checksum.
- #define IX_FPATHACC_CLS_OPCODE_1_OUT_OF_4_MATCH 0x05
  1 out of 4 match

- #define IX_FPATHACC_CLS_OPCODE_A_OR_B_MATCH 0x06
  A or B match.
- #define IX_FPATHACC_CLS_OPCODE_WAN_MAC_LEARN 0x07
  Wan MAC Learn.
- #define IX_FPATHACC_CLS_OPCODE_CONDITIONAL_BRANCH 0x08
  Conditional branch.
- #define IX_FPATHACC_CLS_OPCODE_COPY16 0x09
  Save 16 Bytes.
- #define IX_FPATHACC_CLS_OPCODE_NOP_PAD3 0x0A
  Padding.
- #define IX_FPATHACC_CLS_OPCODE_NOP_PAD2 0x0B
  Padding.
- #define IX_FPATHACC_CLS_OPCODE_NOP_PAD1 0x0C
  Padding.
- #define IX_FPATHACC_CLS_OPCODE_NOP 0x0C
  Padding.
- #define IX_FPATHACC_CLS_DUMMY_BYTE 0xFF
  Reserved.

A.20.1.2 Detailed Description

The definitions for the IXP425 IxFpathAcc Classifier Opcodes.

Opcodes used in classifier templates - NPE-A interprets a sequences of these opcodes known as a template.

Classifier Instruction Format

- **OpName** (Opcode 1 byte): Classifier operation
- **Offset** (Nb 1 byte): 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** (Nd 1 byte): Number of bytes of PDU data needed to perform the checking.
- **Parameters** (Np bytes depending on Opcode): Some opcodes require opcode specific parameters. The number of which depends on the opcode and the OperandCount.

```
 * OpName (value)    Offset  OperandCount Parameters
 *
 * 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 NPE to efficiently implement the NOP
 *                                   instructions above.
 *
```

A.20.2 IXP425 Fast Path Modifier Template

IXP425 Fast Path Modifier TemplateIXP425 Fast Path Modifier TemplateIXP425 Fast Path Modifier TemplateIXP425 Fast Path Modifier TemplateThe definitions for the IXP425 IxFpathAcc Modifier. Structure used in modifier templates - NPE-B interprets a sequences of these known as a template.

```
//==========================================================================
// Modification template  structures/constants
//==========================================================================
// TempMdf structure format:
//
//
//   0         2         4         6         8        10        12
//   +----+----+----+----+----+----+----+----+----+----+----+----+
//   | Opcode  |  mData  |  Dst MAC addr 0   |  Dst MAC addr 1   |
//   |         |  offset |    (bytes 0:3)    |    (bytes 0:3)    |
//   +----+----+----+----+----+----+----+----+----+----+----+----+
//  12        14        16        18        20        22        24
//   +----+----+----+----+----+----+----+----+----+----+----+----+
//   |  Dst MAC addr 2   |  Dst MAC addr 3   | DstMAC0 | DstMAC1 |
//   |    (bytes 0:3)    |    (bytes 0:3)    |  (4:5)  |  (4:5)  |
//   +----+----+----+----+----+----+----+----+----+----+----+----+
//  24        26        28        30        32        34        36
//   +----+----+----+----+----+----+----+----+----+----+----+----+
//   | DstMAC2 | DstMAC3 |   Dst IP addr 0   |   Dst IP addr 1   |
//   |  (4:5)  |  (4:5)  |                   |                   |
//   +----+----+----+----+----+----+----+----+----+----+----+----+
//  36        38        40        42        44        46        48
//   +----+----+----+----+----+----+----+----+----+----+----+----+
//   |   Dst IP addr 2   |   Dst IP addr 3   | DstPort | DstPort |
//   |                   |                   |    0    |    1    |
//   +----+----+----+----+----+----+----+----+----+----+----+----+
//  48        50        52        54        56        58        60
//   +----+----+----+----+----+----+----+----+----+----+----+----+
//   | DstPort | DstPort |   IPCS  |   IPCS  |   IPCS  |   IPCS  |
//   |    2    |    3    |    0    |    1    |    2    |    3    |
//   +----+----+----+----+----+----+----+----+----+----+----+----+
//  60        62        64        66        68
//   +----+----+----+----+----+----+----+----+
//   |  PortCS |  PortCS | PortCS  | PortCS  |
//   |    0    |    1    |    2    |    3    |
//   +----+----+----+----+----+----+----+----|
//
// Note that the Source MAC address and Ethernet type (include in the
// pre-pended Ethernet header when IP protocol processing is done) are
```

```
    // the same for all VCs, so these fields are stored in universally
    // accessable global variables.  Also, in the case of static NAT, the
    // single local Destination IP and Port are stored at index of the
    // respective arrays.
    //
    //==========================================================================
*
```

A.21 IXP425 HSS Access (IxHssAcc) API

IXP425 HSS Access (IxHssAcc) API. IXP425 HSS Access (IxHssAcc) APIIXP425 HSS Access (IxHssAcc) APIIXP425 HSS Access (IxHssAcc) APIThe public API for the IXP425 HssAccess component.

A.21.0.1 Data Structures

- struct IxHssAccConfigParams
  Structure containing HSS configuration parameters.
- struct IxHssAccConfigParams
  Structure containing HSS configuration parameters.
- struct IxHssAccPktHdlcFraming
  This structure contains information required by the NPE to configure the HDLC co-processor.
- struct IxHssAccPktHdlcFraming
  This structure contains information required by the NPE to configure the HDLC co-processor.
- struct IxHssAccPortConfig
  Structure containing HSS port configuration parameters.
- struct IxHssAccPortConfig
  Structure containing HSS port configuration parameters.

A.21.0.2 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.21.0.3 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.21.0.4 Enumerations

- enum IxHssAccHssPort { IX_HSSACC_HSS_PORT_0, IX_HSSACC_HSS_PORT_1, IX_HSSACC_HSS_PORT_MAX }
  The HSS port ID - There are two identical ports (0-1).

- 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.21.0.5 Functions

- PUBLIC IX_STATUS ixHssAccPortInit (IxHssAccHssPort hssPortId, IxHssAccConfigParams *configParams, IxHssAccTdmSlotUsage *tdmMap, IxHssAccLastErrorCallback lastHssErrorCallback)

  Initialise 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 initialising 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.21.0.6 Detailed Description

The public API for the IXP425 HssAccess component.

IxHssAcc is the access layer to the HSS packetised and channelised services

A.21.0.7 Define Documentation

A.21.0.8 #define IX_HSSACC_TSLOTS_PER_HSS_PORT 128

The max number of TDM timeslots supported per HSS port - 4E1's = 32x4 = 128.

Definition at line 72 of file IxHssAcc.h.

A.21.0.9 Typedef Documentation

A.21.0.10 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).
- *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

Definition at line 692 of file IxHssAcc.h.

A.21.0.11 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

Definition at line 601 of file IxHssAcc.h.

A.21.0.12 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

Definition at line 625 of file IxHssAcc.h.

A.21.0.13 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

Definition at line 644 of file IxHssAcc.h.

A.21.0.14 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

Definition at line 666 of file IxHssAcc.h.

A.21.0.15 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.

Definition at line 578 of file IxHssAcc.h.

A.21.0.16 Enumeration Type Documentation

A.21.0.17 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.
  Definition at line 407 of file IxHssAcc.h.

A.21.0.18 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.

Definition at line 419 of file IxHssAcc.h.

A.21.0.19 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.

Definition at line 330 of file IxHssAcc.h.

A.21.0.20 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.

Definition at line 280 of file IxHssAcc.h.

A.21.0.21 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.

Definition at line 268 of file IxHssAcc.h.

A.21.0.22 enum IxHssAccClkSpeed

IxHssAccClkSpeed represents the HSS clock speeds available.

Enumeration Values

- ***IX_HSSACC_CLK_SPEED_512KHZ*** 512KHz
- ***IX_HSSACC_CLK_SPEED_1536KHZ*** 1.536MHz
- ***IX_HSSACC_CLK_SPEED_1544KHZ*** 1.544MHz
- ***IX_HSSACC_CLK_SPEED_1568KHZ*** 1.568MHz
- ***IX_HSSACC_CLK_SPEED_2048KHZ*** 2.048MHz
- ***IX_HSSACC_CLK_SPEED_4096KHZ*** 4.096MHz
- ***IX_HSSACC_CLK_SPEED_8192KHZ*** 8.192MHz
- ***IX_HSSACC_CLK_SPEED_MAX*** Delimiter for error checking.

Definition at line 432 of file IxHssAcc.h.

A.21.0.23 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.

Definition at line 368 of file IxHssAcc.h.

A.21.0.24 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.
  Definition at line 316 of file IxHssAcc.h.

A.21.0.25 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.

Definition at line 304 of file IxHssAcc.h.

A.21.0.26 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.
  Definition at line 343 of file IxHssAcc.h.

A.21.0.27 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.

Definition at line 395 of file IxHssAcc.h.

A.21.0.28 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.

Definition at line 292 of file IxHssAcc.h.

A.21.0.29 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.

  Definition at line 251 of file IxHssAcc.h.

A.21.0.30 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.

Definition at line 237 of file IxHssAcc.h.

A.21.0.31 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.
  Definition at line 209 of file IxHssAcc.h.

A.21.0.32 enum IxHssAccHssPort

The HSS port ID - There are two identical ports (0-1).

Enumeration Values

- ***IX_HSSACC_HSS_PORT_0*** HSS Port 0.
- ***IX_HSSACC_HSS_PORT_1*** HSS Port 1.
- ***IX_HSSACC_HSS_PORT_MAX*** Delimiter for error checks.

Definition at line 196 of file IxHssAcc.h.

A.21.0.33 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
  Definition at line 473 of file IxHssAcc.h.

A.21.0.34 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

Definition at line 484 of file IxHssAcc.h.

A.21.0.35 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 occurrance.
- ***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.

  Definition at line 449 of file IxHssAcc.h.

A.21.0.36 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.

Definition at line 355 of file IxHssAcc.h.

A.21.0.37 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.

  Definition at line 223 of file IxHssAcc.h.

A.21.0.38 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.

Definition at line 380 of file IxHssAcc.h.

A.21.0.39 Function Documentation

A.21.0.40 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).
- *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.21.0.41 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).

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.21.0.42 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).

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.21.0.43 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).

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.21.0.44 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).
- *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 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.21.0.45 IX_STATUS ixHssAccInit (void)

This function is responsible for initialising 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.21.0.46 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 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.21.0.47 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).
- *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) - This value determines whether the service will use HDLC data or the debug, raw data type i.e. no HDLC processing
- *unsigned* blockSizeInWords (in) - The max 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.21.0.48 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).
- *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.21.0.49 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).
- *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.21.0.50 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).
- *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.21.0.51 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).
- *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.21.0.52 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).
- *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.21.0.53 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).
- *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

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. 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.21.0.54 IX_STATUS ixHssAccPortInit (IxHssAccHssPort *hssPortId*, IxHssAccConfigParams * *configParams*, IxHssAccTdmSlotUsage * *tdmMap*, IxHssAccLastErrorCallback *lastHssErrorCallback*)

Initialise 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).
- *IxHssAccConfigParams* *configParams (in) - A 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.21.0.55 void ixHssAccShow (void)

This function will display the current state of the IxHssAcc component. The output is sent to stdout.

Returns

void

A.21.0.56 void ixHssAccStatsInit (void)

This function will reset the IxHssAcc statistics.

Returns

void

A.22 IXP425 NPE-A (IxNpeA) API

IXP425 NPE-A (IxNpeA) API. IXP425 NPE-A (IxNpeA) APIIXP425 NPE-A (IxNpeA) APIIXP425 NPE-A (IxNpeA) APIThe Public API for the IXP425 NPE-A.

A.22.0.1 Data Structures

- struct IxNpeA_AtmVcFp
  Atm Descriptor structure used for fpath.
- struct IxNpeA_NpePacketDescriptor

HSS Packetized NpePacket Descriptor Structure.

- struct IxNpeA_RxAtmVc
  Rx Descriptor definition.
- struct IxNpeA_TxAtmVc
  Tx Descriptor definition.

A.22.0.2 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 HssChannelized 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 HssChannelized 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 HssPacketized 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 HssPacketized 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_MPHYMULTIPORT 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 Xscale.
- #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_HSS0_CHL_RX_TRIG 12
  Hardware QMgr Queue ID for HSS Port 0 Channelized Receive trigger.
- #define IX_NPE_A_QMQ_HSS0_PKT_RX 13
  Hardware QMgr Queue ID for HSS Port 0 Packetized Receive.
- #define IX_NPE_A_QMQ_HSS0_PKT_TX0 14
  Hardware QMgr Queue ID for HSS Port 0 Packetized Transmit queue 0.
- #define IX_NPE_A_QMQ_HSS0_PKT_TX1 15
  Hardware QMgr Queue ID for HSS Port 0 Packetized Transmit queue 1.
- #define IX_NPE_A_QMQ_HSS0_PKT_TX2 16
  Hardware QMgr Queue ID for HSS Port 0 Packetized Transmit queue 2.
- #define IX_NPE_A_QMQ_HSS0_PKT_TX3 17
  Hardware QMgr Queue ID for HSS Port 0 Packetized Transmit queue 3.
- #define IX_NPE_A_QMQ_HSS0_PKT_RX_FREE0 18

Hardware QMgr Queue ID for HSS Port 0 Packetized Receive Free queue 0.

- #define IX_NPE_A_QMQ_HSS0_PKT_RX_FREE1 19
  Hardware QMgr Queue ID for HSS Port 0 Packetized Receive Free queue 1.
- #define IX_NPE_A_QMQ_HSS0_PKT_RX_FREE2 20
  Hardware QMgr Queue ID for HSS Port 0 Packetized Receive Free queue 2.
- #define IX_NPE_A_QMQ_HSS0_PKT_RX_FREE3 21
  Hardware QMgr Queue ID for HSS Port 0 Packetized Receive Free queue 3.
- #define IX_NPE_A_QMQ_HSS0_PKT_TX_DONE 22
  Hardware QMgr Queue ID for HSS Port 0 Packetized Transmit Done queue.
- #define IX_NPE_A_QMQ_HSS1_CHL_RX_TRIG 10
  Hardware QMgr Queue ID for HSS Port 1 Channelized Receive trigger.
- #define IX_NPE_A_QMQ_HSS1_PKT_RX 0
  Hardware QMgr Queue ID for HSS Port 1 Packetized Receive.
- #define IX_NPE_A_QMQ_HSS1_PKT_TX0 5
  Hardware QMgr Queue ID for HSS Port 1 Packetized Transmit queue 0.
- #define IX_NPE_A_QMQ_HSS1_PKT_TX1 6
  Hardware QMgr Queue ID for HSS Port 1 Packetized Transmit queue 1.
- #define IX_NPE_A_QMQ_HSS1_PKT_TX2 7
  Hardware QMgr Queue ID for HSS Port 1 Packetized Transmit queue 2.
- #define IX_NPE_A_QMQ_HSS1_PKT_TX3 8
  Hardware QMgr Queue ID for HSS Port 1 Packetized Transmit queue 3.
- #define IX_NPE_A_QMQ_HSS1_PKT_RX_FREE0 1
  Hardware QMgr Queue ID for HSS Port 1 Packetized Receive Free queue 0.
- #define IX_NPE_A_QMQ_HSS1_PKT_RX_FREE1 2
  Hardware QMgr Queue ID for HSS Port 1 Packetized Receive Free queue 1.
- #define IX_NPE_A_QMQ_HSS1_PKT_RX_FREE2 3
  Hardware QMgr Queue ID for HSS Port 1 Packetized Receive Free queue 2.
- #define IX_NPE_A_QMQ_HSS1_PKT_RX_FREE3 4
  Hardware QMgr Queue ID for HSS Port 1 Packetized Receive Free queue 3.
- #define IX_NPE_A_QMQ_HSS1_PKT_TX_DONE 9
  Hardware QMgr Queue ID for HSS Port 1 Packetized 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.
- #define GFC_MASK 0xf0000000
  Mask to acess GFC.

- #define IX_NPE_A_ATMCELLHEADER_GFC_GET(header) (((header) & GFC_MASK) >> 28)
  return GFC from ATM cell header
- #define IX_NPE_A_ATMCELLHEADER_GFC_SET(header, gfc)
  set GFC into ATM cell header
- #define VPI_MASK 0x0ff00000
  Mask to acess VPI.
- #define IX_NPE_A_ATMCELLHEADER_VPI_GET(header) (((header) & VPI_MASK) >> 20)
  return VPI from ATM cell header
- #define IX_NPE_A_ATMCELLHEADER_VPI_SET(header, vpi)
  set VPI into ATM cell header
- #define VCI_MASK 0x000ffff0
  Mask to acess VCI.
- #define IX_NPE_A_ATMCELLHEADER_VCI_GET(header) (((header) & VCI_MASK) >> 4)
  return VCI from ATM cell header
- #define IX_NPE_A_ATMCELLHEADER_VCI_SET(header, vci)
  set VCI into ATM cell header
- #define PTI_MASK 0x0000000e
  Mask to acess PTI.
- #define IX_NPE_A_ATMCELLHEADER_PTI_GET(header) (((header) & PTI_MASK) >> 1)
  return PTI from ATM cell header
- #define IX_NPE_A_ATMCELLHEADER_PTI_SET(header, pti)
  set PTI into ATM cell header
- #define CLP_MASK 0x00000001
  Mask to acess CLP.
- #define IX_NPE_A_ATMCELLHEADER_CLP_GET(header) ((header) & CLP_MASK)
  return CLP from ATM cell header
- #define IX_NPE_A_ATMCELLHEADER_CLP_SET(header, clp)
  set CLP into ATM cell header
- #define STATUS_MASK 0x80000000
  Mask to acess the rxBitField status.
- #define IX_NPE_A_RXBITFIELD_STATUS_GET(rxbitfield) (((rxbitfield) & STATUS_MASK) >> 31)
  return the rxBitField status
- #define IX_NPE_A_RXBITFIELD_STATUS_SET(rxbitfield, status)
  set the rxBitField status
- #define PORT_MASK 0x7f000000

Mask to acess the rxBitField port.

- #define IX_NPE_A_RXBITFIELD_PORT_GET(rxbitfield) (((rxbitfield) & PORT_MASK) >> 24)

  return the rxBitField port

- #define IX_NPE_A_RXBITFIELD_PORT_SET(rxbitfield, port)

  set the rxBitField port

- #define VCID_MASK 0x00ff0000

  Mask to acess the rxBitField vcId.

- #define IX_NPE_A_RXBITFIELD_VCID_GET(rxbitfield) (((rxbitfield) & VCID_MASK) >> 16)

  return the rxBitField vcId

- #define IX_NPE_A_RXBITFIELD_VCID_SET(rxbitfield, vcid)

  set the rxBitField vcId

- #define CURRMBUFSIZE_MASK 0x0000ffff

  Mask to acess the rxBitField mbuf size.

- #define IX_NPE_A_RXBITFIELD_CURRMBUFSIZE_GET(rxbitfield) ((rxbitfield) & CURRMBUFSIZE_MASK)

  return the rxBitField mbuf size

- #define IX_NPE_A_RXBITFIELD_CURRMBUFSIZE_SET(rxbitfield, currmbufsize)

  set the rxBitField mbuf size

A.22.0.3 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.22.0.4 Detailed Description

The Public API for the IXP425 NPE-A.

A.22.0.5 Define Documentation

A.22.0.6 #define IX_NPE_A_ATMCELLHEADER_CLP_SET(header, clp)

Value

```
do { \
    (header) &= ~CLP_MASK; \
    (header) |= ((clp) & CLP_MASK); \
} while(0)
```

set CLP into ATM cell header

Definition at line 1013 of file IxNpeA.h.

A.22.0.7 #define IX_NPE_A_ATMCELLHEADER_GFC_SET(header, gfc)

Value

```
do { \
    (header) &= ~GFC_MASK; \
    (header) |= (((gfc) << 28) & GFC_MASK); \
} while(0)
```

set GFC into ATM cell header

Definition at line 957 of file IxNpeA.h.

A.22.0.8 #define IX_NPE_A_ATMCELLHEADER_PTI_SET(header, pti)

Value

```
do { \
    (header) &= ~PTI_MASK; \
    (header) |= (((pti) << 1) & PTI_MASK); \
} while(0)
```

set PTI into ATM cell header

Definition at line 999 of file IxNpeA.h.

A.22.0.9 #define IX_NPE_A_ATMCELLHEADER_VCI_SET(header, vci)

Value

```
do { \
    (header) &= ~VCI_MASK; \
    (header) |= (((vci) << 4) & VCI_MASK); \
} while(0)
```

set VCI into ATM cell header

Definition at line 985 of file IxNpeA.h.

A.22.0.10 #define IX_NPE_A_ATMCELLHEADER_VPI_SET(header, vpi)

Value

```
do { \
    (header) &= ~VPI_MASK; \
    (header) |= (((vpi) << 20) & VPI_MASK); \
} while(0)
```

set VPI into ATM cell header

Definition at line 971 of file IxNpeA.h.

A.22.0.11 #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.

Definition at line 168 of file IxNpeA.h.

A.22.0.12 #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

Definition at line 158 of file IxNpeA.h.

A.22.0.13 #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

Definition at line 139 of file IxNpeA.h.

A.22.0.14 #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.

Definition at line 129 of file IxNpeA.h.

A.22.0.15 #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

Definition at line 118 of file IxNpeA.h.

A.22.0.16 #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

Definition at line 108 of file IxNpeA.h.

A.22.0.17 #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

Definition at line 861 of file IxNpeA.h.

A.22.0.18 #define IX_NPE_A_RXBITFIELD_CURRMBUFSIZE_SET(rxbitfield, currmbufsize)

Value

```
do { \
    (rxbitfield) &= ~CURRMBUFSIZE_MASK; \
    (rxbitfield) |= ((currmbufsize) & CURRMBUFSIZE_MASK); \
} while(0)
```

set the rxBitField mbuf size

Definition at line 1092 of file IxNpeA.h.

A.22.0.19 #define IX_NPE_A_RXBITFIELD_PORT_SET(rxbitfield, port)

Value

```
do { \
    (rxbitfield) &= ~PORT_MASK; \
    (rxbitfield) |= (((port) << 24) & PORT_MASK); \
} while(0)
```

set the rxBitField port

Definition at line 1064 of file IxNpeA.h.

A.22.0.20 #define IX_NPE_A_RXBITFIELD_STATUS_SET(rxbitfield, status)

Value

```
do { \
    (rxbitfield) &= ~STATUS_MASK; \
    (rxbitfield) |= (((status) << 31) & STATUS_MASK); \
} while(0)
```

set the rxBitField status

Definition at line 1050 of file IxNpeA.h.

A.22.0.21 #define IX_NPE_A_RXBITFIELD_VCID_SET(rxbitfield, vcid)

Value

```
do { \
    (rxbitfield) &= ~VCID_MASK; \
    (rxbitfield) |= (((vcid) << 16) & VCID_MASK); \
```

```
} while(0)
```

set the rxBitField vcId

Definition at line 1078 of file IxNpeA.h.

A.22.0.22 #define IX_NPE_A_RXDESCRIPTOR_AAL5CRCRESIDUE_OFFSET 36

ATM Descriptor structure offset for Receive Descriptor AAL5 CRC Residue.

Current CRC value for a PDU

Definition at line 498 of file IxNpeA.h.

A.22.0.23 #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.

Definition at line 425 of file IxNpeA.h.

A.22.0.24 #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

Definition at line 396 of file IxNpeA.h.

A.22.0.25 #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

Definition at line 434 of file IxNpeA.h.

A.22.0.26 #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.

Definition at line 444 of file IxNpeA.h.

A.22.0.27 #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.

Definition at line 453 of file IxNpeA.h.

A.22.0.28 #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.

Definition at line 471 of file IxNpeA.h.

A.22.0.29 #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.

Definition at line 462 of file IxNpeA.h.

A.22.0.30 #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.

Definition at line 480 of file IxNpeA.h.

A.22.0.31 #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 Xscale to replace the first FastPath mbuf in the associated chain.

Definition at line 414 of file IxNpeA.h.

A.22.0.32 #define IX_NPE_A_RXDESCRIPTOR_SIZE 40

ATM Descriptor structure offset for Receive Descriptor Size.

The size of the Receive descriptor

Definition at line 507 of file IxNpeA.h.

A.22.0.33 #define IX_NPE_A_RXDESCRIPTOR_STATUS_OFFSET 0

ATM Descriptor structure offset for Receive Descriptor Status field.

It is used for descriptor error reporting.

Definition at line 377 of file IxNpeA.h.

A.22.0.34 #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

Definition at line 489 of file IxNpeA.h.

A.22.0.35 #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

Definition at line 386 of file IxNpeA.h.

A.22.0.36 #define IX_NPE_A_TXDESCRIPTOR_AAL5CRCRESIDUE_OFFSET 24

ATM Descriptor structure offset for Transmit Descriptor AAL5 CRC Residue.

Current CRC value for a PDU

Definition at line 581 of file IxNpeA.h.

A.22.0.37 #define IX_NPE_A_TXDESCRIPTOR_CURRMBUFLEN_OFFSET 2

ATM Descriptor structure offset for Transmit Descriptor Current MBuf Length.

TX - Initialized by the XScale 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.

Definition at line 534 of file IxNpeA.h.

A.22.0.38 #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.

Definition at line 556 of file IxNpeA.h.

A.22.0.39 #define IX_NPE_A_TXDESCRIPTOR_PORT_OFFSET 0

ATM Descriptor structure offset for Transmit Descriptor Port.

Port identifier.

Definition at line 516 of file IxNpeA.h.

A.22.0.40 #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

Definition at line 572 of file IxNpeA.h.

A.22.0.41 Enumeration Type Documentation

A.22.0.42 enum IxNpeA_AalType

NPE-A AAL Type.

Enumeration Values

- ***IX_NPE_A_AAL_TYPE_INVALID*** Invalid AAL type.
- ***IX_NPE_A_AAL_TYPE_0_48*** AAL0 - 48 byte.
- ***IX_NPE_A_AAL_TYPE_0_52*** AAL0 - 52 byte.
- ***IX_NPE_A_AAL_TYPE_5*** AAL5.
- ***IX_NPE_A_AAL_TYPE_OAM*** OAM.
  Definition at line 1155 of file IxNpeA.h.

A.22.0.43 enum IxNpeA_PayloadFormat

NPE-A Payload format 52-bytes & 48-bytes.

Enumeration Values

- ***IX_NPE_A_52_BYTE_PAYLOAD*** 52 byte payload
- ***IX_NPE_A_48_BYTE_PAYLOAD*** 48 byte payload

Definition at line 1167 of file IxNpeA.h.

A.23 IXP425 NPE-Downloader (IxNpeDl) API

IXP425 NPE-Downloader (IxNpeDl) API. IXP425 NPE-Downloader (IxNpeDl) APIIXP425 NPE-Downloader (IxNpeDl) APIIXP425 NPE-Downloader (IxNpeDl) APIThe Public API for the IXP425 NPE Downloader.

A.23.0.1 Modules

- IXP425 NPE Image ID Definition
  Definition of NPE Image ID to be passed to ixNpeDlNpeInitAndStart() as input of type UINT32 which has the following fields format:.

A.23.0.2 Data Structures

- struct IxNpeDlImageId
  Image Id to identify each image contained in an image library.

A.23.0.3 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 occuring 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.
- #define ixNpeDlMicrocodeImageOverride(x) ixNpeDlMicrocodeImageLibraryOverride(x)
  Map old terminology that uses term "image" to new term "image library".
- #define IxNpeDlVersionId IxNpeDlImageId
  Map old terminology that uses term "version" to new term "image".
- #define ixNpeDlVersionDownload(x, y) ixNpeDlImageDownload(x,y)
  Map old terminology that uses term "version" to new term "image".
- #define ixNpeDlAvailableVersionsCountGet(x) ixNpeDlAvailableImagesCountGet(x)
  Map old terminology that uses term "version" to new term "image".
- #define ixNpeDlAvailableVersionsListGet(x, y) ixNpeDlAvailableImagesListGet(x,y)
  Map old terminology that uses term "version" to new term "image".
- #define ixNpeDlLoadedVersionGet(x, y) ixNpeDlLoadedImageGet(x,y)
  Map old terminology that uses term "version" to new term "image".
- #define clientImage clientImageLibrary
  Map old terminology that uses term "image" to new term "image library".
- #define versionIdPtr imageIdPtr
  Map old terminology that uses term "version" to new term "image".
- #define numVersionsPtr numImagesPtr
  Map old terminology that uses term "version" to new term "image".
- #define versionIdListPtr imageIdListPtr
  Map old terminology that uses term "version" to new term "image".
- #define IxNpeDlBuildId IxNpeDlFunctionalityId
  Map old terminology that uses term "buildId" to new term "functionalityId".
- #define buildId functionalityId
  Map old terminology that uses term "buildId" to new term "functionalityId".
- #define IX_NPEDL_MicrocodeImage IX_NPEDL_MicrocodeImageLibrary
  Map old terminology that uses term "image" to new term "image library".

A.23.0.4 Typedefs

- typedef UINT8 IxNpeDlFunctionalityId
  Used to make up Functionality ID field of Image Id.
- typedef UINT8 IxNpeDlMajor
  Used to make up Major Release field of Image Id.
- typedef UINT8 IxNpeDlMinor
  Used to make up Minor Revision field of Image Id.

A.23.0.5 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.23.0.6 Functions

- PUBLIC IX_STATUS ixNpeDlNpeInitAndStart (UINT32 npeImageId)
  Stop, reset, download microcode (firmware) and finally start NPE.
- PUBLIC IX_STATUS ixNpeDlCustomImageNpeInitAndStart (UINT32 *imageLibrary, UINT32 npeImageId)
  Stop, reset, download microcode (firmware) and finally start NPE.
- IX_STATUS ixNpeDlMicrocodeImageLibraryOverride (UINT32 *clientImageLibrary)
  This instructs NPE Downloader to use client-supplied microcode image library.
- PUBLIC IX_STATUS ixNpeDlImageDownload (IxNpeDlImageId *imageIdPtr, BOOL verify)
  Stop, reset, download microcode and finally start NPE.
- PUBLIC IX_STATUS ixNpeDlAvailableImagesCountGet (UINT32 *numImagesPtr)
  Get the number of Images available in a microcode image library.
- PUBLIC IX_STATUS ixNpeDlAvailableImagesListGet (IxNpeDlImageId *imageIdListPtr, UINT32 *listSizePtr)
  Get a list of the images available in a microcode image library.
- PUBLIC IX_STATUS ixNpeDlLoadedImageGet (IxNpeDlNpeId npeId, IxNpeDlImageId *imageIdPtr)
  Gets the Id of the image currently loaded on a particular NPE.
- PUBLIC IX_STATUS ixNpeDlLatestImageGet (IxNpeDlNpeId npeId, IxNpeDlFunctionalityId functionalityId, IxNpeDlImageId *imageIdPtr)
  This instructs NPE Downloader to get Id of the latest version for an image that is specified by the client.
- 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.
- IX_STATUS ixNpeDlUnload (void)
  This function will uninitialise the IxNpeDl component.
- 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.23.0.7 Detailed Description

The Public API for the IXP425 NPE Downloader.

A.23.0.8 Define Documentation

A.23.0.9 #define buildId functionalityId

Map old terminology that uses term "buildId" to new term "functionalityId".

Note: **THIS #define HAS BEEN DEPRECATED AND SHOULD NOT BE USED.** It will be removed in a future release. See ixNpeDlNpeInitAndStart for more information.

Definition at line 509 of file IxNpeDl.h.

A.23.0.10 #define clientImage clientImageLibrary

Map old terminology that uses term "image" to new term "image library".

Note: **THIS #define HAS BEEN DEPRECATED AND SHOULD NOT BE USED.** It will be removed in a future release. See ixNpeDlNpeInitAndStart for more information.

Definition at line 449 of file IxNpeDl.h.

A.23.0.11 #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.

Definition at line 93 of file IxNpeDl.h.

A.23.0.12 #define IX_NPEDL_MicrocodeImage IX_NPEDL_MicrocodeImageLibrary

Map old terminology that uses term "image" to new term "image library".

Note: **THIS #define HAS BEEN DEPRECATED AND SHOULD NOT BE USED.** It will be removed in a future release. See ixNpeDlNpeInitAndStart for more information.

Definition at line 521 of file IxNpeDl.h.

A.23.0.13 #define ixNpeDlAvailableVersionsCountGet(x) ixNpeDlAvailableImagesCountGet(x)

Map old terminology that uses term "version" to new term "image".

Note: **THIS #define HAS BEEN DEPRECATED AND SHOULD NOT BE USED.** It will be removed in a future release. See ixNpeDlNpeInitAndStart for more information.

Definition at line 413 of file IxNpeDl.h.

A.23.0.14 #define ixNpeDlAvailableVersionsListGet(x, y) ixNpeDlAvailableImagesListGet(x,y)

Map old terminology that uses term "version" to new term "image".

Note: **THIS #define HAS BEEN DEPRECATED AND SHOULD NOT BE USED.** It will be removed in a future release. See ixNpeDlNpeInitAndStart for more information.

Definition at line 425 of file IxNpeDl.h.

A.23.0.15 #define IxNpeDlBuildId IxNpeDlFunctionalityId

Map old terminology that uses term "buildId" to new term "functionalityId".

Note: **THIS #define HAS BEEN DEPRECATED AND SHOULD NOT BE USED.** It will be removed in a future release. See ixNpeDlNpeInitAndStart for more information.

Definition at line 497 of file IxNpeDl.h.

A.23.0.16 #define ixNpeDlLoadedVersionGet(x, y) ixNpeDlLoadedImageGet(x,y)

Map old terminology that uses term "version" to new term "image".

Note: **THIS #define HAS BEEN DEPRECATED AND SHOULD NOT BE USED.** It will be removed in a future release. See ixNpeDlNpeInitAndStart for more information.

Definition at line 437 of file IxNpeDl.h.

A.23.0.17 #define ixNpeDlMicrocodeImageOverride(x) ixNpeDlMicrocodeImageLibraryOverride(x)

Map old terminology that uses term "image" to new term "image library".

Note: **THIS #define HAS BEEN DEPRECATED AND SHOULD NOT BE USED.** It will be removed in a future release. See ixNpeDlNpeInitAndStart for more information.

Definition at line 377 of file IxNpeDl.h.

A.23.0.18 #define ixNpeDlVersionDownload(x, y) ixNpeDlImageDownload(x,y)

Map old terminology that uses term "version" to new term "image".

Note: **THIS #define HAS BEEN DEPRECATED AND SHOULD NOT BE USED.** It will be removed in a future release. See ixNpeDlNpeInitAndStart for more information.

Definition at line 401 of file IxNpeDl.h.

A.23.0.19 #define IxNpeDlVersionId IxNpeDlImageId

Map old terminology that uses term "version" to new term "image".

Note: **THIS #define HAS BEEN DEPRECATED AND SHOULD NOT BE USED.** It will be removed in a future release. See ixNpeDlNpeInitAndStart for more information.

Definition at line 389 of file IxNpeDl.h.

A.23.0.20 #define numVersionsPtr numImagesPtr

Map old terminology that uses term "version" to new term "image".

Note: **THIS #define HAS BEEN DEPRECATED AND SHOULD NOT BE USED.** It will be removed in a future release. See ixNpeDlNpeInitAndStart for more information.

Definition at line 473 of file IxNpeDl.h.

A.23.0.21 #define versionIdListPtr imageIdListPtr

Map old terminology that uses term "version" to new term "image".

Note: **THIS #define HAS BEEN DEPRECATED AND SHOULD NOT BE USED.** It will be removed in a future release. See ixNpeDlNpeInitAndStart for more information.

Definition at line 485 of file IxNpeDl.h.

A.23.0.22 #define versionIdPtr imageIdPtr

Map old terminology that uses term "version" to new term "image".

Note: **THIS #define HAS BEEN DEPRECATED AND SHOULD NOT BE USED.** It will be removed in a future release. See ixNpeDlNpeInitAndStart for more information.

Definition at line 461 of file IxNpeDl.h.

A.23.0.23 Typedef Documentation

A.23.0.24 IxNpeDlFunctionalityId

Used to make up Functionality ID field of Image Id.

Note: **THIS typedef HAS BEEN DEPRECATED AND SHOULD NOT BE USED.** It will be removed in a future release. See ixNpeDlNpeInitAndStart for more information.

Definition at line 535 of file IxNpeDl.h.

A.23.0.25 IxNpeDlMajor

Used to make up Major Release field of Image Id.

Note: **THIS typedef HAS BEEN DEPRECATED AND SHOULD NOT BE USED.** It will be removed in a future release. See ixNpeDlNpeInitAndStart for more information.

Definition at line 545 of file IxNpeDl.h.

A.23.0.26 IxNpeDlMinor

Used to make up Minor Revision field of Image Id.

Note: **THIS typedef HAS BEEN DEPRECATED AND SHOULD NOT BE USED.** It will be removed in a future release. See ixNpeDlNpeInitAndStart for more information.

Definition at line 555 of file IxNpeDl.h.

A.23.0.27 Enumeration Type Documentation

A.23.0.28 enum IxNpeDlNpeId

NpeId numbers to identify NPE A, B or C.

Note: In this context, for IXP425 Silicon (B0):

- NPE-A has HDLC, HSS, AAL and UTOPIA Coprocessors.
- NPE-B has Ethernet Coprocessor.
- NPE-C has Ethernet, AES, DES and HASH Coprocessors.
- IXP425 Product Line have different combinations of coprocessors.

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.
  Definition at line 570 of file IxNpeDl.h.

A.23.0.29 Function Documentation

A.23.0.30 PUBLIC IX_STATUS ixNpeDlAvailableImagesCountGet (UINT32 * *numImagesPtr*)

Get the number of Images available in a microcode image library.

Parameters

*UINT32** [out] numImagesPtr - A pointer to the number of images in the image library.

Gets the number of images available in the microcode image library. Then returns this in a variable pointed to by *numImagesPtr* .

Note: **THIS FUNCTION HAS BEEN DEPRECATED AND SHOULD NOT BE USED.** It will be removed in a future release. See ixNpeDlNpeInitAndStart and ixNpeDlCustomImageNpeInitAndStart .

Precondition

- The Client should declare the variable to which numImagesPtr points

Returns

- IX_SUCCESS if the operation was successful
- IX_NPEDL_PARAM_ERR if a parameter error occured
- IX_FAIL otherwise

A.23.0.31 PUBLIC IX_STATUS ixNpeDlAvailableImagesListGet (IxNpeDlImageId * *imageIdListPtr*, UINT32 * *listSizePtr*)

Get a list of the images available in a microcode image library.

Parameters

- *IxNpeDlImageId** [out] imageIdListPtr - Array to contain list of image Ids (memory allocated by Client).
- *UINT32** [inout] listSizePtr - As an input, this param should point to the max number of Ids the *imageIdListPtr* array can hold. NpeDl will replace the input value of this parameter with the number of image Ids actually filled into the array before returning.

  Finds list of images available in the microcode image library. Fills these into the array pointed to by *imageIdListPtr*

Note: **THIS FUNCTION HAS BEEN DEPRECATED AND SHOULD NOT BE USED.** It will be removed in a future release. See ixNpeDlNpeInitAndStart and ixNpeDlCustomImageNpeInitAndStart .

Precondition

- The Client should declare the variable to which numImagesPtr points
- The Client should create an array (*imageIdListPtr*) large enough to contain all the image Ids in the image library

Returns

- IX_SUCCESS if the operation was successful
- IX_NPEDL_PARAM_ERR if a parameter error occured
- IX_FAIL otherwise

A.23.0.32 PUBLIC IX_STATUS ixNpeDlCustomImageNpeInitAndStart (UINT32 * *imageLibrary*, UINT32 *npeImageId*)

Stop, reset, download microcode (firmware) and finally start NPE.

Parameters

UINT32 [in] imageId - Id of the microcode image to download.

This function locates the image specified by the *imageId* parameter from the specified microcode image library which is pointed to by the *imageLibrary* parameter. It then stops and resets the NPE, loads the firmware image onto the NPE, and then restarts the NPE.

This is a facility for users who wish to use an image from an external library of NPE firmware images. To use a standard image from the built-in library, see ixNpeDlNpeInitAndStart instead.

Notes:

- A list of valid image IDs is included in this header file. See #defines with prefix IX_NPEDL_NPEIMAGE_...
- This function was added to simplify the IxNpeDl API. As a result, the following functions are deprecated and will be removed completely in a future release:
- ixNpeDlImageDownload
- ixNpeDlAvailableImagesCountGet
- ixNpeDlAvailableImagesListGet
- ixNpeDlLatestImageGet
- ixNpeDlLoadedImageGet
- ixNpeDlMicrocodeImageLibraryOverride

Precondition

- The Client is responsible for ensuring mutual access to the NPE.
- The image library supplied must be in the correct format for use by the NPE Downloader (IxNpeDl) component. Details of the library format are contained in the Functional Specification document for IxNpeDl.

  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 ususable state.

Returns

- IX_SUCCESS if the download was successful;
- IX_NPEDL_PARAM_ERR if a parameter error occured

- IX_NPEDL_CRITICAL_NPE_ERR if a critical NPE error occured during download
- IX_PARAM_CRITICAL_MICROCODE_ERR if a critical microcode error occured during download
- IX_FAIL if NPE is not available or image is failed to be located. A warning is issued if the NPE is not present.

A.23.0.33 PUBLIC IX_STATUS ixNpeDlImageDownload (IxNpeDlImageId * *imageIdPtr*, BOOL *verify*)

Stop, reset, download microcode and finally start NPE.

Parameters

- *IxNpeDlImageId** [in] imageIdPtr - Pointer to Id of the microcode image 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 *imageIdPtr* , this function locates a particular image of microcode in the microcode image library in memory, and downloads the microcode to a particular NPE.

Note: **THIS FUNCTION HAS BEEN DEPRECATED AND SHOULD NOT BE USED.** It will be removed in a future release. See ixNpeDlNpeInitAndStart and ixNpeDlCustomImageNpeInitAndStart

Precondition

- The Client is responsible for ensuring mutual access to the NPE.

  The Client should use ixNpeDlLatestImageGet() to obtain the latest version of the image before attempting download.

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 ususable state.

Returns

- IX_SUCCESS if the download was successful;
- IX_NPEDL_PARAM_ERR if a parameter error occured
- IX_NPEDL_CRITICAL_NPE_ERR if a critical NPE error occured during download
- IX_PARAM_CRITICAL_MICROCODE_ERR if a critical microcode error occured during download
- IX_FAIL if NPE is not available or image is failed to be located. A warning is issued if the NPE is not present.

A.23.0.34 PUBLIC IX_STATUS ixNpeDlLatestImageGet (IxNpeDlNpeId *npeId*, IxNpeDlFunctionalityId *functionalityId*, IxNpeDlImageId * *imageIdPtr*)

This instructs NPE Downloader to get Id of the latest version for an image that is specified by the client.

Parameters

- *IxNpeDlNpeId* [in] npeId - Id of the target NPE.
- *IxNpeDlFunctionalityId* [in] functionalityId - functionality of the image
- *IxNpeDlImageId** [out] imageIdPtr - Pointer to the where the image id should be stored.

This function sets NPE Downloader to return the id of the latest version for image. The user will select the image by providing a particular NPE (specifying *npeId*) with particular functionality (specifying *FunctionalityId*). The most recent version available as determined by the highest Major and Minor revision numbers is returned in *imageIdPtr* .

Note: **THIS FUNCTION HAS BEEN DEPRECATED AND SHOULD NOT BE USED.** It will be removed in a future release. See ixNpeDlNpeInitAndStart and ixNpeDlCustomImageNpeInitAndStart .

Returns

- IX_SUCCESS if the operation was successful
- IX_NPEDL_PARAM_ERR if a parameter error occured
- IX_FAIL otherwise

A.23.0.35 PUBLIC IX_STATUS ixNpeDlLoadedImageGet (IxNpeDlNpeId *npeId*, IxNpeDlImageId * *imageIdPtr*)

Gets the Id of the image currently loaded on a particular NPE.

Parameters

- *IxNpeDlNpeId* [in] npeId - Id of the target NPE.
- *IxNpeDlImageId** [out] imageIdPtr - Pointer to the where the image id should be stored.

If an image of microcode was previously downloaded successfully to the NPE by NPE Downloader, this function returns in *imageIdPtr* the image Id of that image loaded on the NPE.

Note: **THIS FUNCTION HAS BEEN DEPRECATED AND SHOULD NOT BE USED.** It will be removed in a future release.

Precondition

- The Client has allocated memory to the *imageIdPtr* pointer.

Returns

- IX_SUCCESS if the operation was successful
- IX_NPEDL_PARAM_ERR if a parameter error occured
- IX_FAIL if the NPE doesn't currently have a image loaded

A.23.0.36 IX_STATUS ixNpeDlMicrocodeImageLibraryOverride (UINT32 * *clientImageLibrary*)

This instructs NPE Downloader to use client-supplied microcode image library.

Parameters

*UINT32** [in] clientImageLibrary - Pointer to the client-supplied NPE microcode image library

This function sets NPE Downloader to use a client-supplied microcode image library instead of the standard image library 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 library, local to this component, which should always contain the latest microcode for the NPEs.

Note: **THIS FUNCTION HAS BEEN DEPRECATED AND SHOULD NOT BE USED.** It will be removed in a future release. See ixNpeDlCustomImageNpeInitAndStart .

Precondition

- *clientImageLibrary* should point to a microcode image library valid for use by the NPE Downloader component.

  Postcondition:

- the client-supplied image library 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 occured
- IX_FAIL if the client-supplied image library did not contain a valid signature

A.23.0.37 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.

Note: It is no longer necessary to call this function after downloading a new image to the NPE. It is left on the API only to allow greater control of NPE execution if required. Where appropriate, use ixNpeDlNpeInitAndStart or ixNpeDlCustomImageNpeInitAndStart instead.

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()).

Returns

- IX_SUCCESS if the operation was successful
- IX_NPEDL_PARAM_ERR if a parameter error occured

- IX_FAIL otherwise

A.23.0.38 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 image while leaving the current state of the NPE intact..

Precondition

The Client is responsible for ensuring mutual access to the NPE.

Returns

- IX_SUCCESS if the operation was successful
- IX_NPEDL_PARAM_ERR if a parameter error occured
- IX_FAIL otherwise

A.23.0.39 PUBLIC IX_STATUS ixNpeDlNpeInitAndStart (UINT32 *npeImageId*)

Stop, reset, download microcode (firmware) and finally start NPE.

Parameters

UINT32 [in] imageId - Id of the microcode image to download.

This function locates the image specified by the *imageId* parameter from the default microcode image library which is included internally by this component. It then stops and resets the NPE, loads the firmware image onto the NPE, and then restarts the NPE.

Notes:

- A list of valid image IDs is included in this header file. See #defines with prefix IX_NPEDL_NPEIMAGE_...
- This function was added to simplify the IxNpeDl API. As a result, the following functions are deprecated and will be removed completely in a future release:
- ixNpeDlImageDownload
- ixNpeDlAvailableImagesCountGet
- ixNpeDlAvailableImagesListGet
- ixNpeDlLatestImageGet
- ixNpeDlLoadedImageGet

- ixNpeDlMicrocodeImageLibraryOverride

Precondition

- The Client is responsible for ensuring mutual access to the NPE.

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 ususable state.

Returns

- IX_SUCCESS if the download was successful;
- IX_NPEDL_PARAM_ERR if a parameter error occured
- IX_NPEDL_CRITICAL_NPE_ERR if a critical NPE error occured during download
- IX_PARAM_CRITICAL_MICROCODE_ERR if a critical microcode error occured during download
- IX_FAIL if NPE is not available or image is failed to be located. A warning is issued if the NPE is not present.

A.23.0.40 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. This implicitly stops NPE code execution before resetting the NPE.

Note: It is no longer necessary to call this function before downloading a new image to the NPE. It is left on the API only to allow greater control of NPE execution if required. Where appropriate, use ixNpeDlNpeInitAndStart or ixNpeDlCustomImageNpeInitAndStart instead.

Precondition

- The Client is responsible for ensuring mutual access to the NPE.

Returns

- IX_SUCCESS if the operation was successful
- IX_NPEDL_PARAM_ERR if a parameter error occured
- IX_FAIL otherwise

A.23.0.41 PUBLIC void ixNpeDlStatsReset (void)

This function will reset the statistics of the IxNpeDl component.

Returns

none

A.23.0.42 PUBLIC void ixNpeDlStatsShow (void)

This function will display run-time statistics from the IxNpeDl component.

Returns

none

A.23.0.43 PUBLIC IX_STATUS ixNpeDlUnload (void)

This function will uninitialise the IxNpeDl component.

This function will uninitialise the IxNpeDl component. It should only be called once, and only if the IxNpeDl component has already been initialised by calling any of the following functions:

- ixNpeDlNpeInitAndStart
- ixNpeDlCustomImageNpeInitAndStart
- ixNpeDlImageDownload (deprecated)
- ixNpeDlNpeStopAndReset
- ixNpeDlNpeExecutionStop
- ixNpeDlNpeExecutionStart

If possible, this function should be called before a soft reboot or unloading a kernel module to perform any clean up operations required for IxNpeDl.

The following actions will be performed by this function:

Unmapping of any kernel memory mapped by IxNpeDl

Returns

- IX_SUCCESS if the operation was successful
- IX_FAIL otherwise

A.23.1 IXP425 NPE Image ID Definition

IXP425 NPE Image ID DefinitionIXP425 NPE Image ID DefinitionIXP425 NPE Image ID DefinitionIXP425 NPE Image ID DefinitionDefinition of NPE Image ID to be passed to ixNpeDlNpeInitAndStart() as input of type UINT32 which has the following fields format:.

A.23.1.1 Defines

- #define IX_NPEDL_NPEIMAGE_FIELD_MASK 0xff
  Mask for NPE Image ID's Field.
- #define IX_NPEDL_NPEIMAGE_BIT_LOC_NPEID 24

Location of NPE ID field in term of bit.

- #define IX_NPEDL_NPEIMAGE_BIT_LOC_FUNCTIONALITYID 16

  Location of Functionality ID field in term of bit.

- #define IX_NPEDL_NPEIMAGE_BIT_LOC_MAJOR 8

  Location of Major Release Number field in term of bit.

- #define IX_NPEDL_NPEIMAGE_BIT_LOC_MINOR 0

  Location of Minor Release Number field in term of bit.

- #define IX_NPEDL_NPEIMAGE_NPEA_HSS0 0x00010000

  NPE Image Id for NPE-A with HSS-0 Only feature. It supports 32 channelized and 4 packetized.

- #define IX_NPEDL_NPEIMAGE_NPEA_HSS0_ATM_SPHY_1_PORT 0x00020000

  NPE Image Id for NPE-A with HSS-0 and ATM feature. For HSS, it supports 16/32 channelized and 4/0 packetized. For ATM, it supports AAL5, AAL0 and OAM for UTOPIA SPHY, 1 logical port, 32 VCs. It also has Fast Path support.

- #define IX_NPEDL_NPEIMAGE_NPEA_HSS0_ATM_MPHY_1_PORT 0x00030000

  NPE Image Id for NPE-A with HSS-0 and ATM feature. For HSS, it supports 16/32 channelized and 4/0 packetized. For ATM, it supports AAL5, AAL0 and OAM for UTOPIA MPHY, 1 logical port, 32 VCs. It also has Fast Path support.

- #define IX_NPEDL_NPEIMAGE_NPEA_ATM_MPHY_12_PORT 0x00040000

  NPE Image Id for NPE-A with ATM-Only feature. It supports AAL5, AAL0 and OAM for UTOPIA MPHY, 12 logical ports, 32 VCs. It also has Fast Path support.

- #define IX_NPEDL_NPEIMAGE_NPEA_HSS_2_PORT 0x00090000

  NPE Image Id for NPE-A with HSS-0 and HSS-1 feature. Each HSS port supports 32 channelized and 4 packetized.

- #define IX_NPEDL_NPEIMAGE_NPEA_DMA 0x00150100

  NPE Image Id for NPE-A with DMA-Only feature.

- #define IX_NPEDL_NPEIMAGE_NPEB_ETH 0x01000100

  NPE Image Id for NPE-B with Ethernet-Only feature.

- #define IX_NPEDL_NPEIMAGE_NPEB_ETH_FPATH 0x01010100

  NPE Image Id for NPE-B with Ethernet and Fast Path feature.

- #define IX_NPEDL_NPEIMAGE_NPEB_DMA 0x01020100

  NPE Image Id for NPE-B with DMA-Only feature.

- #define IX_NPEDL_NPEIMAGE_NPEC_ETH 0x02000100

  NPE Image Id for NPE-C with Eth-Only feature.

- #define IX_NPEDL_NPEIMAGE_NPEC_CRYPTO 0x02020100

  NPE Image Id for NPE-C with Crypto-Only feature. For Crypto, it supports DES, SHA-1, MD5.

- #define IX_NPEDL_NPEIMAGE_NPEC_CRYPTO_AES 0x02030100

  NPE Image Id for NPE-C with Crypto-Only feature. For Crypto, it supports AES, DES, SHA-1, MD5.

- #define IX_NPEDL_NPEIMAGE_NPEC_CRYPTO_ETH 0x02040100

NPE Image Id for NPE-C with Crypto and Eth feature. For Crypto, it supports DES, SHA-1, MD5.

- #define IX_NPEDL_NPEIMAGE_NPEC_CRYPTO_AES_ETH 0x02050100
  NPE Image Id for NPE-C with Crypto and Eth feature. For Crypto, it supports AES, DES, SHA-1, MD5.
- #define IX_NPEDL_NPEIMAGE_NPEC_DMA 0x02080100
  NPE Image Id for NPE-C with DMA-Only feature.

A.23.1.2 Detailed Description

Definition of NPE Image ID to be passed to ixNpeDlNpeInitAndStart() as input of type UINT32 which has the following fields format:.

Field [Bit Location]

NPE ID [31 - 24]

NPE Functionality ID [23 - 16]

Major Release Number [15 - 8]

Minor Release Number [7 - 0]

A.23.1.3 Define Documentation

A.23.1.4 #define IX_NPEDL_NPEIMAGE_BIT_LOC_FUNCTIONALITYID 16

Location of Functionality ID field in term of bit.

Note: **THIS #define HAS BEEN DEPRECATED AND SHOULD NOT BE USED.** It will be removed in a future release. See ixNpeDlNpeInitAndStart for more information.

Definition at line 145 of file IxNpeDl.h.

A.23.1.5 #define IX_NPEDL_NPEIMAGE_BIT_LOC_MAJOR 8

Location of Major Release Number field in term of bit.

Note: **THIS #define HAS BEEN DEPRECATED AND SHOULD NOT BE USED.** It will be removed in a future release. See ixNpeDlNpeInitAndStart for more information.

Definition at line 156 of file IxNpeDl.h.

A.23.1.6 #define IX_NPEDL_NPEIMAGE_BIT_LOC_MINOR 0

Location of Minor Release Number field in term of bit.

Note: **THIS #define HAS BEEN DEPRECATED AND SHOULD NOT BE USED.** It will be removed in a future release. See ixNpeDlNpeInitAndStart for more information.

Definition at line 167 of file IxNpeDl.h.

A.23.1.7 #define IX_NPEDL_NPEIMAGE_BIT_LOC_NPEID 24

Location of NPE ID field in term of bit.

Note: **THIS #define HAS BEEN DEPRECATED AND SHOULD NOT BE USED.** It will be removed in a future release. See ixNpeDlNpeInitAndStart for more information.

Definition at line 134 of file IxNpeDl.h.

A.23.1.8 #define IX_NPEDL_NPEIMAGE_FIELD_MASK 0xff

Mask for NPE Image ID's Field.

Note: **THIS #define HAS BEEN DEPRECATED AND SHOULD NOT BE USED.** It will be removed in a future release. See ixNpeDlNpeInitAndStart for more information.

Definition at line 123 of file IxNpeDl.h.

A.23.1.9 #define IX_NPEDL_NPEIMAGE_NPEA_ATM_MPHY_12_PORT 0x00040000

NPE Image Id for NPE-A with ATM-Only feature. It supports AAL5, AAL0 and OAM for UTOPIA MPHY, 12 logical ports, 32 VCs. It also has Fast Path support.

This is intended for use as a parameter with any of the following functions:

- ixNpeDlNpeInitAndStart
- ixNpeDlCustomImageNpeInitAndStart

Definition at line 224 of file IxNpeDl.h.

A.23.1.10 #define IX_NPEDL_NPEIMAGE_NPEA_DMA 0x00150100

NPE Image Id for NPE-A with DMA-Only feature.

This is intended for use as a parameter with any of the following functions:

- ixNpeDlNpeInitAndStart
- ixNpeDlCustomImageNpeInitAndStart

Definition at line 247 of file IxNpeDl.h.

A.23.1.11 #define IX_NPEDL_NPEIMAGE_NPEA_HSS0 0x00010000

NPE Image Id for NPE-A with HSS-0 Only feature. It supports 32 channelized and 4 packetized.

This is intended for use as a parameter with any of the following functions:

- ixNpeDlNpeInitAndStart
- ixNpeDlCustomImageNpeInitAndStart

Definition at line 183 of file IxNpeDl.h.

A.23.1.12 #define IX_NPEDL_NPEIMAGE_NPEA_HSS0_ATM_MPHY_1_PORT 0x00030000

NPE Image Id for NPE-A with HSS-0 and ATM feature. For HSS, it supports 16/32 channelized and 4/0 packetized. For ATM, it supports AAL5, AAL0 and OAM for UTOPIA MPHY, 1 logical port, 32 VCs. It also has Fast Path support.

This is intended for use as a parameter with any of the following functions:

- ixNpeDlNpeInitAndStart
- ixNpeDlCustomImageNpeInitAndStart

Definition at line 211 of file IxNpeDl.h.

A.23.1.13 #define IX_NPEDL_NPEIMAGE_NPEA_HSS0_ATM_SPHY_1_PORT 0x00020000

NPE Image Id for NPE-A with HSS-0 and ATM feature. For HSS, it supports 16/32 channelized and 4/0 packetized. For ATM, it supports AAL5, AAL0 and OAM for UTOPIA SPHY, 1 logical port, 32 VCs. It also has Fast Path support.

This is intended for use as a parameter with any of the following functions:

- ixNpeDlNpeInitAndStart
- ixNpeDlCustomImageNpeInitAndStart

Definition at line 197 of file IxNpeDl.h.

A.23.1.14 #define IX_NPEDL_NPEIMAGE_NPEA_HSS_2_PORT 0x00090000

NPE Image Id for NPE-A with HSS-0 and HSS-1 feature. Each HSS port supports 32 channelized and 4 packetized.

This is intended for use as a parameter with any of the following functions:

- ixNpeDlNpeInitAndStart
- ixNpeDlCustomImageNpeInitAndStart

Definition at line 236 of file IxNpeDl.h.

A.23.1.15 #define IX_NPEDL_NPEIMAGE_NPEB_DMA 0x01020100

NPE Image Id for NPE-B with DMA-Only feature.

This is intended for use as a parameter with any of the following functions:

- ixNpeDlNpeInitAndStart
- ixNpeDlCustomImageNpeInitAndStart

Definition at line 284 of file IxNpeDl.h.

A.23.1.16 #define IX_NPEDL_NPEIMAGE_NPEB_ETH 0x01000100

NPE Image Id for NPE-B with Ethernet-Only feature.

This is intended for use as a parameter with any of the following functions:

- ixNpeDlNpeInitAndStart
- ixNpeDlCustomImageNpeInitAndStart

Definition at line 262 of file IxNpeDl.h.

A.23.1.17 #define IX_NPEDL_NPEIMAGE_NPEB_ETH_FPATH 0x01010100

NPE Image Id for NPE-B with Ethernet and Fast Path feature.

This is intended for use as a parameter with any of the following functions:

- ixNpeDlNpeInitAndStart
- ixNpeDlCustomImageNpeInitAndStart

Definition at line 273 of file IxNpeDl.h.

A.23.1.18 #define IX_NPEDL_NPEIMAGE_NPEC_CRYPTO 0x02020100

NPE Image Id for NPE-C with Crypto-Only feature. For Crypto, it supports DES, SHA-1, MD5.

This is intended for use as a parameter with any of the following functions:

- ixNpeDlNpeInitAndStart
- ixNpeDlCustomImageNpeInitAndStart

Definition at line 311 of file IxNpeDl.h.

A.23.1.19 #define IX_NPEDL_NPEIMAGE_NPEC_CRYPTO_AES 0x02030100

NPE Image Id for NPE-C with Crypto-Only feature. For Crypto, it supports AES, DES, SHA-1, MD5.

This is intended for use as a parameter with any of the following functions:

- ixNpeDlNpeInitAndStart
- ixNpeDlCustomImageNpeInitAndStart

Definition at line 323 of file IxNpeDl.h.

A.23.1.20 #define IX_NPEDL_NPEIMAGE_NPEC_CRYPTO_AES_ETH 0x02050100

NPE Image Id for NPE-C with Crypto and Eth feature. For Crypto, it supports AES, DES, SHA-1, MD5.

This is intended for use as a parameter with any of the following functions:

- ixNpeDlNpeInitAndStart
- ixNpeDlCustomImageNpeInitAndStart

Definition at line 347 of file IxNpeDl.h.

A.23.1.21 #define IX_NPEDL_NPEIMAGE_NPEC_CRYPTO_ETH 0x02040100

NPE Image Id for NPE-C with Crypto and Eth feature. For Crypto, it supports DES, SHA-1, MD5.

This is intended for use as a parameter with any of the following functions:

- ixNpeDlNpeInitAndStart
- ixNpeDlCustomImageNpeInitAndStart

Definition at line 335 of file IxNpeDl.h.

A.23.1.22 #define IX_NPEDL_NPEIMAGE_NPEC_DMA 0x02080100

NPE Image Id for NPE-C with DMA-Only feature.

This is intended for use as a parameter with any of the following functions:

- ixNpeDlNpeInitAndStart
- ixNpeDlCustomImageNpeInitAndStart

Definition at line 358 of file IxNpeDl.h.

A.23.1.23 #define IX_NPEDL_NPEIMAGE_NPEC_ETH 0x02000100

NPE Image Id for NPE-C with Eth-Only feature.

This is intended for use as a parameter with any of the following functions:

- ixNpeDlNpeInitAndStart
- ixNpeDlCustomImageNpeInitAndStart

Definition at line 299 of file IxNpeDl.h.

A.24 IXP425 NPE Message Handler (IxNpeMh) API

IXP425 NPE Message Handler (IxNpeMh) API. IXP425 NPE Message Handler (IxNpeMh) APIIXP425 NPE Message Handler (IxNpeMh) APIIXP425 NPE Message Handler (IxNpeMh) APIThe public API for the IXP425 NPE Message Handler component.

A.24.0.1 Data Structures

- struct IxNpeMhMessage
  The 2-word message structure to send to and receive from the NPEs.

A.24.0.2 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.24.0.3 Typedefs

- typedef UINT32 IxNpeMhMessageId
  message ID
- typedef void(* IxNpeMhCallback)(IxNpeMhNpeId, IxNpeMhMessage)
  This prototype shows the format of a message callback function.

A.24.0.4 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.24.0.5 Functions

- IX_STATUS ixNpeMhInitialize (IxNpeMhNpeInterrupts npeInterrupts)
  This function will initialise the IxNpeMh component.
- IX_STATUS ixNpeMhUnload (void)
  This function will uninitialise 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.

- IX_STATUS ixNpeMhShowReset (IxNpeMhNpeId npeId)

  This function will reset the current state of the IxNpeMh component.

A.24.0.6 Detailed Description

The public API for the IXP425 NPE Message Handler component.

A.24.0.7 Typedef Documentation

A.24.0.8 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 NPE's "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: This function is only a prototype, and will be implemented by the client. It does not need to be re-entrant.

Definition at line 127 of file IxNpeMh.h.

A.24.0.9 Enumeration Type Documentation

A.24.0.10 enum IxNpeMhNpeId

The ID of a particular NPE.

Note: In this context, for IXP425 Silicon (B0):

- NPE-A has HDLC, HSS, AAL and UTOPIA Coprocessors.
- NPE-B has Ethernet Coprocessor.
- NPE-C has Ethernet, AES, DES and HASH Coprocessors.

- IXP425 Product Line have different combinations of coprocessors.

Enumeration Values

- ***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.

Definition at line 76 of file IxNpeMh.h.

A.24.0.11 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.

Definition at line 91 of file IxNpeMh.h.

A.24.0.12 Function Documentation

A.24.0.13 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.24.0.14 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 Xscale 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.24.0.15 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 initialised 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.24.0.16 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 recognised 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 Xscale 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.24.0.17 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.24.0.18 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.24.0.19 IX_STATUS ixNpeMhUnload (void)

This function will uninitialise the IxNpeMh component.

This function will uninitialise the IxNpeMh component. It should only be called once, and only if the IxNpeMh component has already been initialised. No other IxNpeMh API functions should be called until ixNpeMhInitialize is called again. If possible, this function should be called before a soft reboot or unloading a kernel module to perform any clean up operations required for IxNpeMh.

The following actions will be performed by this function:

Unmapping of kernel memory mapped by the function ixNpeMhInitialize .

Returns

The function returns a status indicating success or failure.

A.24.0.20 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.24.0.21 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.25 IXP425 OS Memory Buffer Management (IxOsBuffMgt) API

IXP425 OS Memory Buffer Management (IxOsBuffMgt) API. IXP425 OS Memory Buffer Management (IxOsBuffMgt) APIIXP425 OS Memory Buffer Management (IxOsBuffMgt) APIIXP425 OS Memory Buffer Management (IxOsBuffMgt) APIIXP425 OS Memory Buffer Management.

A.25.0.1 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.25.0.2 Detailed Description

IXP425 OS Memory Buffer Management.

A.26 IXP425 OS Memory Buffer Pool Management (IxOsBuffPoolMgt) API

IXP425 OS Memory Buffer Pool Management (IxOsBuffPoolMgt) API. IXP425 OS Memory Buffer Pool Management (IxOsBuffPoolMgt) APIIXP425 OS Memory Buffer Pool Management (IxOsBuffPoolMgt) APIIXP425 OS Memory Buffer Pool Management (IxOsBuffPoolMgt) APIThe Public API for the Buffer Pool Management component.

A.26.0.1 Data Structures

- struct IxMbufPool

  Implementation of buffer pool structure for use with non-VxWorks OS.

A.26.0.2 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) ixOsBuffPoolMbufAreaSizeGet(count)

  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) ixOsBuffPoolDataAreaSizeGet((count), (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 32

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.26.0.3 Typedefs

- typedef IxMbufPool **IX_MBUF_POOL**

A.26.0.4 Enumerations

- enum IxMbufPoolAllocationType { IX_MBUF_POOL_TYPE_SYS_ALLOC = 0, IX_MBUF_POOL_TYPE_USER_ALLOC }
  Used to indicate how the pool memory was allocated.

A.26.0.5 Functions

- IX_STATUS ixOsBuffPoolInit (IX_MBUF_POOL **poolPtrPtr, int count, int size, char *name)
  This function creates a new buffer pool.
- IX_MBUF_POOL * ixOsBuffPoolAllocate (void)
  This function allocates buffers from the available pool.
- UINT32 ixOsBuffPoolDataAreaSizeGet (int count, int size)
  This function calcultes the size of data memory required to create a new buffer pool.
- IX_MBUF * ixOsBuffPoolMbufInit (int mbufSizeAligned, int dataSizeAligned, IX_MBUF_POOL *poolPtr)

Allocate memory for mbuf and data and initialise mbuf header fields.

- UINT32 ixOsBuffPoolMbufAreaSizeGet (int count)
  This function calcultes the size of mbuf memory required to create 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.
- IX_STATUS ixOsBuffPoolUninit (IX_MBUF_POOL *pool)
  Unitialize buffer pool.

A.26.0.6 Detailed Description

The Public API for the Buffer Pool Management component.

A.26.0.7 Define Documentation

A.26.0.8 #define IX_MBUF_MAX_POOLS 32

The maximum number of pools that can be allocated.

Note: This can safely be increased if more pools are required.

Definition at line 241 of file IxOsBuffPoolMgt.h.

A.26.0.9 #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

Definition at line 197 of file IxOsBuffPoolMgt.h.

A.26.0.10 #define IX_MBUF_POOL_DATA_AREA_SIZE_ALIGNED(count, size) ixOsBuffPoolDataAreaSizeGet((count), (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 - The number of mbufs in the pool.
- *int* [in] size - The 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)

Definition at line 158 of file IxOsBuffPoolMgt.h.

A.26.0.11 #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

Definition at line 211 of file IxOsBuffPoolMgt.h.

A.26.0.12 #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

Definition at line 175 of file IxOsBuffPoolMgt.h.

A.26.0.13 #define IX_MBUF_POOL_MBUF_AREA_SIZE_ALIGNED(count) ixOsBuffPoolMbufAreaSizeGet(count)

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 - the number of buffers the pool will contain

Returns

int - the total size required for the pool mbuf area (aligned)

Definition at line 134 of file IxOsBuffPoolMgt.h.

A.26.0.14 #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

Definition at line 110 of file IxOsBuffPoolMgt.h.

A.26.0.15 Enumeration Type Documentation

A.26.0.16 enum IxMbufPoolAllocationType

Used to indicate how the pool memory was allocated.

Enumeration vaLues

- ***IX_MBUF_POOL_TYPE_SYS_ALLOC*** mbuf pool allocated by the system
- ***IX_MBUF_POOL_TYPE_USER_ALLOC*** mbuf pool allocated by the user

Definition at line 258 of file IxOsBuffPoolMgt.h.

A.26.0.17 Function Documentation

A.26.0.18 IX_MBUF_POOL * ixOsBuffPoolAllocate (void)

This function allocates buffers from the available pool.

Thread Safe: yes

Returns

IX_MBUF_POOL - pointer to mbuf pool.

A.26.0.19 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

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.26.0.20 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).

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

Precondition

- *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.26.0.21 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

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.26.0.22 ixOsBuffPoolDataAreaSizeGet (int *count*, int *size*)

This function calcultes the size of data memory required to create a new buffer pool.

Parameters

- *int* [in] count - The number of buffers to have in the pool.
- *int* [in] size - The size of each buffer in the pool.
  Thread Safe: yes

Returns

UINT32 the memory size required

A.26.0.23 ixOsBuffPoolInit (IX_MBUF_POOL ** *poolPtrPtr*, int *count*, int *size*, char * *name*)

This function creates a new buffer pool.

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 initialises 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

Precondition

poolPtrPtr should point to a valid IX_MBUF_POOL pointer

Postcondition

A pool will be initialised 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.26.0.24 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 initialises 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 params *poolBufPtr* and *poolDataPtr*

Postcondition

A pool will be initialised 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.26.0.25 ixOsBuffPoolMbufAreaSizeGet (int *count*)

This function calcultes the size of mbuf memory required to create a new buffer pool.

Parameters

int [in] count - The number of buffers to have in the pool.

Thread Safe: yes

Returns

UINT32 the memory size required

A.26.0.26 ixOsBuffPoolMbufInit (int *mbufSizeAligned*, int *dataSizeAligned*, IX_MBUF_POOL * *poolPtr*)

Allocate memory for mbuf and data and initialise mbuf header fields.

This function allocates memory for an individual mbuf contained in an mbuf wrapper, defined above. The purpose of the wrapper is to add extra fields to the mbuf header which are hidden from the user. These extra fields are intended for use only by IxOsBuffPoolMgt.

Parameters

- *int* [in] mbufSizeAligned - Size aligned mbuf.
- *int* [in] dataSizeAligned - Size aligned data.
- *IX_MBUF_POOL* *[in]poolPtr - Pointer to buffer pool.

Returns

IX_MBUF_POOL *[in]poolPtr - Pointer to MBuf

A.26.0.27 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 funtion can serve as a useful debugging aid.

Thread Safe: yes

Precondition

poolPtr should point to a valid IX_MBUF_POOL structure

Returns

- IX_SUCCESS if the operation was successful
- IX_FAIL if the pool statistics could not be printed.

A.26.0.28 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 initialised.

Thread Safe: yes

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.26.0.29 ixOsBuffPoolUninit (IX_MBUF_POOL * *pool*)

Unitialize buffer pool.

Parameters

IX_MBUF_POOL *[in]pool - pointer to buffer pool

Returns

- IX_SUCCESS if the operation was successful
- IX_FAIL if the operation was not successful

A.27 IXP425 OS Cache MMU (IxOsCacheMMU) API

IXP425 OS Cache MMU (IxOsCacheMMU) API. IXP425 OS Cache MMU (IxOsCacheMMU) APIIXP425 OS Cache MMU (IxOsCacheMMU) APIIXP425 OS Cache MMU (IxOsCacheMMU) APIThis service provides services to the access components and codelets to abstract out any cache coherency issues and mmu mappings.

A.27.0.1 Defines

- #define IX_ACC_CACHE_ENABLED
  This macro enable cached memory in Access layers.
- #define IX_ACC_DRV_DMA_MALLOC(size) ixOsServCacheDmaAlloc(size)
  Allocate memory for driver use, that will be shared between XScale and NPE's.
- #define IX_ACC_DRV_DMA_FREE(ptr, size) ixOsServCacheDmaFree((ptr),(size))
  Free memory allocated from IX_ACC_DRV_DMA_MALLOC.
- #define IX_MMU_VIRTUAL_TO_PHYSICAL_TRANSLATION(addr) (addr)

Return a virtual address for the provided physical address.

- #define IX_MMU_PHYSICAL_TO_VIRTUAL_TRANSLATION(addr) (addr)
  Return a physical address for the provided virtual address.
- #define IX_XSCALE_CACHE_LINE_SIZE (32)
  IX_XSCALE_CACHE_LINE_SIZE = size of cache line for both flush and invalidate.
- #define IX_ACC_DRAM_PHYS_OFFSET (0x00000000UL)
  PHYS_OFFSET = Physical DRAM offset..
- #define IX_ACC_DATA_CACHE_INVALIDATE(addr, size) cacheInvalidate(DATA_CACHE, addr, size)
  Invalidate a cache range.
- #define IX_ACC_DATA_CACHE_FLUSH(addr, size) cacheFlush(DATA_CACHE, addr, size)
  Flush a cache range to physical memory.

A.27.0.2 Functions

- void * ixOsServCacheDmaAlloc (UINT32 size)
  Allocate memory for driver use, that will be shared between XScale and NPE's.
- void ixOsServCacheDmaFree (void *ptr, UINT32 size)
  Free memory allocated from ixOsServCacheDmaAlloc.

A.27.0.3 Detailed Description

This service provides services to the access components and codelets to abstract out any cache coherency issues and mmu mappings.

A.27.0.4 Define Documentation

A.27.0.5 #define IX_ACC_CACHE_ENABLED

This macro enable cached memory in Access layers.

When defined, this macro enable the use of cached memory in the access layers.

To disable cache on mbufs, #undef the macro

Definition at line 84 of file IxOsCacheMMU.h.

A.27.0.6 #define IX_ACC_DATA_CACHE_FLUSH(addr, size) cacheFlush(DATA_CACHE, addr, size)

Flush a cache range to physical memory.

Flush a cache range to physical memory.

Note: This is typically done prior to submitting a buffer to the NPE's which you expect the NPE to read from. Entire Cache lines will be flushed.

- If memory space used is non cached, then this function does may be null.
- Functionality required:
  - Non-Cached space : Flush CPU WB, No cache Flush.
  - Write Through Enabled : Flush CPU WB, No cache Flush.
  - Copy Back Enabled : Flush CPU WB, Invalidate area specified

Note: There are different implementations for this macro:

Linux OS implementation:

- #define IX_ACC_DATA_CACHE_FLUSH(addr,size) clean_dcache_range((__u32)addr, (__u32)addr + size)

VxWorks OS implementation:

- #define IX_ACC_DATA_CACHE_FLUSH(addr,size) cacheFlush(DATA_CACHE, addr, size)

default implementation:

- #define IX_ACC_DATA_CACHE_FLUSH(addr,size)
  Definition at line 361 of file IxOsCacheMMU.h.

A.27.0.7 #define IX_ACC_DATA_CACHE_INVALIDATE(addr, size) cacheInvalidate(DATA_CACHE, addr, size)

Invalidate a cache range.

Invalidate a cache range.

Note: This is typically done prior to submitting a buffer to the NPE's which you expect the NPE to populate with data.

Note: The size argument must be a multiple of cacheline size, i.e. a multiple if 32bytes for the XSCALE. The argument shall be rounded up to the next 32byte boundry. Extreem care must be taken when invalidating cache lines due.

- If memory space used is non cached, then this function may be null.
- Functionality required:
  - 1.Non-Cached space : No functionality required.
  - 2.Write Through Enabled : Invalidate area specified
  - 3.Copy Back Enabled : Invalidate area specified

Note: There are different implementations for this macro

Linux OS implementation:

#define IX_ACC_DATA_CACHE_INVALIDATE(addr,size)
invalidate_dcache_range((__u32)addr, (__u32)addr + size)

VxWorks OS implementation:

#define IX_ACC_DATA_CACHE_INVALIDATE(addr,size)
cacheInvalidate(DATA_CACHE, addr, size)

default implementation:

#define IX_ACC_DATA_CACHE_INVALIDATE(addr,size)

Definition at line 360 of file IxOsCacheMMU.h.

A.27.0.8 #define IX_ACC_DRAM_PHYS_OFFSET (0x00000000UL)

PHYS_OFFSET = Physical DRAM offset..

Note: There are different implementations for this macro

Linux OS implementation:

#define IX_ACC_DRAM_PHYS_OFFSET (PHYS_OFFSET)

VxWorks OS implementation:

#define IX_ACC_DRAM_PHYS_OFFSET (0x00000000UL)

Definition at line 346 of file IxOsCacheMMU.h.

A.27.0.9 #define IX_ACC_DRV_DMA_FREE(ptr, size) ixOsServCacheDmaFree((ptr),(size))

Free memory allocated from IX_ACC_DRV_DMA_MALLOC.

This function frees the memory allocated from *IX_ACC_DRV_DMA_MALLOC* .

Parameters

- *void* * ptr - pointer to the memory area to be freed.
- *UINT32* size - number of bytes of memory allocated.

Returns

void

See also:

IX_ACC_DRV_DMA_MALLOC

Definition at line 169 of file IxOsCacheMMU.h.

A.27.0.10 #define IX_ACC_DRV_DMA_MALLOC(size) ixOsServCacheDmaAlloc(size)

Allocate memory for driver use, that will be shared between XScale and NPE's.

This macro is used allocate memory for driver use, that will be shared between XScale and NPEs.

Note: The buffer allocated with have the system defined atributes, and as such the Invalidate and flush macros functionality must be updated. The buffer allocated is aligned on a cache line boundary.

Parameters

UINT32 size - number of bytes of memory requested.

Returns

void * Pointer to memory that can be used between XScale and NPE's.

See also:
IX_ACC_DRV_DMA_FREE

Definition at line 150 of file IxOsCacheMMU.h.

A.27.0.11 #define IX_MMU_PHYSICAL_TO_VIRTUAL_TRANSLATION(addr) (addr)

Return a physical address for the provided virtual address.

This macro return a physical address for the provided virtual address

Note: There are different implementations for this macro

Linux OS implementation:
#define IX_MMU_VIRTUAL_TO_PHYSICAL_TRANSLATION(addr) ((addr) ? virt_to_phys((void*)(addr)) : 0)

VxWorks OS implementation:
#define IX_MMU_PHYSICAL_TO_VIRTUAL_TRANSLATION(addr) (addr)

default implementation:
#define IX_MMU_PHYSICAL_TO_VIRTUAL_TRANSLATION(addr) (addr)

Definition at line 359 of file IxOsCacheMMU.h.

A.27.0.12 #define IX_MMU_VIRTUAL_TO_PHYSICAL_TRANSLATION(addr) (addr)

Return a virtual address for the provided physical address.

This macro return a virtual address for the provided physical address.

Note: There are different implementations for this macro

Linux OS implementation:
#define IX_MMU_VIRTUAL_TO_PHYSICAL_TRANSLATION(addr) ((addr) ? virt_to_phys((void*)(addr)) : 0)

VxWorks OS implementation:
#define IX_MMU_VIRTUAL_TO_PHYSICAL_TRANSLATION(addr) (addr)

default implementation:
#define IX_MMU_VIRTUAL_TO_PHYSICAL_TRANSLATION(addr) (addr)

Definition at line 358 of file IxOsCacheMMU.h.

A.27.0.13 #define IX_XSCALE_CACHE_LINE_SIZE (32)

IX_XSCALE_CACHE_LINE_SIZE = size of cache line for both flush and invalidate.

Definition at line 363 of file IxOsCacheMMU.h.

A.27.0.14 Function Documentation

A.27.0.15 void* ixOsServCacheDmaAlloc (UINT32 *size*)

Allocate memory for driver use, that will be shared between XScale and NPE's.

Allocate memory for driver use, that will be shared between XScale and NPE's.

Notes:

- The buffer allocated with have the system defined atributes, and as such the Invalidate and flush macros functionality must be updated.
- The buffer allocated is aligned on a cache line boundary.

Parameters

UINT32 size - number of bytes of memory requested.

Returns

void * Pointer to memory that can be used between XScale and NPE's.

See also:

ixOsServCacheDmaFree

A.27.0.16 void ixOsServCacheDmaFree (void * *ptr*, UINT32 *size*)

Free memory allocated from ixOsServCacheDmaAlloc.

This function frees the memory allocated from *ixOsServCacheDmaAlloc* .

Parameters

- *void* *ptr - pointer to the memory area to be freed.
- *UINT32* size - number of bytes of memory allocated.

Returns

void

See also:

ixOsServCacheDmaMalloc

A.28 IXP425 OS Services (IxOsServices) API

IXP425 OS Services (IxOsServices) API. IXP425 OS Services (IxOsServices) APIIXP425 OS Services (IxOsServices) APIIXP425 OS Services (IxOsServices) APIThis service provides a very thin layer of OS dependency services.

A.28.0.1 Defines

- #define IX_OSSERV_QMGR_MAP_SIZE (0x4000)
  Queue Manager map size.
- #define IX_OSSERV_EXP_REG_MAP_SIZE (0x1000)
  Exp Bus Registers map size.
- #define IX_OSSERV_UART1_MAP_SIZE (0x1000)
  UART1 map size.
- #define IX_OSSERV_UART2_MAP_SIZE (0x1000)
  UART2 map size.
- #define IX_OSSERV_PMU_MAP_SIZE (0x1000)
  PMU map size.
- #define IX_OSSERV_OSTS_MAP_SIZE (0x1000)
  OS Timers map size.
- #define IX_OSSERV_NPEA_MAP_SIZE (0x1000)
  NPE A map size.
- #define IX_OSSERV_NPEB_MAP_SIZE (0x1000)
  NPE B map size.
- #define IX_OSSERV_NPEC_MAP_SIZE (0x1000)
  NPE C map size.
- #define IX_OSSERV_ETHA_MAP_SIZE (0x1000)
  Eth A map size.
- #define IX_OSSERV_ETHB_MAP_SIZE (0x1000)
  Eth B map size.
- #define IX_OSSERV_USB_MAP_SIZE (0x1000)
  USB map size.
- #define IX_OSSERV_GPIO_MAP_SIZE (0x1000)
  GPIO map size.
- #define IX_OSSERV_EXP_BUS_MAP_SIZE (0x08000000)
  Expansion bus map size.
- #define IX_OSSERV_EXP_BUS_CS1_MAP_SIZE (0x01000000)
  CS1 map size.
- #define IX_OSSERV_EXP_BUS_CS4_MAP_SIZE (0x01000000)
  CS4 map size.

- #define IX_OSSERV_MEM_MAP(requestedPhysicalAddress, size) ixOsServMemMap(requestedPhysicalAddress, size, IX_COMPONENT_COHERENCY)
  Maps an I/O memory zone.
- #define IX_OSSERV_MEM_UNMAP(requestedVirtualAddress) ixOsServMemUnmap(requestedVirtualAddress, IX_COMPONENT_COHERENCY)
  unmaps a previously mapped I/O memory zone
- #define IX_OSSERV_MMAP_VIRT_TO_PHYS_TRANSLATION(virtualAddress) ixOsServMemVirtToPhys(virtualAddress, IX_COMPONENT_COHERENCY)
  provides a mapped memory-aware virtual to physical translation

A.28.0.2 Typedefs

- typedef int IX_IRQ_STATUS
  Defines flag to indicate IRQ status.
- typedef pthread_mutex_t IxMutex
  Mutex object.
- typedef pthread_mutex_t IxFastMutex
  Fast mutex object.

A.28.0.3 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.28.0.4 Functions

- PUBLIC IX_STATUS ixOsServIntBind (int level, void(*routine)(void *), void *parameter)
  binds a routine to a hardware interrupt
- PUBLIC IX_STATUS ixOsServIntUnbind (int level)
  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
- PUBLIC void ixOsServUnload (void)
  Used to un-map memory.

A.28.0.5 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 - XScale core clock granularity

*

```
 * OsServices I/O Memory Allocation and Access Routines
 *
 * List of OSes and operating modes supported:
 * (Please update the list when adding a new OS or mode)
 *
 * VxWorks BE
 * VxWorks LE
 * Linux BE
 *
 *
 * USAGE INFORMATION:
 *
 * This file defines OS/Endianess mode, memory mapped I/O access macros,
 * NPE-shared memory routines, SDRAM coherency mode, default component
 * coherency mode and static/dynamic memory mapping for every component.
 * The symbols defined below can be used and some overriden in the component-
specific
 * section of IxOsServicesComponents.h.
 *
 * OS/Endianess defines:
 *
 *   One of the following symbols will be defined:
 *   IX_OSSERV_VXWORKS_BE - component is compiled for VxWorks Big Endian
 *   IX_OSSERV_VXWORKS_LE - component is compiled for VxWorks Little Endian
 *   IX_OSSERV_LINUX_BE   - component is compiled for Linux Big Endian
 *
 * SDRAM coherency mode:
 *
 *   One of the following symbols will be defined:
 *   IX_SDRAM_BE                - SDRAM is in Big Endian mode (default for Big
Endian builds)
 *   IX_SDRAM_LE_ADDRESS_COHERENT - SDRAM is in Little Endian, Address Coherent
Mode (not supported by current software)
 *   IX_SDRAM_LE_DATA_COHERENT    - SDRAM is in Little Endian, Data Coherent
Mode (default for Little Endian builds)
 *
 * Static/dynamic memory mapping:
 *
 *   IX_STATIC_MEMORY_MAP  - component uses statically I/O mapped memory
(default)
 *   IX_DYNAMIC_MEMORY_MAP - component uses OS-specific dynamically mapped I/O
memory (define this in the component
 *                           specific section of IxOsServicesComponents.h to
override the previous default value)
 *
 * Component coherency mode (define or override defaults in the component
section of IxOsServicesComponents.h):
 *
 *   CSR_BE_MAPPING                 - component uses I/O memory in Big Endian
mode (default in Big Endian Builds)
 *   CSR_LE_ADDRESS_COHERENT_MAPPING - component uses I/O memory in Little
Endian, Address Coherent mode
 *   CSR_LE_DATA_COHERENT_MAPPING    - component uses I/O memory in Little
Endian, Data Coherent mode
 *   CSR_NO_MAPPING                 - component does not use I/O memory (I/O
read/write macros are not available)
 *
 * Macros for memory mapped I/O access:
 *
```

```
 *  Unless CSR_NO_MAPPING is defined, each component will have access to the
following set of macros for reading and
 *  writing word (32 bit), short (16 bit) and byte (8 bits) data. The macros
will perform all the necessary endianess conversions
 *  and use appropriate read and write functions in OSes where this is required.
 *
 *  The addresses (wAddr, sAddr, bAddr) should be volatile 32-bit pointers to
UINT32, UINT16 and UINT8 respectively.
 *  The data (wData, sData, bData) should be UINT32, UINT16 and respectively
UINT8 values.
 *
 *   IX_OSSERV_READ_LONG(wAddr)             - returns the 32-bit value at
address wAddr
 *   IX_OSSERV_READ_SHORT(sAddr)            - returns the 16-bit value at
address sAddr
 *   IX_OSSERV_READ_BYTE(bAddr)            - returns the 8-but value at address
bAddr
 *   IX_OSSERV_WRITE_LONG(wAddr, wData)      - writes the 32-bit wData at
address wAddr
 *   IX_OSSERV_WRITE_SHORT(sAddr, sData)    - writes the 16-bit sData at address
sAddr
 *   IX_OSSERV_WRITE_BYTE(bAddr, bData)     - writes the 8-bit bData at address
bAddr
 *
 * Macros for sharing data with the NPEs:
 *
 *  Each component will have access to the following set of macros for reading
and writing word (32 bit) and short
 *  (16 bit) data, plus a macro for copying within SDRAM an array of words to be
shared with an NPE. The macros will
 *  perform all the necessary endianess conversions depending on the SDRAM
endianess and coherency mode to guarantee
 *  correct sharing of data between XScale components and NPEs.
 *
 *  The addresses (wAddr, sAddr, wSrcAddr and wDestAddr) should be 32-bit
pointers to UINT32 and UINT16 respectively (volatility is NOT required).
 *  The data (wData and sData) should be UINT32 and respectively UINT16 values.
 *  The word count for the copy macro (wCount) should be a UINT32 value.
 *
 *   IX_OSSERV_READ_NPE_SHARED_LONG(wAddr)                              - returns
the 32-bit value written by the NPE at address wAddr
 *   IX_OSSERV_READ_NPE_SHARED_SHORT(sAddr)                             - returns
the 16-bit value written by the NPE at address sAddr
 *   IX_OSSERV_WRITE_NPE_SHARED_LONG(wAddr, wData)                       - writes
32-bit wData to be read by the NPE at address wAddr
 *   IX_OSSERV_WRITE_NPE_SHARED_SHORT(sAddr, sData)                      - writes
16-bit sData to be read by the NPE at address sAddr
 *   IX_OSSERV_COPY_NPE_SHARED_LONG_ARRAY(wDestAddr, wSrcAddr, wCount)  -
copies wCount 32-bit words shared with an NPE from wSrcAddr to wDestAddr
 *   IX_OSSERV_SWAP_NPE_SHARED_LONG(wData)                              - returns
the correctly converted (if necessary) 32-bit value between NPE and SDRAM
representation
 *
*
```

A.28.0.6 Define Documentation

A.28.0.7 #define IX_OSSERV_MEM_MAP(requestedPhysicalAddress, size) ixOsServMemMap(requestedPhysicalAddress, size, IX_COMPONENT_COHERENCY)

Maps an I/O memory zone.

Parameters

- *requestedAddress* UINT32 (in) - physical address to map
- *size* UINT32 (in) - size of map (should be large enough to hold the largest offset access made in this zone)

  This macro maps an I/O mapped physical memory zone of the given size into a virtual memory zone accessible by the caller and returns a cookie - the start address of the virtual memory zone. IX_MMU_PHYS_TO_VIRT_TRANSLATION should NOT therefore be used on the returned virtual address. The memory zone should be unmapped using IX_OSSERV_MEM_UNMAP once the caller has finished using this zone (e.g. on driver unload) using the cookie as parameter. The IX_OSSERV_READ/WRITE_LONG/SHORT macros should be used to read and write the mapped memory, adding the necessary offsets to the address cookie.

Warning: Not to be called from interrupt level.

Note: The size parameter is only used for identifying a suitable (i.e. large enough) map in the global memory map (ixOsServGlobalMemoryMap). It is NOT used to actually map the memory zone. Instead, the zone indicated in the global memory map is used. Mapping will work only if the zone is predefined in the global memory map.

Returns

the mapped virtual address or NULL if the operation could not be completed. This virtual address (cookie) has to be saved and used to unmap the memory zone during any clean-up or unload operation, using the IX_OSSERV_MEM_UNMAP macro.

Definition at line 202 of file IxOsServicesMemMap.h.

A.28.0.8 #define IX_OSSERV_MEM_UNMAP(requestedVirtualAddress) ixOsServMemUnmap(requestedVirtualAddress, IX_COMPONENT_COHERENCY)

unmaps a previously mapped I/O memory zone

Parameters

requestedAddress UINT32 (in) - cookie (virtual address) to unmap

This macro unmaps a previously mapped I/O memory zone using the cookie obtained in the mapping operation. The memory zone in question becomes unavailable to the caller once unmapped and the cookie should be discarded.

This macro cannot fail if the given parameter is correct and does not return a value.

Warning: Not to be called from interrupt level

Definition at line 224 of file IxOsServicesMemMap.h.

A.28.0.9 #define IX_OSSERV_MMAP_VIRT_TO_PHYS_TRANSLATION(virtualAddress) ixOsServMemVirtToPhys(virtualAddress, IX_COMPONENT_COHERENCY)

provides a mapped memory-aware virtual to physical translation

Parameters

virtualAddress UINT32 (in) - cookie (virtual address) to translate

This macro searches through the global memory mapped for a virtualAddress match; if found, it will return the physical address defined in the map. Otherwise it will default to IX_MMU_VIRTUAL_TO_PHYSICAL_TRANSLATION.

Returns

the physical address translation

Definition at line 240 of file IxOsServicesMemMap.h.

A.28.0.10 Enumeration Type Documentation

A.28.0.11 enum IxOsServTraceLevels

Trace levels.

Enumeration Values

- ***LOG_NONE*** No trace level.
- ***LOG_USER*** Set trace level to user.
- ***LOG_FATAL*** Set trace level to fatal.
- ***LOG_ERROR*** Set trace level to error.
- ***LOG_WARNING*** Set trace level to warning.
- ***LOG_MESSAGE*** Set trace level to message.
- ***LOG_DEBUG1*** Set trace level to debug1.
- ***LOG_DEBUG2*** Set trace level to debug2.
- ***LOG_DEBUG3*** Set trace level to debug3.
- ***LOG_ALL*** Set trace level to all.

Definition at line 140 of file IxOsServices.h.

A.28.0.12 Function Documentation

A.28.0.13 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.28.0.14 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.28.0.15 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.28.0.16 PUBLIC IX_STATUS ixOsServIntBind (int *level*, void(* *routine*)(void *), void * *parameter*)

binds a routine to a hardware interrupt

Parameters

- *level* int (in) - interrupt level 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 level. 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.28.0.17 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.28.0.18 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.28.0.19 PUBLIC IX_STATUS ixOsServIntUnbind (int *level*)

unbinds a routine from a hardware interrupt

Parameters

vector int (in) - interrupt level to unbind from

This function unbinds from an interrupt level 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.28.0.20 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.28.0.21 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.28.0.22 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.28.0.23 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.28.0.24 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.28.0.25 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.28.0.26 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.28.0.27 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, except for the first invocation

A.28.0.28 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.28.0.29 PUBLIC int ixOsServTimestampGet (void)

used to retrieve the system timestamp

Can be called from interrupt level: yes, except for the first invocation

Returns

the current timestamp value

A.28.0.30 PUBLIC void ixOsServUnload (void)

Used to un-map memory.

Parameters

None

Returns

None

A.29 IXP425 Performance Profiling (IxPerfProfAcc) API

IXP425 Performance Profiling (IxPerfProfAcc) API. IXP425 Performance Profiling (IxPerfProfAcc) APIIXP425 Performance Profiling (IxPerfProfAcc) APIIXP425 Performance Profiling (IxPerfProfAcc) APIIXP425 Performance Profiling Utility component Public API.

A.29.0.1 Data Structures

- struct IxPerfProfAccBusPmuResults
  Results obtained from running the Bus Pmu component. The results are obtained when the get functions is called.
- struct IxPerfProfAccXcycleResults
  Results obtained from Xcycle run.
- struct IxPerfProfAccXscalePmuEvtCnt
  contains results of a counter
- struct IxPerfProfAccXscalePmuResults
  contains results of counters and their overflow
- struct IxPerfProfAccXscalePmuSamplePcProfile
  contains summary of samples taken

A.29.0.2 Defines

- #define IX_PERFPROF_ACC_XSCALE_PMU_MAX_PROFILE_SAMPLES 0xFFFF

This is the maximum number of profiling samples allowed, which can be modified according to the user's discretion.

- #define IX_PERFPROF_ACC_BUS_PMU_MAX_PECS 7

  This is the maximum number of Programmable Event Counters available. This is a hardware specific and fixed value. Do not change.

- #define IX_PERFPROF_ACC_XCYCLE_MAX_NUM_OF_MEASUREMENTS 600

  Max number of measurement allowed. This constant is used when creating storage array for Xcycle. When run in continuous mode, Xcycle will wrap around and re-use buffer.

A.29.0.3 Enumerations

- enum IxPerfProfAccBusPmuEventCounters1 {
  IX_PERFPROF_ACC_BUS_PMU_PEC1_NORTH_NPEA_GRANT_SELECT = 1,
  IX_PERFPROF_ACC_BUS_PMU_PEC1_NORTH_NPEB_GRANT_SELECT,
  IX_PERFPROF_ACC_BUS_PMU_PEC1_NORTH_NPEC_GRANT_SELECT,
  IX_PERFPROF_ACC_BUS_PMU_PEC1_NORTH_BUS_IDLE_SELECT,
  IX_PERFPROF_ACC_BUS_PMU_PEC1_NORTH_NPEA_REQ_SELECT,
  IX_PERFPROF_ACC_BUS_PMU_PEC1_NORTH_NPEB_REQ_SELECT,
  IX_PERFPROF_ACC_BUS_PMU_PEC1_NORTH_NPEC_REQ_SELECT,
  IX_PERFPROF_ACC_BUS_PMU_PEC1_SOUTH_GSKT_GRANT_SELECT,
  IX_PERFPROF_ACC_BUS_PMU_PEC1_SOUTH_ABB_GRANT_SELECT,
  IX_PERFPROF_ACC_BUS_PMU_PEC1_SOUTH_PCI_GRANT_SELECT,
  IX_PERFPROF_ACC_BUS_PMU_PEC1_SOUTH_APB_GRANT_SELECT,
  IX_PERFPROF_ACC_BUS_PMU_PEC1_SOUTH_GSKT_REQ_SELECT,
  IX_PERFPROF_ACC_BUS_PMU_PEC1_SOUTH_ABB_REQ_SELECT,
  IX_PERFPROF_ACC_BUS_PMU_PEC1_SOUTH_PCI_REQ_SELECT,
  IX_PERFPROF_ACC_BUS_PMU_PEC1_SOUTH_APB_REQ_SELECT,
  IX_PERFPROF_ACC_BUS_PMU_PEC1_SDR_0_HIT_SELECT,
  IX_PERFPROF_ACC_BUS_PMU_PEC1_SDR_1_HIT_SELECT,
  IX_PERFPROF_ACC_BUS_PMU_PEC1_SDR_2_HIT_SELECT,
  IX_PERFPROF_ACC_BUS_PMU_PEC1_SDR_3_HIT_SELECT,
  IX_PERFPROF_ACC_BUS_PMU_PEC1_SDR_4_MISS_SELECT,
  IX_PERFPROF_ACC_BUS_PMU_PEC1_SDR_5_MISS_SELECT,
  IX_PERFPROF_ACC_BUS_PMU_PEC1_SDR_6_MISS_SELECT,
  IX_PERFPROF_ACC_BUS_PMU_PEC1_SDR_7_MISS_SELECT }

  Type of bus pmu events supported on PEC 1.

- enum IxPerfProfAccBusPmuEventCounters2 {
  IX_PERFPROF_ACC_BUS_PMU_PEC2_NORTH_NPEA_XFER_SELECT = 24,
  IX_PERFPROF_ACC_BUS_PMU_PEC2_NORTH_NPEB_XFER_SELECT,
  IX_PERFPROF_ACC_BUS_PMU_PEC2_NORTH_NPEC_XFER_SELECT,
  IX_PERFPROF_ACC_BUS_PMU_PEC2_NORTH_BUS_WRITE_SELECT,
  IX_PERFPROF_ACC_BUS_PMU_PEC2_NORTH_NPEA_OWN_SELECT,
  IX_PERFPROF_ACC_BUS_PMU_PEC2_NORTH_NPEB_OWN_SELECT,
  IX_PERFPROF_ACC_BUS_PMU_PEC2_NORTH_NPEC_OWN_SELECT,
  IX_PERFPROF_ACC_BUS_PMU_PEC2_SOUTH_GSKT_XFER_SELECT,
  IX_PERFPROF_ACC_BUS_PMU_PEC2_SOUTH_ABB_XFER_SELECT,
  IX_PERFPROF_ACC_BUS_PMU_PEC2_SOUTH_PCI_XFER_SELECT,
  IX_PERFPROF_ACC_BUS_PMU_PEC2_SOUTH_APB_XFER_SELECT,
  IX_PERFPROF_ACC_BUS_PMU_PEC2_SOUTH_GSKT_OWN_SELECT,
  IX_PERFPROF_ACC_BUS_PMU_PEC2_SOUTH_ABB_OWN_SELECT,
  IX_PERFPROF_ACC_BUS_PMU_PEC2_SOUTH_PCI_OWN_SELECT,
  IX_PERFPROF_ACC_BUS_PMU_PEC2_SOUTH_APB_OWN_SELECT,

IX_PERFPROF_ACC_BUS_PMU_PEC2_SDR_1_HIT_SELECT,
IX_PERFPROF_ACC_BUS_PMU_PEC2_SDR_2_HIT_SELECT,
IX_PERFPROF_ACC_BUS_PMU_PEC2_SDR_3_HIT_SELECT,
IX_PERFPROF_ACC_BUS_PMU_PEC2_SDR_4_HIT_SELECT,
IX_PERFPROF_ACC_BUS_PMU_PEC2_SDR_5_MISS_SELECT,
IX_PERFPROF_ACC_BUS_PMU_PEC2_SDR_6_MISS_SELECT,
IX_PERFPROF_ACC_BUS_PMU_PEC2_SDR_7_MISS_SELECT,
IX_PERFPROF_ACC_BUS_PMU_PEC2_SDR_0_MISS_SELECT }

Type of bus pmu events supported on PEC 2.

- enum IxPerfProfAccBusPmuEventCounters3 {
IX_PERFPROF_ACC_BUS_PMU_PEC3_NORTH_NPEA_RETRY_SELECT = 47,
IX_PERFPROF_ACC_BUS_PMU_PEC3_NORTH_NPEB_RETRY_SELECT,
IX_PERFPROF_ACC_BUS_PMU_PEC3_NORTH_NPEC_RETRY_SELECT,
IX_PERFPROF_ACC_BUS_PMU_PEC3_NORTH_BUS_READ_SELECT,
IX_PERFPROF_ACC_BUS_PMU_PEC3_NORTH_NPEA_WRITE_SELECT,
IX_PERFPROF_ACC_BUS_PMU_PEC3_NORTH_NPEB_WRITE_SELECT,
IX_PERFPROF_ACC_BUS_PMU_PEC3_NORTH_NPEC_WRITE_SELECT,
IX_PERFPROF_ACC_BUS_PMU_PEC3_SOUTH_GSKT_RETRY_SELECT,
IX_PERFPROF_ACC_BUS_PMU_PEC3_SOUTH_ABB_RETRY_SELECT,
IX_PERFPROF_ACC_BUS_PMU_PEC3_SOUTH_PCI_RETRY_SELECT,
IX_PERFPROF_ACC_BUS_PMU_PEC3_SOUTH_APB_RETRY_SELECT,
IX_PERFPROF_ACC_BUS_PMU_PEC3_SOUTH_GSKT_WRITE_SELECT,
IX_PERFPROF_ACC_BUS_PMU_PEC3_SOUTH_ABB_WRITE_SELECT,
IX_PERFPROF_ACC_BUS_PMU_PEC3_SOUTH_PCI_WRITE_SELECT,
IX_PERFPROF_ACC_BUS_PMU_PEC3_SOUTH_APB_WRITE_SELECT,
IX_PERFPROF_ACC_BUS_PMU_PEC3_SDR_2_HIT_SELECT,
IX_PERFPROF_ACC_BUS_PMU_PEC3_SDR_3_HIT_SELECT,
IX_PERFPROF_ACC_BUS_PMU_PEC3_SDR_4_HIT_SELECT,
IX_PERFPROF_ACC_BUS_PMU_PEC3_SDR_5_HIT_SELECT,
IX_PERFPROF_ACC_BUS_PMU_PEC3_SDR_6_MISS_SELECT,
IX_PERFPROF_ACC_BUS_PMU_PEC3_SDR_7_MISS_SELECT,
IX_PERFPROF_ACC_BUS_PMU_PEC3_SDR_0_MISS_SELECT,
IX_PERFPROF_ACC_BUS_PMU_PEC3_SDR_1_MISS_SELECT }

Type of bus pmu events supported on PEC 3.

- enum IxPerfProfAccBusPmuEventCounters4 {
IX_PERFPROF_ACC_BUS_PMU_PEC4_SOUTH_PCI_SPLIT_SELECT = 70,
IX_PERFPROF_ACC_BUS_PMU_PEC4_SOUTH_EXP_SPLIT_SELECT,
IX_PERFPROF_ACC_BUS_PMU_PEC4_SOUTH_APB_GRANT_SELECT,
IX_PERFPROF_ACC_BUS_PMU_PEC4_SOUTH_APB_XFER_SELECT,
IX_PERFPROF_ACC_BUS_PMU_PEC4_SOUTH_GSKT_READ_SELECT,
IX_PERFPROF_ACC_BUS_PMU_PEC4_SOUTH_ABB_READ_SELECT,
IX_PERFPROF_ACC_BUS_PMU_PEC4_SOUTH_PCI_READ_SELECT,
IX_PERFPROF_ACC_BUS_PMU_PEC4_SOUTH_APB_READ_SELECT,
IX_PERFPROF_ACC_BUS_PMU_PEC4_NORTH_ABB_SPLIT_SELECT,
IX_PERFPROF_ACC_BUS_PMU_PEC4_NORTH_NPEA_REQ_SELECT,
IX_PERFPROF_ACC_BUS_PMU_PEC4_NORTH_NPEA_READ_SELECT,
IX_PERFPROF_ACC_BUS_PMU_PEC4_NORTH_NPEB_READ_SELECT,
IX_PERFPROF_ACC_BUS_PMU_PEC4_NORTH_NPEC_READ_SELECT,
IX_PERFPROF_ACC_BUS_PMU_PEC4_SDR_3_HIT_SELECT,
IX_PERFPROF_ACC_BUS_PMU_PEC4_SDR_4_HIT_SELECT,
IX_PERFPROF_ACC_BUS_PMU_PEC4_SDR_5_HIT_SELECT,
IX_PERFPROF_ACC_BUS_PMU_PEC4_SDR_6_HIT_SELECT,
IX_PERFPROF_ACC_BUS_PMU_PEC4_SDR_7_MISS_SELECT,

IX_PERFPROF_ACC_BUS_PMU_PEC4_SDR_0_MISS_SELECT,
IX_PERFPROF_ACC_BUS_PMU_PEC4_SDR_1_MISS_SELECT,
IX_PERFPROF_ACC_BUS_PMU_PEC4_SDR_2_MISS_SELECT }

Type of bus pmu events supported on PEC 4.

- enum IxPerfProfAccBusPmuEventCounters5 {
IX_PERFPROF_ACC_BUS_PMU_PEC5_SOUTH_ABB_GRANT_SELECT = 91,
IX_PERFPROF_ACC_BUS_PMU_PEC5_SOUTH_ABB_XFER_SELECT,
IX_PERFPROF_ACC_BUS_PMU_PEC5_SOUTH_ABB_RETRY_SELECT,
IX_PERFPROF_ACC_BUS_PMU_PEC5_SOUTH_EXP_SPLIT_SELECT,
IX_PERFPROF_ACC_BUS_PMU_PEC5_SOUTH_ABB_REQ_SELECT,
IX_PERFPROF_ACC_BUS_PMU_PEC5_SOUTH_ABB_OWN_SELECT,
IX_PERFPROF_ACC_BUS_PMU_PEC5_SOUTH_BUS_IDLE_SELECT,
IX_PERFPROF_ACC_BUS_PMU_PEC5_NORTH_NPEB_GRANT_SELECT,
IX_PERFPROF_ACC_BUS_PMU_PEC5_NORTH_NPEB_XFER_SELECT,
IX_PERFPROF_ACC_BUS_PMU_PEC5_NORTH_NPEB_RETRY_SELECT,
IX_PERFPROF_ACC_BUS_PMU_PEC5_NORTH_NPEB_REQ_SELECT,
IX_PERFPROF_ACC_BUS_PMU_PEC5_NORTH_NPEB_OWN_SELECT,
IX_PERFPROF_ACC_BUS_PMU_PEC5_NORTH_NPEB_WRITE_SELECT,
IX_PERFPROF_ACC_BUS_PMU_PEC5_NORTH_NPEB_READ_SELECT,
IX_PERFPROF_ACC_BUS_PMU_PEC5_SDR_4_HIT_SELECT,
IX_PERFPROF_ACC_BUS_PMU_PEC5_SDR_5_HIT_SELECT,
IX_PERFPROF_ACC_BUS_PMU_PEC5_SDR_6_HIT_SELECT,
IX_PERFPROF_ACC_BUS_PMU_PEC5_SDR_7_HIT_SELECT,
IX_PERFPROF_ACC_BUS_PMU_PEC5_SDR_0_MISS_SELECT,
IX_PERFPROF_ACC_BUS_PMU_PEC5_SDR_1_MISS_SELECT,
IX_PERFPROF_ACC_BUS_PMU_PEC5_SDR_2_MISS_SELECT,
IX_PERFPROF_ACC_BUS_PMU_PEC5_SDR_3_MISS_SELECT }

Type of bus pmu events supported on PEC 5.

- enum IxPerfProfAccBusPmuEventCounters6 {
IX_PERFPROF_ACC_BUS_PMU_PEC6_SOUTH_PCI_GRANT_SELECT = 113,
IX_PERFPROF_ACC_BUS_PMU_PEC6_SOUTH_PCI_XFER_SELECT,
IX_PERFPROF_ACC_BUS_PMU_PEC6_SOUTH_PCI_RETRY_SELECT,
IX_PERFPROF_ACC_BUS_PMU_PEC6_SOUTH_PCI_SPLIT_SELECT,
IX_PERFPROF_ACC_BUS_PMU_PEC6_SOUTH_PCI_REQ_SELECT,
IX_PERFPROF_ACC_BUS_PMU_PEC6_SOUTH_PCI_OWN_SELECT,
IX_PERFPROF_ACC_BUS_PMU_PEC6_SOUTH_BUS_WRITE_SELECT,
IX_PERFPROF_ACC_BUS_PMU_PEC6_NORTH_NPEC_GRANT_SELECT,
IX_PERFPROF_ACC_BUS_PMU_PEC6_NORTH_NPEC_XFER_SELECT,
IX_PERFPROF_ACC_BUS_PMU_PEC6_NORTH_NPEC_RETRY_SELECT,
IX_PERFPROF_ACC_BUS_PMU_PEC6_NORTH_NPEC_REQ_SELECT,
IX_PERFPROF_ACC_BUS_PMU_PEC6_NORTH_NPEC_OWN_SELECT,
IX_PERFPROF_ACC_BUS_PMU_PEC6_NORTH_NPEB_WRITE_SELECT,
IX_PERFPROF_ACC_BUS_PMU_PEC6_NORTH_NPEC_READ_SELECT,
IX_PERFPROF_ACC_BUS_PMU_PEC6_SDR_5_HIT_SELECT,
IX_PERFPROF_ACC_BUS_PMU_PEC6_SDR_6_HIT_SELECT,
IX_PERFPROF_ACC_BUS_PMU_PEC6_SDR_7_HIT_SELECT,
IX_PERFPROF_ACC_BUS_PMU_PEC6_SDR_0_HIT_SELECT,
IX_PERFPROF_ACC_BUS_PMU_PEC6_SDR_1_MISS_SELECT,
IX_PERFPROF_ACC_BUS_PMU_PEC6_SDR_2_MISS_SELECT,
IX_PERFPROF_ACC_BUS_PMU_PEC6_SDR_3_MISS_SELECT,
IX_PERFPROF_ACC_BUS_PMU_PEC6_SDR_4_MISS_SELECT }

Type of bus pmu events supported on PEC 6.

- enum IxPerfProfAccBusPmuEventCounters7 {
  IX_PERFPROF_ACC_BUS_PMU_PEC7_SOUTH_APB_RETRY_SELECT = 135,
  IX_PERFPROF_ACC_BUS_PMU_PEC7_SOUTH_APB_REQ_SELECT,
  IX_PERFPROF_ACC_BUS_PMU_PEC7_SOUTH_APB_OWN_SELECT,
  IX_PERFPROF_ACC_BUS_PMU_PEC7_SOUTH_BUS_READ_SELECT,
  IX_PERFPROF_ACC_BUS_PMU_PEC7_CYCLE_COUNT_SELECT }

  Type of bus pmu events supported on PEC 7.

- enum IxPerfProfAccXscalePmuEvent {
  IX_PERFPROF_ACC_XSCALE_PMU_EVENT_CACHE_MISS = 0,
  IX_PERFPROF_ACC_XSCALE_PMU_EVENT_CACHE_INSTRUCTION,
  IX_PERFPROF_ACC_XSCALE_PMU_EVENT_STALL,
  IX_PERFPROF_ACC_XSCALE_PMU_EVENT_INST_TLB_MISS,
  IX_PERFPROF_ACC_XSCALE_PMU_EVENT_DATA_TLB_MISS,
  IX_PERFPROF_ACC_XSCALE_PMU_EVENT_BRANCH_EXEC,
  IX_PERFPROF_ACC_XSCALE_PMU_EVENT_BRANCH_MISPREDICT,
  IX_PERFPROF_ACC_XSCALE_PMU_EVENT_INST_EXEC,
  IX_PERFPROF_ACC_XSCALE_PMU_EVENT_FULL_EVERYCYCLE,
  IX_PERFPROF_ACC_XSCALE_PMU_EVENT_ONCE,
  IX_PERFPROF_ACC_XSCALE_PMU_EVENT_DATA_CACHE_ACCESS,
  IX_PERFPROF_ACC_XSCALE_PMU_EVENT_DATA_CACHE_MISS,
  IX_PERFPROF_ACC_XSCALE_PMU_EVENT_DATA_CACHE_WRITEBACK,
  IX_PERFPROF_ACC_XSCALE_PMU_EVENT_SW_CHANGE_PC,
  IX_PERFPROF_ACC_XSCALE_PMU_EVENT_MAX }

  Type of xscale pmu events supported.

- enum IxPerfProfAccStatus { IX_PERFPROF_ACC_STATUS_SUCCESS = IX_SUCCESS,
  IX_PERFPROF_ACC_STATUS_FAIL = IX_FAIL,
  IX_PERFPROF_ACC_STATUS_ANOTHER_UTIL_IN_PROGRESS,
  IX_PERFPROF_ACC_STATUS_XCYCLE_MEASUREMENT_IN_PROGRESS,
  IX_PERFPROF_ACC_STATUS_XCYCLE_NO_BASELINE,
  IX_PERFPROF_ACC_STATUS_XCYCLE_MEASUREMENT_REQUEST_OUT_OF_RANGE, IX_PERFPROF_ACC_STATUS_XCYCLE_PRIORITY_SET_FAIL,
  IX_PERFPROF_ACC_STATUS_XCYCLE_THREAD_CREATE_FAIL,
  IX_PERFPROF_ACC_STATUS_XCYCLE_PRIORITY_RESTORE_FAIL,
  IX_PERFPROF_ACC_STATUS_XCYCLE_MEASUREMENT_NOT_RUNNING,
  IX_PERFPROF_ACC_STATUS_XSCALE_PMU_NUM_INVALID,
  IX_PERFPROF_ACC_STATUS_XSCALE_PMU_EVENT_INVALID,
  IX_PERFPROF_ACC_STATUS_XSCALE_PMU_START_NOT_CALLED,
  IX_PERFPROF_ACC_STATUS_BUS_PMU_MODE_ERROR,
  IX_PERFPROF_ACC_STATUS_BUS_PMU_PEC1_ERROR,
  IX_PERFPROF_ACC_STATUS_BUS_PMU_PEC2_ERROR,
  IX_PERFPROF_ACC_STATUS_BUS_PMU_PEC3_ERROR,
  IX_PERFPROF_ACC_STATUS_BUS_PMU_PEC4_ERROR,
  IX_PERFPROF_ACC_STATUS_BUS_PMU_PEC5_ERROR,
  IX_PERFPROF_ACC_STATUS_BUS_PMU_PEC6_ERROR,
  IX_PERFPROF_ACC_STATUS_BUS_PMU_PEC7_ERROR,
  IX_PERFPROF_ACC_STATUS_BUS_PMU_START_NOT_CALLED }

  Invalid Status Definitions.

- enum IxPerfProfAccBusPmuMode { IX_PERFPROF_ACC_BUS_PMU_MODE_HALT = 0,
  IX_PERFPROF_ACC_BUS_PMU_MODE_SOUTH,
  IX_PERFPROF_ACC_BUS_PMU_MODE_NORTH,
  IX_PERFPROF_ACC_BUS_PMU_MODE_SDRAM }

  State selection of counters.

A.29.0.4 Functions

- PUBLIC IxPerfProfAccStatus ixPerfProfAccXscalePmuEventCountStart (BOOL clkCntDiv, UINT32 numEvents, IxPerfProfAccXscalePmuEvent pmuEvent1, IxPerfProfAccXscalePmuEvent pmuEvent2, IxPerfProfAccXscalePmuEvent pmuEvent3, IxPerfProfAccXscalePmuEvent pmuEvent4)

  This API will start the clock and event counting.

- PUBLIC IxPerfProfAccStatus ixPerfProfAccXscalePmuEventCountStop (IxPerfProfAccXscalePmuResults *eventCountStopResults)

  This API will stop the clock and event counting.

- PUBLIC IxPerfProfAccStatus ixPerfProfAccXscalePmuTimeSampStart (UINT32 samplingRate, BOOL clkCntDiv)

  Starts the time based sampling.

- PUBLIC IxPerfProfAccStatus ixPerfProfAccXscalePmuTimeSampStop (IxPerfProfAccXscalePmuEvtCnt *clkCount, IxPerfProfAccXscalePmuSamplePcProfile *timeProfile)

  Stops the time based sampling.

- PUBLIC IxPerfProfAccStatus ixPerfProfAccXscalePmuEventSampStart (UINT32 numEvents, IxPerfProfAccXscalePmuEvent pmuEvent1, UINT32 eventRate1, IxPerfProfAccXscalePmuEvent pmuEvent2, UINT32 eventRate2, IxPerfProfAccXscalePmuEvent pmuEvent3, UINT32 eventRate3, IxPerfProfAccXscalePmuEvent pmuEvent4, UINT32 eventRate4)

  Starts the event based sampling.

- PUBLIC IxPerfProfAccStatus ixPerfProfAccXscalePmuEventSampStop (IxPerfProfAccXscalePmuSamplePcProfile *eventProfile1, IxPerfProfAccXscalePmuSamplePcProfile *eventProfile2, IxPerfProfAccXscalePmuSamplePcProfile *eventProfile3, IxPerfProfAccXscalePmuSamplePcProfile *eventProfile4)

  Stops the event based sampling.

- PUBLIC void ixPerfProfAccXscalePmuResultsGet (IxPerfProfAccXscalePmuResults *results)

  Reads the current value of the counters and their overflow.

- PUBLIC IxPerfProfAccStatus ixPerfProfAccBusPmuStart (IxPerfProfAccBusPmuMode mode, IxPerfProfAccBusPmuEventCounters1 pecEvent1, IxPerfProfAccBusPmuEventCounters2 pecEvent2, IxPerfProfAccBusPmuEventCounters3 pecEvent3, IxPerfProfAccBusPmuEventCounters4 pecEvent4, IxPerfProfAccBusPmuEventCounters5 pecEvent5, IxPerfProfAccBusPmuEventCounters6 pecEvent6, IxPerfProfAccBusPmuEventCounters7 pecEvent7)

  Initializes all the counters and selects events to be monitored.

- PUBLIC IxPerfProfAccStatus ixPerfProfAccBusPmuStop (void)

  Stops all counters.

- PUBLIC void ixPerfProfAccBusPmuResultsGet (IxPerfProfAccBusPmuResults *BusPmuResults)

  Gets values of all counters.

- PUBLIC void ixPerfProfAccBusPmuPMSRGet (UINT32 *pmsrValue)

  Get values of PMSR.

- PUBLIC IxPerfProfAccStatus ixPerfProfAccXcycleBaselineRun (UINT32 *numBaselineCycle)

  Perform baseline for Xcycle.

- PUBLIC IxPerfProfAccStatus ixPerfProfAccXcycleStart (UINT32 numMeasurementsRequested)

  Start the measurement.

- PUBLIC IxPerfProfAccStatus ixPerfProfAccXcycleStop (void)

  Stop the Xcycle measurement.

- PUBLIC IxPerfProfAccStatus ixPerfProfAccXcycleResultsGet (IxPerfProfAccXcycleResults *xcycleResult)

  Get the results of Xcycle measurement.

- PUBLIC BOOL ixPerfProfAccXcycleInProgress (void)

  Check if Xcycle is running.

A.29.0.5 Detailed Description

IXP425 Performance Profiling Utility component Public API.

A.29.0.6 Define Documentation

A.29.0.7 #define IX_PERFPROF_ACC_BUS_PMU_MAX_PECS 7

This is the maximum number of Programmable Event Counters available. This is a hardware specific and fixed value. Do not change.

Definition at line 76 of file IxPerfProfAcc.h.

A.29.0.8 #define IX_PERFPROF_ACC_XCYCLE_MAX_NUM_OF_MEASUREMENTS 600

Max number of measurement allowed. This constant is used when creating storage array for Xcycle. When run in continuous mode, Xcycle will wrap around and re-use buffer.

Definition at line 87 of file IxPerfProfAcc.h.

A.29.0.9 #define IX_PERFPROF_ACC_XSCALE_PMU_MAX_PROFILE_SAMPLES 0xFFFF

This is the maximum number of profiling samples allowed, which can be modified according to the user's discretion.

Definition at line 65 of file IxPerfProfAcc.h.

A.29.0.10 Enumeration Type Documentation

A.29.0.11 enum IxPerfProfAccBusPmuEventCounters1

Type of bus pmu events supported on PEC 1.

Lists all bus pmu events.

Enumeration Values

- ***IX_PERFPROF_ACC_BUS_PMU_PEC1_NORTH_NPEA_GRANT_SELECT*** Select North NPEA grant on PEC1.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC1_NORTH_NPEB_GRANT_SELECT*** Select North NPEB grant on PEC1.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC1_NORTH_NPEC_GRANT_SELECT*** Select North NPEC grant on PEC1.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC1_NORTH_BUS_IDLE_SELECT*** Select North bus idle on PEC1.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC1_NORTH_NPEA_REQ_SELECT*** Select North NPEA req on PEC1.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC1_NORTH_NPEB_REQ_SELECT*** Select North NPEB req on PEC1.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC1_NORTH_NPEC_REQ_SELECT*** Select North NPEC req on PEC1.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC1_SOUTH_GSKT_GRANT_SELECT*** Select south gasket grant on PEC1.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC1_SOUTH_ABB_GRANT_SELECT*** Select south abb grant on PEC1.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC1_SOUTH_PCI_GRANT_SELECT*** Select south pci grant on PEC1.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC1_SOUTH_APB_GRANT_SELECT*** Select south apb grant on PEC1.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC1_SOUTH_GSKT_REQ_SELECT*** Select south gasket request on PEC1.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC1_SOUTH_ABB_REQ_SELECT*** Select south abb request on PEC1.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC1_SOUTH_PCI_REQ_SELECT*** Select south pci request on PEC1.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC1_SOUTH_APB_REQ_SELECT*** Select south apb request on PEC1.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC1_SDR_0_HIT_SELECT*** Select sdram0 hit on PEC1.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC1_SDR_1_HIT_SELECT*** Select sdram1 hit on PEC1.

- ***IX_PERFPROF_ACC_BUS_PMU_PEC1_SDR_2_HIT_SELECT*** Select sdram2 hit on PEC1.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC1_SDR_3_HIT_SELECT*** Select sdram3 hit on PEC1.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC1_SDR_4_MISS_SELECT*** Select sdram4 miss on PEC1.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC1_SDR_5_MISS_SELECT*** Select sdram5 miss on PEC1.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC1_SDR_6_MISS_SELECT*** Select sdram6 miss on PEC1.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC1_SDR_7_MISS_SELECT*** Select sdram7 miss on PEC1.

Definition at line 174 of file IxPerfProfAcc.h.

A.29.0.12 enum IxPerfProfAccBusPmuEventCounters2

Type of bus pmu events supported on PEC 2.

Lists all bus pmu events.

Enumeration Values

- ***IX_PERFPROF_ACC_BUS_PMU_PEC2_NORTH_NPEA_XFER_SELECT*** Select North NPEA transfer on PEC2.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC2_NORTH_NPEB_XFER_SELECT*** Select North NPEB transfer on PEC2.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC2_NORTH_NPEC_XFER_SELECT*** Select North NPEC transfer on PEC2.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC2_NORTH_BUS_WRITE_SELECT*** Select North bus write on PEC2.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC2_NORTH_NPEA_OWN_SELECT*** Select North NPEA own on PEC2.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC2_NORTH_NPEB_OWN_SELECT*** Select North NPEB own on PEC2.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC2_NORTH_NPEC_OWN_SELECT*** Select North NPEC own on PEC2.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC2_SOUTH_GSKT_XFER_SELECT*** Select South gasket transfer on PEC2.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC2_SOUTH_ABB_XFER_SELECT*** Select South abb transfer on PEC2.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC2_SOUTH_PCI_XFER_SELECT*** Select South pci transfer on PEC2.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC2_SOUTH_APB_XFER_SELECT*** Select South apb transfer on PEC2.

- ***IX_PERFPROF_ACC_BUS_PMU_PEC2_SOUTH_GSKT_OWN_SELECT*** Select South gasket own on PEC2.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC2_SOUTH_ABB_OWN_SELECT*** Select South abb own on PEC2.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC2_SOUTH_PCI_OWN_SELECT*** Select South pci own on PEC2.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC2_SOUTH_APB_OWN_SELECT*** Select South apb own transfer on PEC2.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC2_SDR_1_HIT_SELECT*** Select sdram1 hit on PEC2.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC2_SDR_2_HIT_SELECT*** Select sdram2 hit on PEC2.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC2_SDR_3_HIT_SELECT*** Select sdram3 hit on PEC2.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC2_SDR_4_HIT_SELECT*** Select sdram4 hit on PEC2.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC2_SDR_5_MISS_SELECT*** Select sdram5 miss on PEC2.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC2_SDR_6_MISS_SELECT*** Select sdram6 miss on PEC2.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC2_SDR_7_MISS_SELECT*** Select sdram7 miss on PEC2.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC2_SDR_0_MISS_SELECT*** Select sdram0 miss on PEC2.

  Definition at line 212 of file IxPerfProfAcc.h.

A.29.0.13 enum IxPerfProfAccBusPmuEventCounters3

Type of bus pmu events supported on PEC 3.

Lists all bus pmu events.

Enumeration Values

- ***IX_PERFPROF_ACC_BUS_PMU_PEC3_NORTH_NPEA_RETRY_SELECT*** Select north NPEA retry on PEC3.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC3_NORTH_NPEB_RETRY_SELECT*** Select north NPEB retry on PEC3.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC3_NORTH_NPEC_RETRY_SELECT*** Select north NPEC retry on PEC3.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC3_NORTH_BUS_READ_SELECT*** Select north bus read on PEC3.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC3_NORTH_NPEA_WRITE_SELECT*** Select north NPEA write on PEC3.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC3_NORTH_NPEB_WRITE_SELECT*** Select north NPEB write on PEC3.

- ***IX_PERFPROF_ACC_BUS_PMU_PEC3_NORTH_NPEC_WRITE_SELECT*** Select north NPEC wirte on PEC3.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC3_SOUTH_GSKT_RETRY_SELECT*** Select south gasket retry on PEC3.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC3_SOUTH_ABB_RETRY_SELECT*** Select south abb retry on PEC3.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC3_SOUTH_PCI_RETRY_SELECT*** Select south pci retry on PEC3.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC3_SOUTH_APB_RETRY_SELECT*** Select south apb retry on PEC3.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC3_SOUTH_GSKT_WRITE_SELECT*** Select south gasket write on PEC3.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC3_SOUTH_ABB_WRITE_SELECT*** Select south abb write on PEC3.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC3_SOUTH_PCI_WRITE_SELECT*** Select south pci write on PEC3.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC3_SOUTH_APB_WRITE_SELECT*** Select south apb write on PEC3.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC3_SDR_2_HIT_SELECT*** Select sdram2 hit on PEC3.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC3_SDR_3_HIT_SELECT*** Select sdram3 hit on PEC3.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC3_SDR_4_HIT_SELECT*** Select sdram4 hit on PEC3.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC3_SDR_5_HIT_SELECT*** Select sdram5 hit on PEC3.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC3_SDR_6_MISS_SELECT*** Select sdram6 miss on PEC3.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC3_SDR_7_MISS_SELECT*** Select sdram7 miss on PEC3.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC3_SDR_0_MISS_SELECT*** Select sdram0 miss on PEC3.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC3_SDR_1_MISS_SELECT*** Select sdram1 miss on PEC3.

Definition at line 250 of file IxPerfProfAcc.h.

A.29.0.14 enum IxPerfProfAccBusPmuEventCounters4

Type of bus pmu events supported on PEC 4.

Lists all bus pmu events.

Enumeration Values

- ***IX_PERFPROF_ACC_BUS_PMU_PEC4_SOUTH_PCI_SPLIT_SELECT*** Select south pci split on PEC4.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC4_SOUTH_EXP_SPLIT_SELECT*** Select south expansion split on PEC4.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC4_SOUTH_APB_GRANT_SELECT*** Select south apb grant on PEC4.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC4_SOUTH_APB_XFER_SELECT*** Select south apb transfer on PEC4.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC4_SOUTH_GSKT_READ_SELECT*** Select south gasket read on PEC4.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC4_SOUTH_ABB_READ_SELECT*** Select south abb read on PEC4.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC4_SOUTH_PCI_READ_SELECT*** Select south pci read on PEC4.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC4_SOUTH_APB_READ_SELECT*** Select south apb read on PEC4.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC4_NORTH_ABB_SPLIT_SELECT*** Select north abb split on PEC4.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC4_NORTH_NPEA_REQ_SELECT*** Select north NPEA req on PEC4.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC4_NORTH_NPEA_READ_SELECT*** Select north NPEA read on PEC4.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC4_NORTH_NPEB_READ_SELECT*** Select north NPEB read on PEC4.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC4_NORTH_NPEC_READ_SELECT*** Select north NPEC read on PEC4.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC4_SDR_3_HIT_SELECT*** Select sdram3 hit on PEC4.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC4_SDR_4_HIT_SELECT*** Select sdram4 hit on PEC4.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC4_SDR_5_HIT_SELECT*** Select sdram5 hit on PEC4.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC4_SDR_6_HIT_SELECT*** Select sdram6 hit on PEC4.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC4_SDR_7_MISS_SELECT*** Select sdram7 miss on PEC4.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC4_SDR_0_MISS_SELECT*** Select sdram0 miss on PEC4.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC4_SDR_1_MISS_SELECT*** Select sdram1 miss on PEC4.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC4_SDR_2_MISS_SELECT*** Select sdram2 miss on PEC4.

Definition at line 288 of file IxPerfProfAcc.h.

A.29.0.15 enum IxPerfProfAccBusPmuEventCounters5

Type of bus pmu events supported on PEC 5.

Lists all bus pmu events.

Enumeration Values

- ***IX_PERFPROF_ACC_BUS_PMU_PEC5_SOUTH_ABB_GRANT_SELECT*** Select south abb grant on PEC5.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC5_SOUTH_ABB_XFER_SELECT*** Select south abb transfer on PEC5.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC5_SOUTH_ABB_RETRY_SELECT*** Select south abb retry on PEC5.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC5_SOUTH_EXP_SPLIT_SELECT*** Select south expansion split on PEC5.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC5_SOUTH_ABB_REQ_SELECT*** Select south abb request on PEC5.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC5_SOUTH_ABB_OWN_SELECT*** Select south abb own on PEC5.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC5_SOUTH_BUS_IDLE_SELECT*** Select south bus idle on PEC5.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC5_NORTH_NPEB_GRANT_SELECT*** Select north NPEB grant on PEC5.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC5_NORTH_NPEB_XFER_SELECT*** Select north NPEB transfer on PEC5.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC5_NORTH_NPEB_RETRY_SELECT*** Select north NPEB retry on PEC5.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC5_NORTH_NPEB_REQ_SELECT*** Select north NPEB request on PEC5.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC5_NORTH_NPEB_OWN_SELECT*** Select north NPEB own on PEC5.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC5_NORTH_NPEB_WRITE_SELECT*** Select north NPEB write on PEC5.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC5_NORTH_NPEB_READ_SELECT*** Select north NPEB read on PEC5.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC5_SDR_4_HIT_SELECT*** Select north sdram4 hit on PEC5.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC5_SDR_5_HIT_SELECT*** Select north sdram5 hit on PEC5.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC5_SDR_6_HIT_SELECT*** Select north sdram6 hit on PEC5.

- ***IX_PERFPROF_ACC_BUS_PMU_PEC5_SDR_7_HIT_SELECT*** Select north sdram7 hit on PEC5.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC5_SDR_0_MISS_SELECT*** Select north sdram0 miss on PEC5.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC5_SDR_1_MISS_SELECT*** Select north sdram1 miss on PEC5.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC5_SDR_2_MISS_SELECT*** Select north sdram2 miss on PEC5.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC5_SDR_3_MISS_SELECT*** Select north sdram3 miss on PEC5.

Definition at line 324 of file IxPerfProfAcc.h.

A.29.0.16 enum IxPerfProfAccBusPmuEventCounters6

Type of bus pmu events supported on PEC 6.

Lists all bus pmu events.

Enumeration Values

- ***IX_PERFPROF_ACC_BUS_PMU_PEC6_SOUTH_PCI_GRANT_SELECT*** Select south pci grant on PEC6.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC6_SOUTH_PCI_XFER_SELECT*** Select south pci transfer on PEC6.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC6_SOUTH_PCI_RETRY_SELECT*** Select south pci retry on PEC6.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC6_SOUTH_PCI_SPLIT_SELECT*** Select south pci split on PEC6.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC6_SOUTH_PCI_REQ_SELECT*** Select south pci request on PEC6.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC6_SOUTH_PCI_OWN_SELECT*** Select south pci own on PEC6.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC6_SOUTH_BUS_WRITE_SELECT*** Select south pci write on PEC6.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC6_NORTH_NPEC_GRANT_SELECT*** Select north NPEC grant on PEC6.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC6_NORTH_NPEC_XFER_SELECT*** Select north NPEC transfer on PEC6.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC6_NORTH_NPEC_RETRY_SELECT*** Select north NPEC retry on PEC6.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC6_NORTH_NPEC_REQ_SELECT*** Select north NPEC request on PEC6.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC6_NORTH_NPEC_OWN_SELECT*** Select north NPEC own on PEC6.

- ***IX_PERFPROF_ACC_BUS_PMU_PEC6_NORTH_NPEB_WRITE_SELECT*** Select north NPEB write on PEC6.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC6_NORTH_NPEC_READ_SELECT*** Select north NPEC read on PEC6.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC6_SDR_5_HIT_SELECT*** Select sdram5 hit on PEC6.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC6_SDR_6_HIT_SELECT*** Select sdram6 hit on PEC6.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC6_SDR_7_HIT_SELECT*** Select sdram7 hit on PEC6.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC6_SDR_0_HIT_SELECT*** Select sdram0 hit on PEC6.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC6_SDR_1_MISS_SELECT*** Select sdram1 miss on PEC6.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC6_SDR_2_MISS_SELECT*** Select sdram2 miss on PEC6.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC6_SDR_3_MISS_SELECT*** Select sdram3 miss on PEC6.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC6_SDR_4_MISS_SELECT*** Select sdram4 miss on PEC6.

Definition at line 361 of file IxPerfProfAcc.h.

A.29.0.17 enum IxPerfProfAccBusPmuEventCounters7

Type of bus pmu events supported on PEC 7.

Lists all bus pmu events.

Enumeration Values

- ***IX_PERFPROF_ACC_BUS_PMU_PEC7_SOUTH_APB_RETRY_SELECT*** Select south apb retry on PEC7.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC7_SOUTH_APB_REQ_SELECT*** Select south apb request on PEC7.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC7_SOUTH_APB_OWN_SELECT*** Select south apb own on PEC7.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC7_SOUTH_BUS_READ_SELECT*** Select south bus read on PEC7.
- ***IX_PERFPROF_ACC_BUS_PMU_PEC7_CYCLE_COUNT_SELECT*** Select cycle count on PEC7.

Definition at line 398 of file IxPerfProfAcc.h.

A.29.0.18 enum IxPerfProfAccBusPmuMode

State selection of counters.

These states will be used to determine the counters whose values are to be read.

Enumeration Values

- ***IX_PERFPROF_ACC_BUS_PMU_MODE_HALT*** halt state
- ***IX_PERFPROF_ACC_BUS_PMU_MODE_SOUTH*** south state
- ***IX_PERFPROF_ACC_BUS_PMU_MODE_NORTH*** north state
- ***IX_PERFPROF_ACC_BUS_PMU_MODE_SDRAM*** SDRAM state.
  Definition at line 519 of file IxPerfProfAcc.h.

A.29.0.19 enum IxPerfProfAccStatus

Invalid Status Definitions.

These status will be used by the APIs to return to the user.

Enumeration Values

- ***IX_PERFPROF_ACC_STATUS_SUCCESS*** success
- ***IX_PERFPROF_ACC_STATUS_FAIL*** fail
- ***IX_PERFPROF_ACC_STATUS_ANOTHER_UTIL_IN_PROGRESS*** another utility in progress
- ***IX_PERFPROF_ACC_STATUS_XCYCLE_MEASUREMENT_IN_PROGRESS*** measurement in progress
- ***IX_PERFPROF_ACC_STATUS_XCYCLE_NO_BASELINE*** no baseline yet
- ***IX_PERFPROF_ACC_STATUS_XCYCLE_MEASUREMENT_REQUEST_OUT_OF_RANGE*** Measurement chosen is out of range.
- ***IX_PERFPROF_ACC_STATUS_XCYCLE_PRIORITY_SET_FAIL*** Cannot set task priority.
- ***IX_PERFPROF_ACC_STATUS_XCYCLE_THREAD_CREATE_FAIL*** Fail create thread.
- ***IX_PERFPROF_ACC_STATUS_XCYCLE_PRIORITY_RESTORE_FAIL*** cannot restore priority
- ***IX_PERFPROF_ACC_STATUS_XCYCLE_MEASUREMENT_NOT_RUNNING*** xcycle not running
- ***IX_PERFPROF_ACC_STATUS_XSCALE_PMU_NUM_INVALID*** invalid number entered
- ***IX_PERFPROF_ACC_STATUS_XSCALE_PMU_EVENT_INVALID*** invalid pmu event
- ***IX_PERFPROF_ACC_STATUS_XSCALE_PMU_START_NOT_CALLED*** a start process was not called before attempting a stop or results get
- ***IX_PERFPROF_ACC_STATUS_BUS_PMU_MODE_ERROR*** invalid mode
- ***IX_PERFPROF_ACC_STATUS_BUS_PMU_PEC1_ERROR*** invalid pec1 entered
- ***IX_PERFPROF_ACC_STATUS_BUS_PMU_PEC2_ERROR*** invalid pec2 entered
- ***IX_PERFPROF_ACC_STATUS_BUS_PMU_PEC3_ERROR*** invalid pec3 entered
- ***IX_PERFPROF_ACC_STATUS_BUS_PMU_PEC4_ERROR*** invalid pec4 entered
- ***IX_PERFPROF_ACC_STATUS_BUS_PMU_PEC5_ERROR*** invalid pec5 entered

- ***IX_PERFPROF_ACC_STATUS_BUS_PMU_PEC6_ERROR*** invalid pec6 entered
- ***IX_PERFPROF_ACC_STATUS_BUS_PMU_PEC7_ERROR*** invalid pec7 entered
- ***IX_PERFPROF_ACC_STATUS_BUS_PMU_START_NOT_CALLED*** a start process was not called before attempting a stop

Definition at line 457 of file IxPerfProfAcc.h.

A.29.0.20 enum IxPerfProfAccXscalePmuEvent

Type of xscale pmu events supported.

Lists all xscale pmu events. The maximum is a default value that the user should not exceed.

Enumeration Values

- ***IX_PERFPROF_ACC_XSCALE_PMU_EVENT_CACHE_MISS*** cache miss
- ***IX_PERFPROF_ACC_XSCALE_PMU_EVENT_CACHE_INSTRUCTION*** cache instruction
- ***IX_PERFPROF_ACC_XSCALE_PMU_EVENT_STALL*** event stall
- ***IX_PERFPROF_ACC_XSCALE_PMU_EVENT_INST_TLB_MISS*** instruction tlb miss
- ***IX_PERFPROF_ACC_XSCALE_PMU_EVENT_DATA_TLB_MISS*** data tlb miss
- ***IX_PERFPROF_ACC_XSCALE_PMU_EVENT_BRANCH_EXEC*** branch executed
- ***IX_PERFPROF_ACC_XSCALE_PMU_EVENT_BRANCH_MISPREDICT*** branch mispredict
- ***IX_PERFPROF_ACC_XSCALE_PMU_EVENT_INST_EXEC*** instruction executed
- ***IX_PERFPROF_ACC_XSCALE_PMU_EVENT_FULL_EVERYCYCLE*** Stall - data cache buffers are full.
- This event occurs every cycle where condition present
- ***IX_PERFPROF_ACC_XSCALE_PMU_EVENT_ONCE*** Stall - data cache buffers are full.This event occurs once for each contiguous sequence.
- ***IX_PERFPROF_ACC_XSCALE_PMU_EVENT_DATA_CACHE_ACCESS*** data cache access
- ***IX_PERFPROF_ACC_XSCALE_PMU_EVENT_DATA_CACHE_MISS*** data cache miss
- ***IX_PERFPROF_ACC_XSCALE_PMU_EVENT_DATA_CACHE_WRITEBACK*** data cache writeback
- ***IX_PERFPROF_ACC_XSCALE_PMU_EVENT_SW_CHANGE_PC*** sw change pc
- ***IX_PERFPROF_ACC_XSCALE_PMU_EVENT_MAX*** max value

Definition at line 417 of file IxPerfProfAcc.h.

A.29.0.21 Function Documentation

A.29.0.22 ixPerfProfAccBusPmuPMSRGet (UINT32 * *pmsrValue*)

Get values of PMSR.

This API gets the Previous Master Slave Register value and returns it to the calling function. This value indicates which master or slave accessed the north, south bus or sdram last. The value returned by this function is a 32 bit value and is read from location of an offset 0x0024 of the base value.

The PMSR value returned indicate the following:

```
*****************************************************************************
******* *  Bit    *  Name  *      Description
* *
*
*****************************************************************************
******* * [31:18]  *Reserved*
*
*****************************************************************************
******* * [17:12] *  PSS   * Indicates which of the slaves on
* *         *        *  ARBS was previously
* *         *        * accessed by the AHBS.
*  *         *        * [000001] Expansion Bus
* *         *        * [000010] SDRAM Controller
* *         *        * [000100] PCI
* *         *        * [001000] Queue Manager
*  *         *        * [010000] AHB-APB Bridge
*  *         *        * [100000] Reserved
*
*****************************************************************************
******* * [11:8]  *  PSN   * Indicates which of the Slaves on
* *         *        * ARBN was previously
* *         *        * accessed the AHBN.
* *         *        * [0001] SDRAM Controller
* *         *        * [0010] AHB-AHB Bridge
* *         *        * [0100] Reserved
* *         *        * [1000] Reserved
*
*****************************************************************************
******* *  [7:4]  *  PMS   * Indicates which of the Masters on
* *         *        * ARBS was previously
* *         *        * accessing the AHBS.
* *         *        * [0001] Gasket
* *         *        * [0010] AHB-AHB Bridge
* *         *        * [0100] PCI
* *         *        * [1000] APB
*
*****************************************************************************
******* *  [3:0]  *  PMN   * Indicates which of the Masters on
* *         *        * ARBN was previously
* *         *        * accessing the AHBN.
* *         *        * [0001] NPEA
* *         *        * [0010] NPEB
* *         *        * [0100] NPEC
* *         *        * [1000] Reserved
*
*****************************************************************************
*******
```

Parameters

*UINT32 *pmsrValue* - Pointer to return PMSR value. Users need to allocate storage for psmrValue.

Returns

none

Reentrant : no

ISR Callable : no

A.29.0.23 ixPerfProfAccBusPmuResultsGet (IxPerfProfAccBusPmuResults * *BusPmuResults*)

Gets values of all counters.

This function is responsible for getting all the counter values from the lower API and putting it into an array for the user.

Parameters

- "IxPerfProfAccBusPmuResults *busPmuResults"

Pointer to a structure of arrays to store all counter values.

Returns

none

Reentrant : no

ISR Callable : no

A.29.0.24 ixPerfProfAccBusPmuStart (IxPerfProfAccBusPmuMode *mode*, IxPerfProfAccBusPmuEventCounters1 *pecEvent1*, IxPerfProfAccBusPmuEventCounters2 *pecEvent2*, IxPerfProfAccBusPmuEventCounters3 *pecEvent3*, IxPerfProfAccBusPmuEventCounters4 *pecEvent4*, IxPerfProfAccBusPmuEventCounters5 *pecEvent5*, IxPerfProfAccBusPmuEventCounters6 *pecEvent6*, IxPerfProfAccBusPmuEventCounters7 *pecEvent7*)

Initializes all the counters and selects events to be monitored.

Function initializes all the counters and assigns the events associated with the counters. Users send in the mode and events they want to count. This API verifies if the combination chosen is appropriate and sets all the registers accordingly. Selecting HALT mode will result in an error. User should use ixPerfProfAccBusPmuStop() to HALT.

Parameters

- *IxPerfProfAccStateBusPmuMode* mode - Mode selection.
- *IxPerfProfAccBusPmuEventCounters1* pecEvent1 - Event for PEC1.
- *IxPerfProfAccBusPmuEventCounters2* pecEvent2 - Event for PEC2.
- *IxPerfProfAccBusPmuEventCounters3* pecEvent3 - Event for PEC3.

- *IxPerfProfAccBusPmuEventCounters4* pecEvent4 - Event for PEC4.
- *IxPerfProfAccBusPmuEventCounters5* pecEvent5 - Event for PEC5.
- *IxPerfProfAccBusPmuEventCounters6* pecEvent6 - Event for PEC6.
- *IxPerfProfAccBusPmuEventCounters7* pecEvent7 - Event for PEC7.

Returns

- IX_PERFPROF_ACC_STATUS_SUCCESS - Initialization executed successfully.
- IX_PERFPROF_ACC_STATUS_BUS_PMU_MODE_ERROR - Error in selection of mode. Only NORTH, SOUTH and SDRAM modes are allowed.
- IX_PERFPROF_ACC_STATUS_BUS_PMU_PEC1_ERROR - Error in selection of event for PEC1
- IX_PERFPROF_ACC_STATUS_BUS_PMU_PEC2_ERROR - Error in selection of event for PEC2
- IX_PERFPROF_ACC_STATUS_BUS_PMU_PEC3_ERROR - Error in selection of event for PEC3
- IX_PERFPROF_ACC_STATUS_BUS_PMU_PEC4_ERROR - Error in selection of event for PEC4
- IX_PERFPROF_ACC_STATUS_BUS_PMU_PEC5_ERROR - Error in selection of event for PEC5
- IX_PERFPROF_ACC_STATUS_BUS_PMU_PEC6_ERROR - Error in selection of event for PEC6
- IX_PERFPROF_ACC_STATUS_BUS_PMU_PEC7_ERROR - Error in selection of event for PEC7
- IX_PERFPROF_ACC_STATUS_ANOTHER_UTIL_IN_PROGRESS - another utility is running
- IX_PERFPROF_ACC_STATUS_FAIL - Failed to start because interrupt service routine fails to bind.

Reentrant : no

ISR Callable : no

A.29.0.25 ixPerfProfAccBusPmuStop (void)

Stops all counters.

This function stops all the PECs by setting the halt bit in the ESR.

Returns

- IX_PERFPROF_ACC_STATUS_SUCCESS - Counters successfully halted.
- IX_PERFPROF_ACC_STATUS_FAIL - Counters could'nt be halted.
- IX_PERFPROF_ACC_STATUS_BUS_PMU_START_NOT_CALLED - the ixPerfProfAccBusPmuStart() function is not called.

Reentrant : no

ISR Callable : no

A.29.0.26 ixPerfProfAccXcycleBaselineRun (UINT32 * *numBaselineCycle*)

Perform baseline for Xcycle.

Parameters

*UINT32 [out] *numBaselineCycle* - pointer to baseline value after calibration. Calling function are responsible for allocating memory space for this pointer.

Global Data :

None.

This function MUST be run before the Xcycle tool can be used. This function must be run immediately when the OS boots up with no other addition programs running. Addition note : This API will measure the time needed to perform a fix amount of CPU instructions (~ 1 second worth of loops) as a highest priority task and with interrupt disabled. The time measured is known as the baseline - interpreted as the shortest time needed to complete the amount of CPU instructions. The baseline is returned as unit of time in 66Mhz clock tick.

Returns

- IX_PERFPROF_ACC_STATUS_SUCCESS - successful run, result is returned
- IX_PERFPROF_ACC_STATUS_XCYCLE_PRIORITY_SET_FAIL - failed to change task priority
- IX_PERFPROF_ACC_STATUS_XCYCLE_PRIORITY_RESTORE_FAIL - failed to restore task priority
- IX_PERFPROF_ACC_STATUS_ANOTHER_UTIL_IN_PROGRESS - another utility is running
- IX_PERFPROF_ACC_STATUS_XCYCLE_MEASUREMENT_IN_PROGRESS - Xcycle tool has already started

Reentrant : no

ISR Callable : no

A.29.0.27 ixPerfProfAccXcycleInProgress (void)

Check if Xcycle is running.

Parameters

None

Global Data: None.

Check if Xcycle measuring task is running.

Returns

- TRUE - Xcycle is running

- FALSE - Xcycle is not running

Reentrant : no

ISR Callable : no

A.29.0.28 ixPerfProfAccXcycleResultsGet (IxPerfProfAccXcycleResults * *xcycleResult*)

Get the results of Xcycle measurement.

Parameters

*IxPerfProfAccXcycleResults [out] *xcycleResult* Pointer to results of last measurements. Calling function are responsible for allocating memory space for this pointer.

Global Data :

None.

Retrieve the results of last measurement. User should use ixPerfProfAccXcycleInProgress() to check if measurement is completed before getting the results.

Returns

- IX_PERFPROF_ACC_STATUS_SUCCESS - successful
- IX_PERFPROF_ACC_STATUS_FAIL - result is not complete.
- IX_PERFPROF_ACC_STATUS_XCYCLE_NO_BASELINE - baseline is performed
- IX_PERFPROF_ACC_STATUS_XCYCLE_MEASUREMENT_IN_PROGRESS - Xcycle tool is still running

Reentrant : no

ISR Callable : no

A.29.0.29 ixPerfProfAccXcycleStart (UINT32 *numMeasurementsRequested*)

Start the measurement.

Parameters

UINT32 [in] numMeasurementsRequested - number of measurements to perform. Value can be 0 to IX_PERFPROF_ACC_XCYCLE_MAX_NUM_OF_MEASUREMENTS. 0 indicate continuous measurement.

Global Data :

None.

Start the measurements immediately. numMeasurementsRequested specifies number of measurements to run. If numMeasurementsRequested is set to 0, the measurement will be performed continuously until IxPerfProfAccXcycleStop() is called. It is estimated that 1 measurement takes approximately 1 second during low CPU utilization, therefore 128

measurement takes approximately 128 sec. When CPU utilization is high, the measurement will take longer. This function spawn a task the perform the measurement and returns. The measurement may continue even if this function returns.

Note: Under heavy CPU utilization, the task spawn by this function may starve and fail to respond to stop command. User may need to kill the task manually in this case.

There are only IX_PERFPROF_ACC_XCYCLE_MAX_NUM_OF_MEASUREMENTS storage available so storing is wrapped around if measurements are more than IX_PERFPROF_ACC_XCYCLE_MAX_NUM_OF_MEASUREMENTS.

Returns

- IX_PERFPROF_ACC_STATUS_SUCCESS - successful start, a thread is created in the background to perform measurement.
- IX_PERFPROF_ACC_STATUS_XCYCLE_PRIORITY_SET_FAIL - failed to set task priority
- IX_PERFPROF_ACC_STATUS_XCYCLE_THREAD_CREATE_FAIL - failed to create thread to perform measurement.
- IX_PERFPROF_ACC_STATUS_XCYCLE_NO_BASELINE - baseline is not available
- IX_PERFPROF_ACC_STATUS_XCYCLE_MEASUREMENT_REQUEST_OUT_OF_RANGE - value is larger than IX_PERFPROF_ACC_XCYCLE_MAX_NUM_OF_MEASUREMENTS
- IX_PERFPROF_ACC_STATUS_XCYCLE_MEASUREMENT_IN_PROGRESS - Xcycle tool has already started
- IX_PERFPROF_ACC_STATUS_ANOTHER_UTIL_IN_PROGRESS - another utility is running

Reentrant : no

ISR Callable : no

A.29.0.30 ixPerfProfAccXcycleStop (void)

Stop the Xcycle measurement.

Parameters

None

Global Data :None.

Stop Xcycle measurements immediately. If the measurements have stopped or not started, return IX_PERFPROF_STATUS_XCYCLE_MEASUREMENT_NOT_RUNNING.

Note: This function does not stop measurement cold. The measurement thread may need a few seconds to complete the last measurement. User needs to use ixPerfProfAccXcycleInProgress() to determine if measurement is indeed completed.

Returns

- IX_PERFPROF_ACC_STATUS_SUCCESS - successful measurement is stopped

- IX_PERFPROF_STATUS_XCYCLE_MEASUREMENT_NOT_RUNNING - no measurement running

Reentrant : no

ISR Callable : no

A.29.0.31 ixPerfProfAccXscalePmuEventCountStart (BOOL *clkCntDiv*, UINT32 *numEvents*, IxPerfProfAccXscalePmuEvent *pmuEvent1*, IxPerfProfAccXscalePmuEvent *pmuEvent2*, IxPerfProfAccXscalePmuEvent *pmuEvent3*, IxPerfProfAccXscalePmuEvent *pmuEvent4*)

This API will start the clock and event counting.

Parameters

- *BOOL [in] clkCntDiv* - enables/disables the clock divider. When true, the divider is enabled and the clock count will be incremented by one at each 64th processor clock cycle. When false, the divider is disabled and the clock count will be incremented at every processor clock cycle.
- *UINT32 [in] numEvents* - the number of PMU events that are to be monitored as specified by the user. For clock counting only, this is set to zero.
- *IxPerfProfAccXscalePmuEvent [in] pmuEvent1* - the specific PMU event to be monitored by counter 1
- *IxPerfProfAccXscalePmuEvent [in] pmuEvent2* - the specific PMU event to be monitored by counter 2
- *IxPerfProfAccXscalePmuEvent [in] pmuEvent3* - the specific PMU event to be monitored by counter 3
- *IxPerfProfAccXscalePmuEvent [in] pmuEvent4* - the specific PMU event to be monitored by counter 4

This API will start the clock and xscale PMU event counting. Up to 4 events can be monitored simultaneously. This API has to be called before ixPerfProfAccXscalePmuEventCountStop can be called.

Returns

- IX_PERFPROF_ACC_STATUS_SUCCESS if clock and events counting are started successfully
- IX_PERFPROF_ACC_STATUS_FAIL if unable to start the counting
- IX_PERFPROF_ACC_STATUS_XSCALE_PMU_NUM_INVALID if the number of events specified is out of the valid range
- IX_PERFPROF_ACC_STATUS_XSCALE_PMU_EVENT_INVALID if the value of the PMU event specified does not exist
- IX_PERFPROF_ACC_STATUS_ANOTHER_UTIL_IN_PROGRESS - another utility is running

Reentrant : no

ISR Callable : no

A.29.0.32 ixPerfProfAccXscalePmuEventCountStop (IxPerfProfAccXscalePmuResults * *eventCountStopResults*)

This API will stop the clock and event counting.

Parameters

*IxPerfProfAccXscalePmuResults [out] *eventCountStopResults* - pointer to struct containing results of counters and their overflow. It is the users's responsibility to allocate the memory for this pointer.

This API will stop the clock and xscale PMU events that are being counted. The results of the clock and events count will be stored in the pointer allocated by the user. It can only be called once IxPerfProfAccEventCountStart has been called.

Returns

- IX_PERFPROF_ACC_STATUS_SUCCESS if clock and events counting are stopped successfully
- IX_PERFPROF_ACC_STATUS_XSCALE_PMU_START_NOT_CALLED if ixPerfProfAccXscalePmuEventCountStart is not called first.

Reentrant : no

ISR Callable : no

A.29.0.33 ixPerfProfAccXscalePmuEventSampStart (UINT32 *numEvents*, IxPerfProfAccXscalePmuEvent *pmuEvent1*, UINT32 *eventRate1*, IxPerfProfAccXscalePmuEvent *pmuEvent2*, UINT32 *eventRate2*, IxPerfProfAccXscalePmuEvent *pmuEvent3*, UINT32 *eventRate3*, IxPerfProfAccXscalePmuEvent *pmuEvent4*, UINT32 *eventRate4*)

Starts the event based sampling.

Parameters

- *UINT32 [in] <numEvents>* - the number of PMU events that are to be monitored as specified by the user. The value should be between 1-4 events at a time.
- *IxPerfProfAccXscalePmuEvent [in] pmuEvent1* - the specific PMU event to be monitored by counter 1
- *UINT32 [in] eventRate1* - sampling rate of counter 1. The rate is the number of events before a sample taken. If 0 is specified, the the full counter value (0xFFFFFFFF) is used. The rate must not be greater than the full counter value.
- *IxPerfProfAccXscalePmuEvent [in] pmuEvent2* - the specific PMU event to be monitored by counter 2
- *UINT32 [in] eventRate2* - sampling rate of counter 2. The rate is the number of events before a sample taken. If 0 is specified, the full counter value (0xFFFFFFFF) is used. The rate must not be greater than the full counter value.

- *IxPerfProfAccXscalePmuEvent [in] pmuEvent3* - the specific PMU event to be monitored by counter 3
- *UINT32 [in] eventRate3* - sampling rate of counter 3. The rate is the number of events before a sample taken. If 0 is specified, the full counter value (0xFFFFFFFF) is used. The rate must not be greater than the full counter value.
- *IxPerfProfAccXscalePmuEvent [in] pmuEvent4* - the specific PMU event to be monitored by counter 4
- *UINT32 [in] eventRate4* - sampling rate of counter 4. The rate is the number of events before a sample taken. If 0 is specified, the full counter value (0xFFFFFFFF) is used. The rate must not be greater than the full counter value.

Starts the event based sampling to determine the frequency with which events are being executed. The sampling rate is the number of events, as specified by the user, before a counter overflow interrupt is generated. A sample is taken at each counter overflow interrupt. At each sample,the value of the program counter determines the corresponding location in the code. Each of these occurrences are recorded to determine the frequency with which the Xscale code in each event is executed. This API has to be called before ixPerfProfAccXscalePmuEventSampStop can be called.

Returns

- IX_PERFPROF_ACC_STATUS_SUCCESS if event based sampling is started successfully
- IX_PERFPROF_ACC_STATUS_FAIL if unable to start the sampling
- IX_PERFPROF_ACC_STATUS_XSCALE_PMU_NUM_INVALID if the number of events specified is out of the valid range
- IX_PERFPROF_ACC_STATUS_XSCALE_PMU_EVENT_INVALID if the value of the PMU event specified does not exist
- IX_PERFPROF_ACC_STATUS_ANOTHER_UTIL_IN_PROGRESS - another utility is running

Reentrant : no

ISR Callable : no

A.29.0.34 ixPerfProfAccXscalePmuEventSampStop (IxPerfProfAccXscalePmuSamplePcProfile * *eventProfile1*, IxPerfProfAccXscalePmuSamplePcProfile * *eventProfile2*, IxPerfProfAccXscalePmuSamplePcProfile * *eventProfile3*, IxPerfProfAccXscalePmuSamplePcProfile * *eventProfile4*)

Stops the event based sampling.

Parameters

- *IxPerfProfAccXscalePmuSamplePcProfile [out] *eventProfile1* - pointer to the array of profiles for each program counter value; the user should set the size of the array to IX_PERFPROF_ACC_XSCALE_PMU_MAX_PROFILE_SAMPLES. It is the users's responsibility to allocate memory for this pointer.
- *IxPerfProfAccXscalePmuSamplePcProfile [out] *eventProfile2* - pointer to the array of profiles for each program counter value; the user should set the size of the array to

IX_PERFPROF_ACC_XSCALE_PMU_MAX_PROFILE_SAMPLES. It is the users's responsibility to allocate memory for this pointer.

- *IxPerfProfAccXscalePmuSamplePcProfile [out] *eventProfile3* - pointer to the array of profiles for each program counter value; the user should set the size of the array to IX_PERFPROF_ACC_XSCALE_PMU_MAX_PROFILE_SAMPLES. It is the users's responsibility to allocate memory for this pointer.
- *IxPerfProfAccXscalePmuSamplePcProfile [out] *eventProfile4* - pointer to the array of profiles for each program counter value; the user should set the size of the array to IX_PERFPROF_ACC_XSCALE_PMU_MAX_PROFILE_SAMPLES. It is the users's responsibility to allocate memory for this pointer.

This API stops the event based sampling. The results are stored in the pointers allocated by the user. It can only be called once ixPerfProfAccEventSampStart has been called.

Returns

- IX_PERFPROF_ACC_STATUS_SUCCESS if event based sampling is stopped successfully
- IX_PERFPROF_ACC_STATUS_XSCALE_PMU_START_NOT_CALLED if ixPerfProfAccEventSampStart not called first.

Reentrant : no

ISR Callable : no

A.29.0.35 ixPerfProfAccXscalePmuResultsGet (IxPerfProfAccXscalePmuResults * *results*)

Reads the current value of the counters and their overflow.

Parameters

*IxPerfProfAccXscalePmuResults [out] *results* - pointer to the results struct. It is the user's responsibility to allocate memory for this pointer

This API reads the value of all four event counters and the clock counter, and the associated overflows. It does not give results associated with sampling, i.e. PC and their frequencies. This API can be called at any time once a process has been started. If it is called before a process has started the user should be aware that the values it contains are default values and might be meaningless. The values of the counters are stored in the pointer allocated by the client.

Returns

None

Reentrant : no

ISR Callable : no

A.29.0.36 ixPerfProfAccXscalePmuTimeSampStart (UINT32 *samplingRate*, BOOL *clkCntDiv*)

Starts the time based sampling.

Parameters

- *UINT32 [in] samplingRate* - sampling rate is the number of clock counts before a counter overflow interrupt is generated, at which, a sample is taken; the rate specified cannot be greater than the counter size of 32bits or set to zero.
- *BOOL [in] clkCntDiv* - enables/disables the clock divider. When true, the divider is enabled and the clock count will be incremented by one at each 64th processor clock cycle. When false, the divider is disabled and the clock count will be incremented at every processor clock cycle.

This API starts the time based sampling to determine the frequency with which lines of code are being executed. Sampling is done at the rate specified by the user. At each sample,the value of the program counter is determined. Each of these occurrences are recorded to determine the frequency with which the Xscale code is being executed. This API has to be called before ixPerfProfAccXscalePmuTimeSampStop can be called.

Returns

- IX_PERFPROF_ACC_STATUS_SUCCESS if time based sampling is started successfully
- IX_PERFPROF_ACC_STATUS_FAIL if unable to start the sampling
- IX_PERFPROF_ACC_STATUS_ANOTHER_UTIL_IN_PROGRESS - another utility is running

Reentrant : no

ISR Callable : no

A.29.0.37 ixPerfProfAccXscalePmuTimeSampStop (IxPerfProfAccXscalePmuEvtCnt * *clkCount*, IxPerfProfAccXscalePmuSamplePcProfile * *timeProfile*)

Stops the time based sampling.

Parameters

- *IxPerfProfAccXscalePmuEvtCnt [out] *clkCount* - pointer to the struct containing the final clock count and its overflow. It is the user's responsibility to allocate the memory for this pointer.
- *IxPerfProfAccXscalePmuSamplePcProfile [out] *timeProfile* - pointer to the array of profiles for each program counter value; the user should set the size of the array to IX_PERFPROF_ACC_XSCALE_PMU_MAX_PROFILE_SAMPLES. It is the user's responsibility to allocate the memory for this pointer.

  This API stops the time based sampling. The results are stored in the pointers allocated by the user. It can only be called once ixPerfProfAccXscalePmuTimeSampStart has been called.

Returns

- IX_PERFPROF_ACC_STATUS_SUCCESS if time based sampling is stopped successfully
- IX_PERFPROF_ACC_STATUS_XSCALE_PMU_START_NOT_CALLED if ixPerfProfAccXscalePmuTimeSampStart not called first

Reentrant : no

ISR Callable : no

A.30 IXP425 Queue Manager (IxQMgr) API

IXP425 Queue Manager (IxQMgr) API. IXP425 Queue Manager (IxQMgr) APIIXP425 Queue Manager (IxQMgr) APIIXP425 Queue Manager (IxQMgr) APIThe public API for the IXP425 QMgr component.

A.30.0.1 Data Structures

- struct IxQMgrQInlinedReadWriteInfo
  Internal structure to facilitate inlining functions in IxQMgr.h.

A.30.0.2 Defines

- #define IX_QMGR_SAVED_COMPONENT_NAME IX_COMPONENT_NAME
  Because this file contains inline functions which will be compiled into other components, we need to ensure that the IX_COMPONENT_NAME define is set to ix_qmgr while this code is being compiled.
- #define **IX_COMPONENT_NAME** ix_qmgr
- #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_AVAILABLE_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.
- #define ixQMgrDispatcherLoopRun ixQMgrDispatcherLoopRunA0
  Map old function name ixQMgrDispatcherLoopRun() to ixQMgrDispatcherLoopRunA0().

A.30.0.3 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.
- typedef void(* IxQMgrDispatcherFuncPtr)(IxQMgrDispatchGroup group)
  QMgr Dispatcher Loop function pointer.

A.30.0.4 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.30.0.5 Functions

- PUBLIC IX_STATUS ixQMgrInit (void)
  Initialise the QMgr.
- PUBLIC IX_STATUS ixQMgrUnload (void)
  Uninitialise the QMgr.
- PUBLIC void ixQMgrShow (void)
  Describe queue configuration and statistics for active queues.
- PUBLIC IX_STATUS ixQMgrQShow (IxQMgrQId qId)
  Display aqueue 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 fo 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_STATUS ixQMgrQReadMWordsMinus1 (IxQMgrQId qId, UINT32 *entry)
  This function reads the remaining of the q entry for queues configured with many words. (the first word of the entry is already read in the inlined function and the entry pointer already incremented.
- IX_QMGR_INLINE PUBLIC IX_STATUS ixQMgrQRead (IxQMgrQId qId, UINT32 *entryPtr)
  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 ixQMgrDispatcherLoopRunA0 (IxQMgrDispatchGroup group)
  Run the callback dispatcher.
- PUBLIC void ixQMgrDispatcherLoopRunB0 (IxQMgrDispatchGroup group)
  Run the callback dispatcher.
- PUBLIC IX_STATUS ixQMgrNotificationCallbackSet (IxQMgrQId qId, IxQMgrCallback callback, IxQMgrCallbackId callbackId)
  Set the notification callback for a queue.
- PUBLIC void ixQMgrDispatcherLoopGet (IxQMgrDispatcherFuncPtr *qDispatcherFuncPtr)
  Get QMgr DispatcherLoopRun for respective silicon device.
- PUBLIC void ixQMgrStickyInterruptRegEnable (void)
  Enable AQM's sticky interrupt register behaviour only available on B0 Silicon.

A.30.0.6 Variables

- IxQMgrQInlinedReadWriteInfo **ixQMgrQInlinedReadWriteInfo** []
- UINT32 **ixQMgrAqmIfQueLowStatRegAddr** []
- UINT32 **ixQMgrAqmIfQueLowStatBitsOffset** []
- UINT32 **ixQMgrAqmIfQueLowStatBitsMask**
- UINT32 **ixQMgrAqmIfQueUppStat0RegAddr**

- UINT32 **ixQMgrAqmIfQueUppStat1RegAddr**
- UINT32 **ixQMgrAqmIfQueUppStat0BitMask** []
- UINT32 **ixQMgrAqmIfQueUppStat1BitMask** []

A.30.0.7 Detailed Description

The public API for the IXP425 QMgr component.

IxQMgr is a low level interface to the AHB Queue Manager

A.30.0.8 Define Documentation

A.30.0.9 #define IX_QMGR_CB_ALREADY_SET 7

Callback set error return code.

The specified callback has already been for this queue

Definition at line 266 of file IxQMgr.h.

A.30.0.10 #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.
Definition at line 393 of file IxQMgr.h.

A.30.0.11 #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.

Definition at line 118 of file IxQMgr.h.

A.30.0.12 #define IX_QMGR_INVALID_CB_ID 6

Invalid callback identifier return code.

Invalid callback id

Definition at line 253 of file IxQMgr.h.

A.30.0.13 #define IX_QMGR_INVALID_INT_SOURCE_ID 9

Invalid queue interrupt source identifier return code.

Invalid queue interrupt source given for notification enable

Definition at line 291 of file IxQMgr.h.

A.30.0.14 #define IX_QMGR_INVALID_Q_ENTRY_SIZE 4

Invalid entry size return code.

Invalid queue entry size for a queue read/write

Definition at line 228 of file IxQMgr.h.

A.30.0.15 #define IX_QMGR_INVALID_Q_ID 5

Invalid queue identifier return code.

Invalid queue id, not in range 0-63

Definition at line 241 of file IxQMgr.h.

A.30.0.16 #define IX_QMGR_INVALID_Q_WM 11

Invalid queue watermark return code.

Invalid queue watermark given for watermark set

Definition at line 317 of file IxQMgr.h.

A.30.0.17 #define IX_QMGR_INVALID_QSIZE 10

Invalid queue size error code.

Invalid queue size not one of 16,32, 64, 128

Definition at line 305 of file IxQMgr.h.

A.30.0.18 #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

Definition at line 135 of file IxQMgr.h.

A.30.0.19 #define IX_QMGR_MAX_QID IX_QMGR_QUEUE_63

Maximum queue identifier.

This constant is used to indicate the largest queue identifier

Definition at line 161 of file IxQMgr.h.

A.30.0.20 #define IX_QMGR_MAX_QNAME_LEN 16

Maximum queue name length.

This constant is used to indicate the maximum null terminated string length (excluding '') for a queue name

Definition at line 189 of file IxQMgr.h.

A.30.0.21 #define IX_QMGR_MIN_QID IX_QMGR_QUEUE_0

Minimum queue identifier.

This constant is used to indicate the smallest queue identifier

Definition at line 148 of file IxQMgr.h.

A.30.0.22 #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

Definition at line 175 of file IxQMgr.h.

Referenced by ixQMgrQBurstRead(), ixQMgrQBurstWrite(), ixQMgrQRead(), ixQMgrQStatusGet(), and ixQMgrQWrite().

A.30.0.23 #define IX_QMGR_NO_AVAILABLE_SRAM 8

Sram consumed return code.

All AQM Sram is consumed by queue configuration

Definition at line 279 of file IxQMgr.h.

A.30.0.24 #define IX_QMGR_PARAMETER_ERROR 3

Parameter error return code (NULL pointer etc..).

parameter error out of range/invalid

Definition at line 215 of file IxQMgr.h.

A.30.0.25 #define IX_QMGR_Q_ALREADY_CONFIGURED 13

Queue already configured return code.

Returned to client to indicate that a queue has already been configured

Definition at line 343 of file IxQMgr.h.

A.30.0.26 #define IX_QMGR_Q_INVALID_PRIORITY 16

Invalid priority return code.

Invalid priority, not one of 0,1,2

Definition at line 381 of file IxQMgr.h.

A.30.0.27 #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

Definition at line 331 of file IxQMgr.h.

A.30.0.28 #define IX_QMGR_Q_OVERFLOW 15

Overflow return code.

Overflow on a queue write has occurred

Definition at line 369 of file IxQMgr.h.

Referenced by ixQMgrQBurstWrite(), and ixQMgrQWrite().

A.30.0.29 #define IX_QMGR_Q_UNDERFLOW 14

Underflow return code.

Underflow on a queue read has occurred

Definition at line 356 of file IxQMgr.h.

Referenced by ixQMgrQBurstRead(), and ixQMgrQRead().

A.30.0.30 #define IX_QMGR_SAVED_COMPONENT_NAME IX_COMPONENT_NAME

Because this file contains inline functions which will be compiled into other components, we need to ensure that the IX_COMPONENT_NAME define is set to ix_qmgr while this code is being compiled.

This will ensure that the correct implementation is provided for the memory access macros IX_OSSERV_READ_LONG and IX_OSSERV_WRITE_LONG which are used in this file. This must be done before including "IxOsServicesMemAccess.h"

Definition at line 86 of file IxQMgr.h.

A.30.0.31 #define IX_QMGR_WARNING 2

Warning return code.

Execution complete, but there is a special case to handle

Definition at line 202 of file IxQMgr.h.

A.30.0.32 #define ixQMgrDispatcherLoopRun ixQMgrDispatcherLoopRunA0

Map old function name ixQMgrDispatcherLoopRun() to ixQMgrDispatcherLoopRunA0().

Definition at line 405 of file IxQMgr.h.

A.30.0.33 Typedef Documentation

A.30.0.34 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

Definition at line 692 of file IxQMgr.h.

A.30.0.35 IxQMgrCallbackId

Uniquely identifies a callback function.

A unique callback identifier associated with each callback registered by clients.

Definition at line 680 of file IxQMgr.h.

A.30.0.36 IxQMgrDispatcherFuncPtr

QMgr Dispatcher Loop function pointer.

This defines the interface for QMgr Dispather functions.

Parameters

IxQMgrDispatchGroup group(in) - the group of the queue of which the dispatcher will run

Definition at line 707 of file IxQMgr.h.

A.30.0.37 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 - bit-5
- UF - bit-4
- F - bit-3

- NF - bit-2
- NE - bit-1
- E - bit-0

Definition at line 516 of file IxQMgr.h.

A.30.0.38 Enumeration Type Documentation

A.30.0.39 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.

Definition at line 644 of file IxQMgr.h.

A.30.0.40 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.
  Definition at line 661 of file IxQMgr.h.

A.30.0.41 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

Definition at line 574 of file IxQMgr.h.

Referenced by ixQMgrQBurstRead(), and ixQMgrQBurstWrite().

A.30.0.42 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.

Definition at line 426 of file IxQMgr.h.

A.30.0.43 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*** Insure that this is greater than largest queue size supported by the hardware.

Definition at line 592 of file IxQMgr.h.

A.30.0.44 enum IxQMgrQStatusMask

Queue status mask.

Masks for extracting the individual status flags from the IxQMgrStatus word.

Definition at line 529 of file IxQMgr.h.

A.30.0.45 enum IxQMgrSourceId

Queue interrupt source select.

This enum defines the different source conditions on a queue that result in an interupt 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.

Definition at line 552 of file IxQMgr.h.

A.30.0.46 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

Definition at line 620 of file IxQMgr.h.

A.30.0.47 Function Documentation

A.30.0.48 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 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_AVAILABLE_SRAM, all AQM SRAM is consumed by the queue configuration.

A.30.0.49 ixQMgrDispatcherLoopGet (IxQMgrDispatcherFuncPtr * *qDispatcherFuncPtr*)

Get QMgr DispatcherLoopRun for respective silicon device.

This function get the function pointer for ixQMgrDispatcherLoopRunA0() for A0 or ixQMgrDispatcherLoopRunB0() for B0 Silicon.

Parameters

IxQMgrDispatcherFuncPtr(out) *qDispatcherFuncPtr - the function pointer of QMgr Dispatcher

A.30.0.50 ixQMgrDispatcherLoopRunA0 (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

Note: This function may be called from interrupt or task context. However, for optimal performance the choice of context depends also on the operating system used.

For optimal performance with the WindRiver VxWorks OS the recommendation is to use a polling task to call the Dispatcher, for the following reasons:

- The system cannot handle more than 42000 interrupts per second. As a result, the system may crash at 25% of the maximum traffic rate for Ethernet traffic.
- Using a polling task to call the Dispatcher works fine.

For optimal performance with the Linux OS the recommendation is to use interrupts to call the Dispatcher, for the following reasons:

- A task loop cannot run more than 100 times per second (HZ). As an example, if an Ethernet port uses a queue size of 128 entries, the maximum achievable traffic rate is 12800 packets per second. (the system does not crash, but this is 8% of wire speed)
- Using queue interrupts to call the Dispatcher works fine.

A.30.0.51 ixQMgrDispatcherLoopRunB0 (IxQMgrDispatchGroup *group*)

Run the callback dispatcher.

The enhanced version of ixQMgrDispatcherLoopRun () for B0 silicon. 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

Note: Compared with ixQMgrDispatcherLoopRunA0(), ixQMgrDispatcherLoopRunB0() is more efficient and takes advantange of QMgr optimization in B0 silicon.

This function may be called from interrupt or task context. However, for optimal performance the choice of context depends also on the operating system used.

For optimal performance with the WindRiver VxWorks OS the recommendation is to use a polling task to call the Dispatcher, for the following reasons:

- The system cannot handle more than 42000 interrupts per second. As a result, the system may crash at 25% of the maximum traffic rate for Ethernet traffic.
- Using a polling task to call the Dispatcher works fine.

  For optimal performance with the Linux OS the recommendation is to use interrupts to call the Dispatcher, for the following reasons:
- A task loop cannot run more than 100 times per second (HZ). As an example, if an Ethernet port uses a queue size of 128 entries, the maximum achievable traffic rate is 12800 packets per second. (the system does not crash, but this is 8% of wire speed)
- Using queue interrupts to call the Dispatcher works fine.

A.30.0.52 ixQMgrDispatcherPrioritySet (IxQMgrQId *qId*, IxQMgrPriority *priority*)

Set the dispatch priority of a queue.

This function is called to set the disptach 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.30.0.53 ixQMgrInit (void)

Initialise the QMgr.

This function must be called before and other QMgr function. It sets up internal data structures.

Returns

- IX_SUCCESS, the IxQMgr successfully initialised
- IX_FAIL, failed to initialize the Qmgr

A.30.0.54 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 - the queue idenfifier
- *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.30.0.55 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.30.0.56 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 notofication 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 constistent

A.30.0.57 IX_QMGR_INLINE PUBLIC IX_STATUS 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 read. This number should be greater than 0
- *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

  Definition at line 1211 of file IxQMgr.h.

  References IX_QMGR_MIN_QUEUPP_QID, IX_QMGR_Q_ENTRY_SIZE1, IX_QMGR_Q_UNDERFLOW, IX_SUCCESS, IxQMgrQEntrySizeInWords, IxQMgrQInlinedReadWriteInfo::qAccRegAddr, IxQMgrQInlinedReadWriteInfo::qEntrySizeInWords, IxQMgrQInlinedReadWriteInfo::qUflowStatBitMask, IxQMgrQInlinedReadWriteInfo::qUOStatRegAddr, and IxQMgrQInlinedReadWriteInfo::qWriteCount.

A.30.0.58 IX_QMGR_INLINE PUBLIC IX_STATUS 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.

Note: 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

  Definition at line 1553 of file IxQMgr.h.

  References IX_QMGR_MIN_QUEUPP_QID, IX_QMGR_Q_ENTRY_SIZE1, IX_QMGR_Q_OVERFLOW, IX_SUCCESS, IxQMgrQEntrySizeInWords, IxQMgrQInlinedReadWriteInfo::qAccRegAddr, IxQMgrQInlinedReadWriteInfo::qEntrySizeInWords, IxQMgrQInlinedReadWriteInfo::qOflowStatBitMask, IxQMgrQInlinedReadWriteInfo::qSizeInEntries, IxQMgrQInlinedReadWriteInfo::qUOStatRegAddr, and IxQMgrQInlinedReadWriteInfo::qWriteCount.

A.30.0.59 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 " terminated array of 16 charachters 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.30.0.60 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 - the queue idenfifier
- *unsigned(out)* *numEntries - the number of entries in a queue

Returns

- IX_SUCCESS, got the number of entries for the queue
- IX_QMGR_PARAMETER_ERROR, invalid paramter(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.30.0.61 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. N.B: 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 paramter(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 inpected the queue entry.

A.30.0.62 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. N.B: 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 paramter(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.30.0.63 IX_QMGR_INLINE PUBLIC IX_STATUS ixQMgrQRead (IxQMgrQId *qId*, UINT32 * *entryPtr*)

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

  Definition at line 1060 of file IxQMgr.h.

  References IX_QMGR_MIN_QUEUPP_QID, IX_QMGR_Q_ENTRY_SIZE1, IX_QMGR_Q_UNDERFLOW, IX_SUCCESS, ixQMgrQReadMWordsMinus1(), IxQMgrQInlinedReadWriteInfo::qAccRegAddr, IxQMgrQInlinedReadWriteInfo::qConfigRegAddr, IxQMgrQInlinedReadWriteInfo::qEntrySizeInWords, IxQMgrQInlinedReadWriteInfo::qReadCount, IxQMgrQInlinedReadWriteInfo::qSizeInEntries, IxQMgrQInlinedReadWriteInfo::qUflowStatBitMask, and IxQMgrQInlinedReadWriteInfo::qUOStatRegAddr.

A.30.0.64 IX_STATUS ixQMgrQReadMWordsMinus1 (IxQMgrQId *qId*, UINT32 * *entry*)

This function reads the remaining of the q entry for queues configured with many words. (the first word of the entry is already read in the inlined function and the entry pointer already incremented.

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

  Referenced by ixQMgrQRead().

A.30.0.65 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 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_PARAMETER_ERROR, invalid paramter(s).
- IX_QMGR_Q_NOT_CONFIGURED, queue not configured for this QId

- IX_QMGR_Q_UNDERFLOW, attempt to read from an empty queue

A.30.0.66 ixQMgrQShow (IxQMgrQId *qId*)

Display aqueue configuration and statistics for a queue.

This function shows queue configuration and statistics for a queue.

Parameters

IxQMgrQId (in)qId - the queue identifier.

Returns

- IX_SUCCESS, success
- IX_QMGR_Q_NOT_CONFIGURED, queue not configured for this QId

A.30.0.67 ixQMgrQSizeInEntriesGet (IxQMgrQId *qId*, unsigned * *qSizeInEntries*)

Return the size of a queue in entries.

This function returns the the size of the queue in entriese.

Parameters

- *IxQMgrQId(in)* qId - the queue identifier
- *IxQMgrQSize(out)* *qSizeInEntries - queue size in entries

Returns

- IX_SUCCESS, successfully retrieved the number of full entrie
- IX_QMGR_Q_NOT_CONFIGURED, queue not configured for this QId
- IX_QMGR_PARAMETER_ERROR, invalid parameter(s).

A.30.0.68 IX_QMGR_INLINE PUBLIC IX_STATUS 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.

Definition at line 1748 of file IxQMgr.h.

References IX_QMGR_MIN_QUEUPP_QID, IX_SUCCESS, IxQMgrQInlinedReadWriteInfo::qOflowStatBitMask, IxQMgrQInlinedReadWriteInfo::qUflowStatBitMask, and IxQMgrQInlinedReadWriteInfo::qUOStatRegAddr.

A.30.0.69 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, the specified qId has not been configured
- IX_QMGR_PARAMETER_ERROR, invalid paramter.

A.30.0.70 IX_QMGR_INLINE PUBLIC IX_STATUS 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

  Definition at line 1406 of file IxQMgr.h.

  References IX_QMGR_MIN_QUEUPP_QID, IX_QMGR_Q_ENTRY_SIZE1, IX_QMGR_Q_OVERFLOW, IX_SUCCESS, IxQMgrQInlinedReadWriteInfo::qAccRegAddr, IxQMgrQInlinedReadWriteInfo::qConfigRegAddr, IxQMgrQInlinedReadWriteInfo::qEntrySizeInWords, IxQMgrQInlinedReadWriteInfo::qOflowStatBitMask, IxQMgrQInlinedReadWriteInfo::qSizeInEntries, IxQMgrQInlinedReadWriteInfo::qUOStatRegAddr, and IxQMgrQInlinedReadWriteInfo::qWriteCount.

A.30.0.71 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 paramter(s).
- IX_QMGR_Q_NOT_CONFIGURED, queue not configured for this QId
- IX_QMGR_Q_OVERFLOW, attempt to write to a full queue

A.30.0.72 ixQMgrShow (void)

Describe queue configuration and statistics for active queues.

This function shows active queues, their configurations and statistics.

Returns

void

A.30.0.73 ixQMgrStickyInterruptRegEnable (void)

Enable AQM's sticky interrupt register behaviour only available on B0 Silicon.

When AQM's sticky interrupt register is enabled, interrupt register bit will only be cleared when a '1' is written to interrupt register bit and the interrupting condition is satisfied, i.e.queue condition does not exist.

Note: This function must be called before any queue is enabled. Calling this function after queue is enabled will cause undefined results.

Returns

none

A.30.0.74 ixQMgrUnload (void)

Uninitialise the QMgr.

This function will perform the tasks required to unload the QMgr component cleanly. This includes unmapping kernel memory. This should be called before a soft reboot or unloading of a kernel module.

Precondition

It should only be called if ixQMgrInit has already been called.

Postcondition

No QMgr functions should be called until ixQMgrInit is called again.

Returns

- IX_SUCCESS, the IxQMgr successfully uninitialised
- IX_FAIL, failed to uninitialize the Qmgr

A.30.0.75 ixQMgrWatermarkSet (IxQMgrQId *qId*, IxQMgrWMLevel *ne*, IxQMgrWMLevel *nf*)

Set the Nearly Empty and Nearly Full Watermarks fo 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 queu
- IX_QMGR_Q_NOT_CONFIGURED, queue not configured for this QId
- IX_QMGR_INVALID_Q_WM, invalid watermark
- IX_QMGR_WARNING, the status register may not be constistent

A.31 IXP425 Timer Control (IxTimerCtrl) API

IXP425 Timer Control (IxTimerCtrl) API. IXP425 Timer Control (IxTimerCtrl) APIIXP425 Timer Control (IxTimerCtrl) APIIXP425 Timer Control (IxTimerCtrl) APIThe public API for the IXP425 Timer Control Component.

A.31.0.1 Defines

- #define IX_TIMERCTRL_NO_FREE_TIMERS 2
  Timer schedule return code.
- #define IX_TIMERCTRL_PARAM_ERROR 3
  Timer schedule return code.

A.31.0.2 Typedefs

- typedef void(* IxTimerCtrlTimerCallback)(void *userParam)
  A typedef for a pointer to a timer callback function.

A.31.0.3 Enumerations

- enum IxTimerCtrlPurpose { **IxTimerCtrlAdslPurpose**, **IxTimerCtrlMaxPurpose** }
  List used to identify the users of timers.

A.31.0.4 Functions

- IX_STATUS ixTimerCtrlSchedule (IxTimerCtrlTimerCallback func, void *userParam, IxTimerCtrlPurpose purpose, UINT32 relativeTime, 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)
  Initialise the Timer Control Component.
- void ixTimerCtrlShow (void)

Display the status of the Timer Control Component.

A.31.0.5 Detailed Description

The public API for the IXP425 Timer Control Component.

A.31.0.6 Define Documentation

A.31.0.7 #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.

Definition at line 77 of file IxTimerCtrl.h.

A.31.0.8 #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.

Definition at line 90 of file IxTimerCtrl.h.

A.31.0.9 Typedef Documentation

A.31.0.10 typedef void(* IxTimerCtrlTimerCallback)(void *userParam)

A typedef for a pointer to a timer callback function.

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.

Definition at line 110 of file IxTimerCtrl.h.

A.31.0.11 Enumeration Type Documentation

A.31.0.12 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).

Definition at line 122 of file IxTimerCtrl.h.

A.31.0.13 Function Documentation

A.31.0.14 ixTimerCtrlCancel (unsigned *id*)

Cancels a scheduled callback.

Parameters

id - the id of the callback to be cancelled.

Returns

- IX_SUCCESS - The timer was successfully stopped.
- IX_FAIL - The id parameter did not corrrespond 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 accesed by another thread.

A.31.0.15 ixTimerCtrlInit (void)

Initialise 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.31.0.16 ixTimerCtrlSchedule (IxTimerCtrlTimerCallback *func*, void * *userParam*, IxTimerCtrlPurpose *purpose*, UINT32 *relativeTime*, 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 100 ms. 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.
- *relativeTime* (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 accesed by another thread.

A.31.0.17 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) - 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.
- *interval* - the interval in milliseconds between calls to func.
- *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 accesed by another thread.

A.31.0.18 ixTimerCtrlShow (void)

Display the status of the Timer Control Component.

Returns

void

Note: Displays a list of running timers. This function is not re-entrant. This function does not suspend the calling thread.

A.32 IXP425 Types (IxTypes)

IXP425 Types (IxTypes). IXP425 Types (IxTypes)IXP425 Types (IxTypes)IXP425 Types (IxTypes)Basic data types used by the IXP425 project.

A.32.0.1 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** static

A.32.0.2 Typedefs

- typedef void(* **IxVoidFnPtr**)(void)
- typedef void(* **IxVoidFnVoidPtr**)(void *)
- typedef int(* **FUNCPTR**)(void)
- 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 unsigned long long **UINT64**
- typedef void **VOID**
- typedef volatile UINT32 **VUINT32**
- typedef volatile INT32 **VINT32**
- typedef UINT32 **IX_STATUS**

A.32.0.3 Detailed Description

Basic data types used by the IXP425 project.

A.33 IXP425 UART Access (IxUARTAcc) API

IXP425 UART Access (IxUARTAcc) API. IXP425 UART Access (IxUARTAcc) APIIXP425 UART Access (IxUARTAcc) APIIXP425 UART Access (IxUARTAcc) APIIXP425 UARTAcc Driver Public API.

A.33.0.1 Modules

- Defines for Default Values
  Default values which can be used for UART configuration.
- Defines for IOCTL Commands
  IOCTL Commands (Request codes) which can be used with ixUARTIoctl .
- Defines for IOCTL Arguments
  POSIX style IOCTL arguments which can be used with ixUARTIoctl .

A.33.0.2 Data Structures

- struct ixUARTDev
  Device descriptor for the UART.
- struct ixUARTDev
  Device descriptor for the UART.
- struct ixUARTStats
  Statistics for the UART.
- struct ixUARTStats
  Statistics for the UART.

A.33.0.3 Enumerations

- enum ixUARTMode { INTERRUPT = 0, POLLED, LOOPBACK }
  The mode to set to UART to.

A.33.0.4 Functions

- PUBLIC IX_STATUS ixUARTInit (ixUARTDev *pUART)
  Initialise 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.33.0.5 Detailed Description

IXP425 UARTAcc Driver Public API.

A.33.0.6 Enumeration Type Documentation

A.33.0.7 enum ixUARTMode

The mode to set to UART to.

Enumeration Values

- ***INTERRUPT*** Interrupt mode.
- ***POLLED*** Polled mode.
- ***LOOPBACK*** Loopback mode.

Definition at line 310 of file IxUART.h.

A.33.0.8 Function Documentation

A.33.0.9 IX_STATUS ixUARTInit (ixUARTDev * *pUART*)

Initialise the UART. This puts the chip in a quiescent state.

Parameters

pUART - pointer to UART structure describing our device.

Precondition

The base address for the UART must contain a valid value. Also the baud rate and hardware options must contain sensible values otherwise the defaults will be used as defined in ixUART.h

Postcondition

UART is initialized and ready to send and receive data.

Note: This function should only be called once per device.

Returns

- *IX_SUCCESS* - UART device successfully initialised.
- *IX_FAIL* - Critical error, device not initialised.

A.33.0.10 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* - an 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.

See also:

IoctlCommandDefines
IoctlArgDefines

A.33.0.11 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.

Precondition

UART device must be initialised.

Returns

- *IX_SUCCESS* - character was successfully read.
- *IX_FAIL* - input buffer empty (try again).

A.33.0.12 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.

Precondition

UART device must be initialised.

Returns

- *IX_SUCCESS* - character was successfully transmitted.
- *IX_FAIL* - output buffer is full (try again).

A.33.1 Defines for Default Values

Defines for Default ValuesDefines for Default ValuesDefines for Default ValuesDefines for Default ValuesDefault values which can be used for UART configuration.

A.33.1.1 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.

A.33.1.2 Detailed Description

Default values which can be used for UART configuration.

See also:

ixUARTDev

A.33.2 Defines for IOCTL Commands

Defines for IOCTL Commands. Defines for IOCTL CommandsDefines for IOCTL CommandsDefines for IOCTL CommandsIOCTL Commands (Request codes) which can be used with ixUARTIoctl .

A.33.2.1 Defines

- #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.

A.33.2.2 Detailed Description

IOCTL Commands (Request codes) which can be used with ixUARTIoctl .

A.33.2.3 Define Documentation

A.33.2.4 #define IX_BAUD_GET 1

Get the baud rate.

Definition at line 158 of file IxUART.h.

A.33.2.5 #define IX_BAUD_SET 0

Set the baud rate.

Definition at line 149 of file IxUART.h.

A.33.2.6 #define IX_MODE_GET 3

Get the current UART mode of operation.

Definition at line 174 of file IxUART.h.

A.33.2.7 #define IX_MODE_SET 2

Set the UART mode of operation.

Definition at line 165 of file IxUART.h.

A.33.2.8 #define IX_OPTS_GET 5

Get the UART device options.

Definition at line 192 of file IxUART.h.

A.33.2.9 #define IX_OPTS_SET 4

Set the UART device options.

Definition at line 183 of file IxUART.h.

A.33.2.10 #define IX_STATS_GET 6

Get the UART statistics.

Definition at line 201 of file IxUART.h.

A.33.3 Defines for IOCTL Arguments

Defines for IOCTL Arguments. Defines for IOCTL ArgumentsDefines for IOCTL ArgumentsDefines for IOCTL ArgumentsPOSIX style IOCTL arguments which can be used with ixUARTIoctl .

A.33.3.1 Defines

- #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.33.3.2 Detailed Description

POSIX style IOCTL arguments which can be used with ixUARTIoctl .

See also:

ixUARTMode

A.33.3.3 Define Documentation

A.33.3.4 #define CLOCAL 0x1

Software flow control.

Definition at line 223 of file IxUART.h.

A.33.3.5 #define CREAD 0x2

Enable interrupt receiver.

Definition at line 232 of file IxUART.h.

A.33.3.6 #define CS5 0x0

5 bits

Definition at line 250 of file IxUART.h.

A.33.3.7 #define CS6 0x4

6 bits

Definition at line 259 of file IxUART.h.

A.33.3.8 #define CS7 0x8

7 bits

Definition at line 268 of file IxUART.h.

A.33.3.9 #define CS8 0xc

8 bits

Definition at line 277 of file IxUART.h.

A.33.3.10 #define CSIZE 0xc

Characters size.

Definition at line 241 of file IxUART.h.

A.33.3.11 #define PARENB 0x40

Parity detection enabled (else disabled).

Definition at line 295 of file IxUART.h.

A.33.3.12 #define PARODD 0x80

Odd parity (else even).

Definition at line 304 of file IxUART.h.

A.33.3.13 #define STOPB 0x20

Send two stop bits (else one).

Definition at line 286 of file IxUART.h.

A.34 IXP400 Version ID (IxVersionId)

IXP400 Version ID (IxVersionId)IXP400 Version ID (IxVersionId)IXP400 Version ID (IxVersionId)IXP400 Version ID (IxVersionId)Version Identifiers.

A.34.0.1 Defines

- #define IX_VERSION_ID "1_3"
  Version Identifier String.
- #define IX_VERSION_INTERNAL_ID "SQA2_1"
  Internal Release Identifier String.
- #define IX_VERSION_COMPATIBLE_TORNADO "Tornado2_2"
  Compatible Tornado Version Identifier.
- #define IX_VERSION_COMPATIBLE_LINUX "MVL3_0"
  Compatible Linux Version Identifier.

A.34.0.2 Detailed Description

Version Identifiers.

A.34.0.3 Define Documentation

A.34.0.4 #define IX_VERSION_ID "1_3"

Version Identifier String.

This string will be updated with each customer release of the IXP400 Software.

Definition at line 59 of file IxVersionId.h.

A.34.0.5 #define IX_VERSION_INTERNAL_ID "SQA2_1"

Internal Release Identifier String.

This string will be updated with each internal release (SQA drop) of the IXP400 Software.

Definition at line 67 of file IxVersionId.h.

A.35 IXP425 USB Driver Public API

IXP425 USB Driver Public API. IXP425 USB Driver Public APIIXP425 USB Driver Public APIIXP425 USB Driver Public APIIXP425 USB Driver Public API.

A.35.0.1 Data Structures

- struct USBDevice
  USBDevice.
- struct USBSetupPacket
  Standard USB Setup packet components, see the USB Specification 1.1.

A.35.0.2 Defines

- #define IX_USB_MBLK IX_MBUF
  Memory buffer.
- #define IX_USB_MBLK_DATA(buf) IX_MBUF_MDATA(buf)
  Return pointer to the data in the mbuf.
- #define IX_USB_MBLK_LEN(buf) IX_MBUF_MLEN(buf)
  Return pointer to the data length.
- #define IX_USB_MBLK_FREE(buf) if (buf) { IX_MBUF_POOL_PUT(buf); }
  Returns a buffer to the buffer pool.
- #define IX_USB_MBLK_PKT_LEN(buf) IX_MBUF_PKT_LEN(buf)
  Return pointer to the total length of all the data in the mbuf chain for this packet.
- #define IX_USB_HAS_GET_ERROR_STRING
  define to enable ixUSBErrorStringGet()
- #define IX_USB_HAS_ENDPOINT_INFO_SHOW
  define to enable ixUSBEndpointInfoShow()
- #define IX_USB_HAS_STATISTICS_SHOW
  define to enable ixUSBStatisticsShow()
- #define IX_USB_STATS_SHOW_PER_ENDPOINT_INFO
  define to enable per-endpoint information in ixUSBStatisticsShow()
- #define IX_USB_HAS_VERBOSE_WARN_TRACE_MACRO
  define to enable verbose warning tracing

- #define IX_USB_HAS_ASSERT_MACRO
  define to enable assertion macro
- #define IX_USB_HAS_CT_ASSERT_MACRO
  define to enable compile-time assertion macro
- #define IX_USB_HAS_INT_BIND_MACRO
  define to enable interrupt handler binding for VxWorks
- #define UDC_REGISTERS_BASE IX_OSSERV_USB_PHYS_BASE
  Base I/O address.
- #define UDC_IRQ IXP425_INT_LVL_USB
  IRQ.
- #define NUM_ENDPOINTS 16
  Number of endpoints.
- #define SETUP_PACKET_SIZE 8
  SETUP packet size.
- #define CONTROL_FIFO_SIZE 16
  CONTROL endpoint FIFO depth.
- #define CONTROL_PACKET_SIZE 16
  CONTROL endpoint packet size.
- #define INTERRUPT_FIFO_SIZE 8
  INTERRUPT endpoint FIFO depth.
- #define INTERRUPT_PACKET_SIZE 8
  INTERRUPT endpoint packet size.
- #define BULK_FIFO_SIZE 64
  BULK endpoint FIFO depth.
- #define BULK_PACKET_SIZE 64
  BULK endpoint packet size.
- #define ISOCHRONOUS_FIFO_SIZE 256
  ISOCHRONOUS endpoint FIFO depth.
- #define ISOCHRONOUS_PACKET_SIZE 256
  ISOCHRONOUS endpoint packet size.
- #define MAX_TRANSFER_SIZE 2048
  Maximum data size for one transaction in bytes (bulk or control).
- #define MAX_QUEUE_SIZE 100
  *Maximum outgoing queue size per endpoint, in elements Uses MAX_QUEUE_SIZE * (sizeof(void *)) bytes.*
- #define MEM_POOL_SIZE 10240
  Memory pool for data transactions.
- #define TRANSACTION_TIMEOUT_RX 500
  Maximum acceptable delay in transactions (timestamp ticks), Rx, 0 disables.

- #define TRANSACTION_TIMEOUT_TX 500
  Maximum acceptable delay in transactions (timestamp ticks), Tx, 0 disables.
- #define IX_USB_ERROR_BASE 4096
  USB error base.
- #define IX_USB_ERROR (IX_USB_ERROR_BASE + 0)
  error due to unknown reasons
- #define IX_USB_INVALID_DEVICE (IX_USB_ERROR_BASE + 1)
  invalid USBDevice structure passed as parameter or no device present
- #define IX_USB_NO_PERMISSION (IX_USB_ERROR_BASE + 2)
  no permission for attempted operation
- #define IX_USB_REDUNDANT (IX_USB_ERROR_BASE + 3)
  redundant operation
- #define IX_USB_SEND_QUEUE_FULL (IX_USB_ERROR_BASE + 4)
  send queue full
- #define IX_USB_NO_ENDPOINT (IX_USB_ERROR_BASE + 5)
  invalid endpoint
- #define IX_USB_NO_IN_CAPABILITY (IX_USB_ERROR_BASE + 6)
  no IN capability on endpoint
- #define IX_USB_NO_OUT_CAPABILITY (IX_USB_ERROR_BASE + 7)
  no OUT capability on endpoint
- #define IX_USB_NO_TRANSFER_CAPABILITY (IX_USB_ERROR_BASE + 8)
  transfer type incompatible with endpoint
- #define IX_USB_ENDPOINT_STALLED (IX_USB_ERROR_BASE + 9)
  endpoint stalled
- #define IX_USB_INVALID_PARMS (IX_USB_ERROR_BASE + 10)
  invalid parameter(s)
- #define IX_USB_DEVICE_DISABLED (IX_USB_ERROR_BASE + 11)
  device is disabled
- #define IX_USB_NO_STALL_CAPABILITY (IX_USB_ERROR_BASE + 12)
  no STALL capability
- #define **logMsg** printk
- #define EP_DIRECTION(x) ((x) & (USB_IN | USB_OUT))
  Macro used to extract the endpoint direction from an EPDescriptorTable[] entry.
- #define EP_TYPE(x) ((x) & (USB_CONTROL | USB_BULK | USB_ISOCHRONOUS | USB_INTERRUPT))
  Macro used to extract the endpoint type from an EPDescriptorTable[] entry.
- #define MIN(a, b) ((a) < (b)? (a) : (b))
  Compares two values and returns the minimum.
- #define MAX(a, b) ((a) > (b)? (a) : (b))

Compares two values and returns the maximum.

- #define QUEUE_WRAP(tail) ((tail) >= (MAX_QUEUE_SIZE) ? ((tail) - (MAX_QUEUE_SIZE)) : (tail))
  Ajusts the tail of a queue implemented in a circular buffer by wrapping at the buffer boundary.
- #define SWAP_USB_WORD(wPtr) if (0);
  USB byte swapping routine for a little endian platform.
- #define REG_GET(reg_ptr) IX_OSSERV_READ_LONG(reg_ptr)
  read generic register access via register pointers
- #define REG_SET(reg_ptr, val) IX_OSSERV_WRITE_LONG(reg_ptr, val)
  write generic register access via register pointers
- #define DREG_GET(reg_ptr) (IX_OSSERV_READ_LONG(reg_ptr) & UDC_UDDR_RW_MASK)
  generic data register read access via register pointers
- #define DREG_SET(reg_ptr, val) IX_OSSERV_WRITE_LONG(reg_ptr, val & UDC_UDDR_RW_MASK)
  generic data register write access via register pointers
- #define CONTEXT(device) ((USBDeviceContext *)((device)->deviceContext))
  get context from device pointer
- #define REGISTERS(device) (CONTEXT(device)->registers)
  get registers from device pointer
- #define EP0CONTROL(device) (&(CONTEXT(device)->ep0ControlData))
  get endpoint 0 control data from device pointer
- #define EVENTS(device) (&(CONTEXT(device)->eventProcessor))
  get event processor from device pointer
- #define COUNTERS(device) (&(CONTEXT(device)->counters))
  get device counters
- #define OPERATION(device) (&(CONTEXT(device)->deviceOperation))
  get device operation
- #define EPSTATUS(device, endpointNumber) (&(CONTEXT(device)->epStatusData[endpointNumber]))
  get endpoint status from device pointer and endpoint number
- #define EPQUEUE(device, endpointNumber) (&(EPSTATUS((device), (endpointNumber))->queue))
  get endpoint queue from device pointer and endpoint number
- #define EPCOUNTERS(device, endpointNumber) (&(EPSTATUS((device), (endpointNumber))->counters))
  get endpoint counters from device pointer and endpoint number
- #define RETURN_OK(device)
  set IX_SUCCESS on device and return IX_SUCCESS
- #define RETURN_ERROR(device)
  set IX_USB_ERROR on device and return IX_FAIL

- #define RETURN_INVALID_PARMS(device)
  set IX_USB_INVALID_PARAMS on device and return IX_FAIL
- #define RETURN_REDUNDANT(device)
  set IX_USB_REDUNDANT on device and return IX_FAIL
- #define RETURN_INVALID_DEVICE(device)
  set IX_USB_INVALID_PARAMS on device and return IX_FAIL
- #define RETURN_NO_ENDPOINT(device)
  set IX_USB_INVALID_PARAMS on device and return IX_FAIL
- #define RETURN_ENDPOINT_STALLED(device)
  set IX_USB_ENDPOINT_STALLED on device and return IX_FAIL
- #define RETURN_SEND_QUEUE_FULL(device)
  set IX_USB_SEND_QUEUE_FULL on device and return IX_FAIL
- #define RETURN_NO_IN_CAPABILITY(device)
  set IX_USB_NO_IN_CAPABILITY on device and return IX_FAIL
- #define RETURN_NO_STALL_CAPABILITY(device)
  set IX_USB_NO_STALL_CAPABILITY on device and return IX_FAIL
- #define RETURN_NO_PERMISSION(device)
  set IX_USB_NO_PERMISSION on device and return IX_FAIL
- #define CHECK_DEVICE(device)
  sanity checks for device existence
- #define CHECK_DEVICE_ENABLED(device)
  sanity checks for device enable status
- #define CHECK_ENDPOINT(device, endpointNumber)
  sanity check for endpoint existence
- #define CHECK_ENDPOINT_STALL(device, endpointNumber)
  sanity check for endpoint stall
- #define CHECK_EVENT_MASK(device, eventMask)
  sanity check for event masks
- #define CHECK_ENDPOINT_QUEUE(epData)
  sanity check for endpoint queue size
- #define CHECK_ENDPOINT_IN_CAPABILITY(epData, device)
  sanity check for endpoint IN capability
- #define IX_USB_TRACE if (0) logMsg
  no trace macro
- #define IX_USB_LOCK 0
  dummy critial data section lock
- #define IX_USB_UNLOCK(state)
  dummy critial data section unlock
- #define IX_USB_IRQ_LOCK 0

dummy irq lock

- #define IX_USB_IRQ_UNLOCK(state)
  dummy irq unlock
- #define USB_CONTEXT_SIZE 7968
  USB context size.

A.35.0.3 Typedefs

- typedef UINT16 **USBEventSet**
- typedef void(* **USBEventCallback**)(USBDevice *device, USBEventSet events)
- typedef void(* **USBSetupCallback**)(USBDevice *device, const char *packet)
- typedef void(* **USBReceiveCallback**)(USBDevice *device, UINT16 sourceEndpoint, IX_USB_MBLK *receiveBuffer)

A.35.0.4 Enumerations

- enum USBEndpointDirection { **USB_NO_DATA** = 0, **USB_IN** = 0x01, **USB_OUT** = 0x02, **USB_IN_OUT** = USB_IN | USB_OUT }
  USB endpoint direction.
- enum USBEndpointType { **USB_CONTROL** = 0x10, **USB_BULK** = 0x20, **USB_INTERRUPT** = 0x40, **USB_ISOCHRONOUS** = 0x80 }

Note: *the values are set for compatibility with USBEndpointDirection.*

- enum USBEventMap { **USB_NO_EVENT** = 0, **USB_RESET** = 0x01, **USB_SUSPEND** = 0x02, **USB_RESUME** = 0x04, USB_SOF = 0x08, **USB_DEVICE_EVENTS** = USB_RESET | USB_SUSPEND | USB_RESUME, **USB_BUS_EVENTS** = USB_SOF, **USB_ALL_EVENTS** = USB_DEVICE_EVENTS | USB_BUS_EVENTS }
  USB Event Map.
- enum USBDeviceFlags { **ENABLE_RX_SEQ** = 0x01, **ENABLE_TX_SEQ** = 0x02 }
  USB Device Flags.
- enum USBEndpointNumber { **ENDPOINT_0** = 0, **ENDPOINT_1**, **ENDPOINT_2**, **ENDPOINT_3**, **ENDPOINT_4**, **ENDPOINT_5**, **ENDPOINT_6**, **ENDPOINT_7**, **ENDPOINT_8**, **ENDPOINT_9**, **ENDPOINT_10**, **ENDPOINT_11**, **ENDPOINT_12**, **ENDPOINT_13**, **ENDPOINT_14**, **ENDPOINT_15** }
  USB endpoint number.
- enum USBStdRequestType { **GET_STATUS_REQUEST** = 0x00, **CLEAR_FEATURE_REQUEST** = 0x01, **SET_FEATURE_REQUEST** = 0x03, **SET_ADDRESS_REQUEST** = 0x05, **GET_DESCRIPTOR_REQUEST** = 0x06, **SET_DESCRIPTOR_REQUEST** = 0x07, **GET_CONFIGURATION_REQUEST** = 0x08, **SET_CONFIGURATION_REQUEST** = 0x09, **GET_INTERFACE_REQUEST** = 0x0a, **SET_INTERFACE_REQUEST** = 0x0b, **SYNCH_FRAME_REQUEST** = 0x0c }
  Standard USB request types.
- enum USBStdDescriptorType { **USB_DEVICE_DESCRIPTOR** = 0x01, **USB_CONFIGURATION_DESCRIPTOR** = 0x02, **USB_STRING_DESCRIPTOR** = 0x03, **USB_INTERFACE_DESCRIPTOR** = 0x04, **USB_ENDPOINT_DESCRIPTOR** = 0x05 }

Standard USB descriptor types.

- enum USBStdFeatureSelector { **ENDPOINT_STALL** = 0x0, **DEVICE_REMOTE_WAKEUP** = 0x1 }
  Standard USB SET/CLEAR_FEATURE feature selector.
- enum USBStdLanguageId { **USB_ENGLISH_LANGUAGE** = 0x09 }
  Standard language IDs used by USB.
- enum USBStdEndpointType { **USB_CONTROL_ENDPOINT** = 0x00, **USB_ISOCHRONOUS_ENDPOINT** = 0x01, **USB_BULK_ENDPOINT** = 0x02, **USB_INTERRUPT_ENDPOINT** = 0x03 }
  Standard USB endpoint types.
- enum USBStdEndpointDirection { **USB_ENDPOINT_OUT** = 0x0, **USB_ENDPOINT_IN** = 1 }
  Standard USB directions.

A.35.0.5 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.
- PUBLIC IX_STATUS ixUSBStatisticsShow (USBDevice *device)
  Display device state and statistics.
- PUBLIC const char * ixUSBErrorStringGet (UINT32 errorCode)
  Convert an error code into a human-readable string error message.
- PUBLIC IX_STATUS ixUSBEndpointInfoShow (USBDevice *device)
  Display endpoint information table.

A.35.0.6 Detailed Description

IXP425 USB Driver Public API.

A.35.0.7 Define Documentation

A.35.0.8 #define CHECK_DEVICE(device)

Value

```
if (device == NULL)                                                          \
    {                                                                        \
        return IX_FAIL;                                                      \
    }                                                                        \
                                                                             \
    if (CONTEXT(device)->checkPattern != USB_DEVICE_CONTEXT_CHECK_PATTERN)   \
    {                                                                        \
        return IX_FAIL;                                                      \
    }
```

sanity checks for device existence

Definition at line 273 of file usbmacros.h.

A.35.0.9 #define CHECK_DEVICE_ENABLED(device)

Value

```
if (!CONTEXT(device)->enabled)                  \
    {                                           \
        device->lastError = IX_USB_DEVICE_DISABLED; \
        return IX_FAIL;                         \
    }
```

sanity checks for device enable status

Definition at line 285 of file usbmacros.h.

A.35.0.10 #define CHECK_ENDPOINT(device, endpointNumber)

Value

```
if (endpointNumber >= NUM_ENDPOINTS)        \
    {                                       \
        RETURN_NO_ENDPOINT(device);         \
    }
```

sanity check for endpoint existence

Definition at line 293 of file usbmacros.h.

A.35.0.11 #define CHECK_ENDPOINT_IN_CAPABILITY(epData, device)

Value

```
if ((epData->direction & USB_IN) == 0)          \
    {                                               \
        RETURN_NO_IN_CAPABILITY(device);            \
    }
```

sanity check for endpoint IN capability

Definition at line 326 of file usbmacros.h.

A.35.0.12 #define CHECK_ENDPOINT_QUEUE(epData)

Value

```
if (epData->queue.len == MAX_QUEUE_SIZE)        \
    {                                               \
        RETURN_SEND_QUEUE_FULL(epData->device);     \
    }
```

sanity check for endpoint queue size

Definition at line 319 of file usbmacros.h.

A.35.0.13 #define CHECK_ENDPOINT_STALL(device, endpointNumber)

Value

```
{                                                                       \
        BOOL stallState;                                                \
        ixUSBIsEndpointStalled(device, endpointNumber, &stallState);    \
                                                                        \
        if (stallState)                                                 \
        {                                                               \
            RETURN_ENDPOINT_STALLED(device);                            \
        }                                                               \
    }
```

sanity check for endpoint stall

Definition at line 300 of file usbmacros.h.

A.35.0.14 #define CHECK_EVENT_MASK(device, eventMask)

Value

```
if ((eventMask & ~(USB_BUS_EVENTS | USB_DEVICE_EVENTS)) != 0) \
    {                                                         \
        RETURN_INVALID_PARMS(device);                         \
    }
```

sanity check for event masks

Definition at line 312 of file usbmacros.h.

A.35.0.15 #define EP_DIRECTION(x) ((x) & (USB_IN | USB_OUT))

Macro used to extract the endpoint direction from an EPDescriptorTable[] entry.

Parameters

x int (in) - the endpoint description entry

Returns

the endpoint direction (USB_IN, USB_OUT or USB_IN_OUT)

Definition at line 80 of file usbmacros.h.

A.35.0.16 #define EP_TYPE(x) ((x) & (USB_CONTROL | USB_BULK | USB_ISOCHRONOUS | USB_INTERRUPT))

Macro used to extract the endpoint type from an EPDescriptorTable[] entry.

Parameters

x int (in) - the endpoint description entry

Returns

the endpoint type (USB_CONTROL, USB_BULK, USB_ISOCHRONOUS, USB_INTERRUPT)

Definition at line 92 of file usbmacros.h.

A.35.0.17 #define MAX(a, b) ((a) > (b)? (a) : (b))

Compares two values and returns the maximum.

Parameters

- *a* - first value
- *b* - second value

Returns

maximum of the two input values

Definition at line 121 of file usbmacros.h.

A.35.0.18 #define MIN(a, b) ((a) < (b)? (a) : (b))

Compares two values and returns the minimum.

Parameters

- *a* - first value
- *b* - second value

Returns

minimum of the two input values

Definition at line 106 of file usbmacros.h.

A.35.0.19 #define QUEUE_WRAP(tail) ((tail) >= (MAX_QUEUE_SIZE) ? ((tail) - (MAX_QUEUE_SIZE)) : (tail))

Adjusts the tail of a queue implemented in a circular buffer by wrapping at the buffer boundary.

Parameters

tail int - virtual tail offset

Returns

the real adjusted tail offset

Definition at line 135 of file usbmacros.h.

A.35.0.20 #define RETURN_ENDPOINT_STALLED(device)

Value

```
device->lastError = IX_USB_ENDPOINT_STALLED;  \
    return IX_FAIL;
```

set IX_USB_ENDPOINT_STALLED on device and return IX_FAIL

Definition at line 248 of file usbmacros.h.

A.35.0.21 #define RETURN_ERROR(device)

Value

```
device->lastError = IX_USB_ERROR;  \
    return IX_FAIL;
```

set IX_USB_ERROR on device and return IX_FAIL

Definition at line 223 of file usbmacros.h.

A.35.0.22 #define RETURN_INVALID_DEVICE(device)

Value

```
device->lastError = IX_USB_INVALID_DEVICE;  \
    return IX_FAIL;
```

set IX_USB_INVALID_PARAMS on device and return IX_FAIL

Definition at line 238 of file usbmacros.h.

A.35.0.23 #define RETURN_INVALID_PARMS(device)

Value

```
device->lastError = IX_USB_INVALID_PARMS;  \
    return IX_FAIL;
```

set IX_USB_INVALID_PARAMS on device and return IX_FAIL

Definition at line 228 of file usbmacros.h.

A.35.0.24 #define RETURN_NO_ENDPOINT(device)

Value

```
device->lastError = IX_USB_NO_ENDPOINT;  \
    return IX_FAIL;
```

set IX_USB_INVALID_PARAMS on device and return IX_FAIL

Definition at line 243 of file usbmacros.h.

A.35.0.25 #define RETURN_NO_IN_CAPABILITY(device)

Value

```
device->lastError = IX_USB_NO_IN_CAPABILITY; \
    return IX_FAIL;
```

set IX_USB_NO_IN_CAPABILITY on device and return IX_FAIL

Definition at line 258 of file usbmacros.h.

A.35.0.26 #define RETURN_NO_PERMISSION(device)

Value

```
device->lastError = IX_USB_NO_PERMISSION; \
    return IX_FAIL;
```

set IX_USB_NO_PERMISSION on device and return IX_FAIL

Definition at line 268 of file usbmacros.h.

A.35.0.27 #define RETURN_NO_STALL_CAPABILITY(device)

Value

```
device->lastError = IX_USB_NO_STALL_CAPABILITY; \
    return IX_FAIL;
```

set IX_USB_NO_STALL_CAPABILITY on device and return IX_FAIL

Definition at line 263 of file usbmacros.h.

A.35.0.28 #define RETURN_OK(device)

Value

```
device->lastError = IX_SUCCESS; \
    return IX_SUCCESS;
```

set IX_SUCCESS on device and return IX_SUCCESS

Definition at line 218 of file usbmacros.h.

A.35.0.29 #define RETURN_REDUNDANT(device)

Value

```
device->lastError = IX_USB_REDUNDANT;  \
    return IX_FAIL;
```

set IX_USB_REDUNDANT on device and return IX_FAIL

Definition at line 233 of file usbmacros.h.

A.35.0.30 #define RETURN_SEND_QUEUE_FULL(device)

Value

```
device->lastError = IX_USB_SEND_QUEUE_FULL;  \
    return IX_FAIL;
```

set IX_USB_SEND_QUEUE_FULL on device and return IX_FAIL

Definition at line 253 of file usbmacros.h.

A.35.0.31 Enumeration Type Documentation

A.35.0.32 enum USBEndpointType

Note: the values are set for compatibility with USBEndpointDirection. These are not standard USB endpoint types to be used in descriptors. (See usbstd.h .)

Definition at line 75 of file usbconstants.h.

A.35.0.33 enum USBEventMap

USB Event Map.

Enumeration Values

USB_SOF Start Of Frame.

Definition at line 86 of file usbconstants.h.

A.35.0.34 Function Documentation

A.35.0.35 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) - sumbitted 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.35.0.36 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.35.0.37 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 and **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.35.0.38 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 interations 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. A check to verify that the USB device is present is performed and a warning is issued if the device is not present.

Warning: This function is not reentrant.

Returns

IX_SUCCESS if the initialization was successful; a warning is issued if the specified USB device is not present. IX_FAIL otherwise,in which case a detailed error code will be set in the *lastError* field, unless the *device* parameter is NULL.

A.35.0.39 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.35.0.40 PUBLIC void ixUSBEndpointInfoShow (USBDevice * *device*)

Display endpoint information table.

Parameters

device USBDevice * (in) - a structure identifying the device

Returns

none

A.35.0.41 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 and **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.35.0.42 PUBLIC const char * ixUSBErrorStringGet (UINT32 *errorCode*)

Convert an error code into a human-readable string error message.

Parameters

errorCode UINT32 (in) - erorr code as defined in usberrors.h

Returns

a `const char *` pointer to the error message

A.35.0.43 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.35.0.44 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.35.0.45 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*) or **false** otherwise**true**

Returns

IX_SUCCESS or IX_FAIL if the device pointer is invalid or the endpoint doesn't exist

A.35.0.46 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.35.0.47 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.35.0.48 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.35.0.49 PUBLIC IX_STATUS ixUSBStatisticsShow (USBDevice * *device*)

Display device state and statistics.

Parameters

device USBDevice * (in) - a structure identifying the device

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.

Codelet Reference B

This appendix documents the codelets for the Intel® IXP400 Software v.1.3's access-layer APIs.

This reference chapter was generated automatically using the most recent source code available when this document was generated, and is useful for reference purposes. Be advised that the software may have been subsequently updated and the user should use the source code for the most accurate reference information.

B.1 Codelet Reference Index

Codelet Reference Index . . . 625
IXP425 ATM Codelet (IxAtmCodelet) API . . . 626
Defines . . . 626
Functions . . . 627
Detailed Description . . . 627
Define Documentation . . . 631
Function Documentation . . . 631
IXP425 DMA Access Codelet (IxDmaAccCodelet) API . . . 632
Defines . . . 632
Functions . . . 632
Detailed Description . . . 632
Define Documentation . . . 633
Function Documentation . . . 633
IXP425 Ethenet Aal5 (IxEthAal5App) API . . . 633
Defines . . . 634
Functions . . . 635
Detailed Description . . . 635
Define Documentation . . . 639
Function Documentation . . . 640
IXP425 Ethernet Access Codelet (IxEthAccCodelet) API . . . 641
Defines . . . 641
Functions . . . 642
Detailed Description . . . 642
Define Documentation . . . 644
Function Documentation . . . 645
IXP425 Fast Path Access Codelet (IxFpathAccCodelet) API . . . 646
IXP425 HSS Access Codelet (IxHssAccCodelet) API . . . 654
Defines . . . 654
Functions . . . 654
Detailed Description . . . 654
IXP425 PerfProf Access Codelet . . . 656
Defines . . . 656
Enumerations . . . 656
Functions . . . 656
Detailed Description . . . 657

Define Documentation 658
Enumeration Type Documentation 658
IXP425 Timers (IxTimersCodelet) API 659
Typedefs 659
Enumerations 659
Functions 659
Detailed Description 660
Typedef Documentation 660
Enumeration Type Documentation 660
Function Documentation 661
IXP425 USB RNDIS Codelet (IxUSBRNDIS) API 663
Modules 663
Functions 663
Detailed Description 664
Function Documentation 665
IXP425 USB RNDIS End Driver Codelet (IxUSBRNDIS) API 666
Defines 667
Functions 667
Detailed Description 667
Function Documentation 667
IXP425 USB RNDIS Vendor Codelet (IxUSBRNDIS) 668
Defines 668
Detailed Description 668
IXP425 Crypto Access Codelet 668
Defines 668
Enumerations 670
Functions 670
Detailed Description 670
Define Documentation 672
Enumeration Type Documentation 674
Function Documentation 674

B.2 IXP425 ATM Codelet (IxAtmCodelet) API

IXP425 ATM Codelet (IxAtmCodelet) APIIXP425 ATM Codelet (IxAtmCodelet) APIIXP425 ATM Codelet (IxAtmCodelet) APIIXP425 ATM Codelet (IxAtmCodelet) APIThis 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.

B.2.1 Defines

- #define IX_ATM_CODELET_SWLOOPBACK_PORT_RATE 1962
  Port rate for Software Loopback.
- #define IX_ATM_CODELET_REMOTELOOPBACK_PORT_RATE 1962
  Port rate for Remote Looopback.

- #define IX_ATMCODELET_START_VPI (1)
  The first VPI value.
- #define IX_ATMCODELET_START_VCI (32)
  The first VCI value.
- #define IX_ATMCODELET_NUM_AAL5_CHANNELS 32
  32 Channels for AAL5
- #define IX_ATMCODELET_NUM_AAL0_48_CHANNELS 32
  32 Channels for AAL0 48-bytes
- #define IX_ATMCODELET_NUM_AAL0_52_CHANNELS 32
  32 Channels for AAL0 52-bytes

B.2.2 Functions

- PUBLIC IX_STATUS ixAtmCodeletMain (IxAtmCodeletMode modeType, IxAtmCodeletAalType aalType)
  This function is used as a single point of execution for ATM codelet.

B.2.3 Detailed Description

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.

This codelet also 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.

Disclaimer Note

For Linux Platform

- When 'insmod' the ATM codelet object, it will begin to send AAL packets and display the transmit and receive statistics every 15 second
- Unable to 'rmmod'. User will not be able to type anything in the command line due to: a) the return carriage indicating that the task is sending AAL packets, and b) the failure to kill the thread which is used to transmit AAL packets

VxWorks User Guide

ixAtmCodeletMain() function is used as a single point of execution for Atm Codelet, which allows the user to enter selections for different type of modes and AAL type. The function also allows the user to execute OAM ping either in UTOPIA or Software Loopback mode. In all modes, the transmit and receive statistics will be displayed every 15secs.

Usage : ixAtmCodeletMain (modeType, aalType) modeType: 0 = Utopia Loopback Mode 1 = Software Loopback Mode 2 = Remote Loopback Mode 3 = F4 & F5 cells OAM Ping in UTOPIA Loopback mode 4 = F4 & F5 cells OAM Ping in Software Loopback mode aalType: 1 = AAL5 2 = AAL0_48 3 = AAL0_52

Linux User Guide

The idea of using the ixAtmCodeletMain() as a single point of execution for ATM codelet is similar in VxWorks User Guide. It also allows user to execute OAM ping. Similarly, all modes will display the transmit and receive statistics every 15secs. This function will be executed when user issue'insmod' in command prompt

Usage : # insmod csr_codelets_atm.o modeType= aalType= Where x: 0 = Utopia Loopback Mode 1 = Software Loopback Mode 2 = Remote Loopback Mode 3 = F4 & F5 cells OAM Ping in UTOPIA Loopback mode 4 = F4 & F5 cells OAM Ping in Software Loopback modeWhere y: 1 = AAL5 2 = AAL0_48 3 = AAL0_52

Note for VxWorks and Unix Usage

- IX_ATM_CODELET_SWLOOPBACK_PORT_RATE and IX_ATM_CODELET_REMOTELOOPBACK_PORT_RATE defined in this header file allows the user to change the port rate (in cells/sec) accordingly. The port rate works when using ADSL connection. The default port rate for both loopback is set to 1962cells/sec (~832kbps)
- IX_ATMCODELET_START_VPI and IX_ATMCODELET_START_VCI defined in this header file can be modified by the user. By default, VPI and VCI are set to 1 and 32 respectively
- IX_ATMCODELET_NUM_AAL5_CHANNELS, IX_ATMCODELET_NUM_AAL0_48_CHANNELS, and IX_ATMCODELET_NUM_AAL0_52_CHANNELS can be changed by the user. By default, they are set to 32 channels
- OAM Ping in UTOPIA Loopback mode will perform the following sequence in forever loop: i) Send AAL packets, ii) Display the transmit and receive statistics, and iii) Perform OAM Ping F4 and F5 (ETE and Segment) and display OAM statistics
- OAM Ping in Software Loopback will perform the following sequence in forever loop: i) Display the transmit and receive statistics, and ii) OAM Ping F4 and F5 (ETE and Segement) and display OAM Statistics

ATM Features

- An interface is provided to setup Aal5 or Aal0 (48 or 52 bytes) Transmit and Recieve VCs on one port. Only UBR VCs can be setup. When a channel is setup using this interface both a Transmit and a Recieve VC is created.
- An interface is provided to remove all registered Aal5/Aal0 VCs.
- Both remote and local loopback of Aal5 or Aal0 is provided by this codelet. Local loopback refers to loopback provided by the UTOPIA interface. In local loopback packets generated by the IXP425 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 performa 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 Recieve 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.

IXP425 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.

IXP425 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

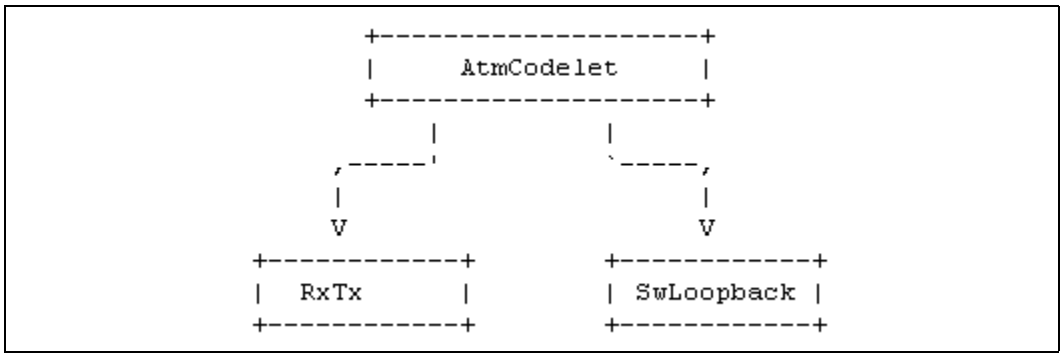
IxAtmCodelet modes of operation

This codelet can be initialised 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 interupts. This means that TxDone/RxLo will be perfomed 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 IXP425 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.

Figure 82. IxAtmCodelet sub-components



AtmCodelet — This implements the API functions.

RxTx — 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 intefaces used internally for getting and returning vxWorks mbufs.

UTOPIA Recieve 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.

OAM 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 independant of sending responses to parent loopback cells received.
- An interface is provided to query OAM traffic statistics.

OAM uses the following IXP425 ATM Components

- 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.
- AtmUtils. This codelet provides VC setup facilities using IxAtmdAcc.

Note: If validation board is used (instead of using IXDP425 development board), then uncomment VALIDATION_PLATFORM_USED flag in component.mk

B.2.4 Define Documentation

- #define IX_ATM_CODELET_REMOTELOOPBACK_PORT_RATE 1962
  Port rate for Remote Looopback.
  By default the port rate for remote loopback is set at PCR = 1962 cells/sec (832kbps)
  Definition at line 341 of file IxAtmCodelet.h.
- #define IX_ATM_CODELET_SWLOOPBACK_PORT_RATE 1962
  Port rate for Software Loopback.
  By default the port rate for software loopback is set at PCR = 1962 cells/sec (832kbps)
  Definition at line 328 of file IxAtmCodelet.h.
- #define IX_ATMCODELET_NUM_AAL0_48_CHANNELS 32
  32 Channels for AAL0 48-bytes
  Definition at line 381 of file IxAtmCodelet.h.
- #define IX_ATMCODELET_NUM_AAL0_52_CHANNELS 32
  32 Channels for AAL0 52-bytes
  Definition at line 390 of file IxAtmCodelet.h.
- #define IX_ATMCODELET_NUM_AAL5_CHANNELS 32
  32 Channels for AAL5
  Definition at line 372 of file IxAtmCodelet.h.
- #define IX_ATMCODELET_START_VCI (32)
  The first VCI value.
  By default the VCI is set to 2
  Definition at line 363 of file IxAtmCodelet.h.
- #define IX_ATMCODELET_START_VPI (1)
  The first VPI value.
  By default the VPI is set to 1
  Definition at line 352 of file IxAtmCodelet.h.

B.2.5 Function Documentation

- ixAtmCodeletMain (IxAtmCodeletMode *modeType*, IxAtmCodeletAalType *aalType*)
  This function is used as a single point of execution for ATM codelet.

Parameters

- *IxAtmCodeletMode modeType* The type of mode use, either Utopia, Software or Remote loopback mode. It also consists of OAM Ping in Utopia or Software loopback mode.
- *IxAtmCodeletAalType aalType* The type of AAL: AAL5 or AAL0 with 48- or 52-bytes cell

Returns

- IX_SUCCESS Mode and AAL type successfully setup IX_FAIL Invalid AAL or mode type, or error in setting up the modes

B.3 IXP425 DMA Access Codelet (IxDmaAccCodelet) API

IXP425 DMA Access Codelet (IxDmaAccCodelet) APIIXP425 DMA Access Codelet (IxDmaAccCodelet) APIIXP425 DMA Access Codelet (IxDmaAccCodelet) APIIXP425 DMA Access Codelet (IxDmaAccCodelet) APIIXP425 DMA Access component API.

B.3.1 Defines

- #define IX_DMA_CODELET_TRANSFER_LENGTH 128
  The length of the transfer size if 128 bytes.

B.3.2 Functions

- IX_STATUS ixDmaAccCodeletMain (void)
  This function is the entry point to the Dma Access codelet. It will initialise the Dma codelet which in turn initialises the necessary components.

B.3.3 Detailed Description

IXP425 DMA Access component API.

This file contains a main interface of the Dma Access Codelet that initialises the DmaAcc codelet and execute Dma transfer using ixDmaAccCodeletTestPerform() function for various DMA transfer mode, addressing mode and transfer width. The block size used in this codelet are 8,1024,16384,32768,65528 bytes. For each Dma configuration, the performance will be measured and the average rate (in Mbps) will be displayed

VxWorks User Guide

Usage : -> *ixDmaAccCodeletMain()*

Note: 1. Once the function is executed, the codelet will display the results

2. The formulae to calculate the rate is:

*Rate (in Mbps) = ((length * 8) / (ticks / 66))*

Linux User Guide

Usage : *# insmod csr_codelets_dmaAcc.o*

Notes:

- Once the function is executed, the codelet will display the results
- The formulae to calculate the rate is:

*Rate (in Mbps) = ((length * 8) / (ticks / 66))*

DmaAcc Codelet Features

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 Addressess 2. Fixed Source to Incremental Destination Addressess 3. Incremental Source to Fixed Destination Addressess

DMA Transfer Widths 1. 32-bit Transfer 2. 16-bit Transfer 3. 8-bit Transfer 4. Burst Transfer

NOTE : The user must initialise the system with ixDmaAccCodeletInit prior to calling the function ixDmaAccCodeletiTestPerform()

Performance will execute PERFORMANCE_NUM_LOOP (i.e. 100 runs) in order to calculate the average rate for each Dma transfer configuration

B.3.4 Define Documentation

- #define IX_DMA_CODELET_TRANSFER_LENGTH 128

  The length of the transfer size if 128 bytes.

  It can be changed for Dma transfer. The range is between 1-65535 bytes

  Definition at line 143 of file IxDmaAccCodelet.h.

B.3.5 Function Documentation

- void ixDmaAccCodeletMain (void)

  This function is the entry point to the Dma Access codelet. It will initialise the Dma codelet which in turn initialises the necessary components.

  Once it has successfully initialise the Dma Codelet, this function will continue to perform valid DMA transfer using IxDmaAccCodeletTestPerform()

Parameters

none

Returns

none

B.4 IXP425 Ethenet Aal5 (IxEthAal5App) API

IXP425 Ethenet Aal5 (IxEthAal5App) APIIXP425 Ethenet Aal5 (IxEthAal5App) APIIXP425 Ethenet Aal5 (IxEthAal5App) APIIXP425 Ethenet Aal5 (IxEthAal5App) APIIXP425 Ethernet Aal5 Codelet component API.

B.4.1 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
- #define IX_EAA_MAC1 00
  Define default for MAC1 address for ixEAAAddMac() function.
- #define IX_EAA_MAC2 00
  Define default for MAC2 address for ixEAAAddMac() function.
- #define IX_EAA_MAC3 00
  Define default for MAC3 address for ixEAAAddMac() function.
- #define IX_EAA_MAC4 00
  Define default for MAC4 address for ixEAAAddMac() function.
- #define IX_EAA_MAC5 03
  Define default for MAC5 address for ixEAAAddMac() function.
- #define IX_EAA_MAC6 01
  Define default for MAC6 address for ixEAAAddMac() function.

- #define IX_EAA_PORT0 0
  Define default for Port number for ixEAAAddMac() function.

B.4.2 Functions

- PUBLIC IX_STATUS ixEthAal5AppCodeletMain (IxEAAModeType modeType)
  This is the main function that executes the EthAal5App codelet.

B.4.3 Detailed Description

IXP425 Ethernet Aal5 Codelet component API.

- IxEthAal5App application is also called as IXP425 Mini Bridge application which bridges traffic between Ethernet and Utopia ports or Ethernet and ADSL ports. It uses ixEthAcc,ixAtmdAcc, ixAtmm, ixAtmSch and ixQmgr software components.

VxWorks User Guide

ixEthAal5AppCodeletMain() function is used as a single point of execution for EthAal5 Codelet, which allows the user to enter in 2 type of modes: Utopia or ADSL, in order to operate together with ethernet.

Usage : ixEthAal5AppCodeletMain (modeType) modeType: 1 = Utopia 2 = ADSL

Linux User Guide

ixEthAal5AppCodeletMain() function is used as a single point of execution for EthAal5 Codelet, which allows the user to enter in 2 type of modes: Utopia or ADSL, in order to operate together with ethernet.

Usage : # insmod csr_codelets_ethAal5App.o modeType= Where x: 1 = Utopia 2 = ADSL

VxWorks and Linux Usage

In order to observe the current traffic counters, the ixEAAShow() function is executed every 15seconds. This applys to both Utopia and ADSL mode.

Features

- This Application currently supports 2 Ethernet ports and up to 8 Utopia phys,which are initialized by default at the start of application. Ethernet frames are transferred across ATM link (through Utopia interface) using AAL5 protocol and Ethernet frame encapsulation described by RFC 1483. MAC address learning is performed on Ethernet frames, received by Ethernet ports and ATM interface (encapsulated). Application filters packets basing on destination MAC addresses - packets are forwarded to other port only if the port has ever received packet/frame with the same source MAC address. Forwarding is done only between Ethernet and Utopia port. Two simplifications were made to keep code simple:
- Application doesn't allow packet forwarding between Ethernet ports (nor Utopia ports).
- flooding (forwarding frames/packets with unknown MAC addresses) is not supported. Two IxEthAal5App will never transfer any packets between each other, because initialy MAC data base is empty, so all packets will be filtered out. However there is function ixEAAAddMAC

which can be used to add MAC address to the data base and assign it to one of available ports. To enable simplified flooding see comments in ixEAAEthRxCallback.

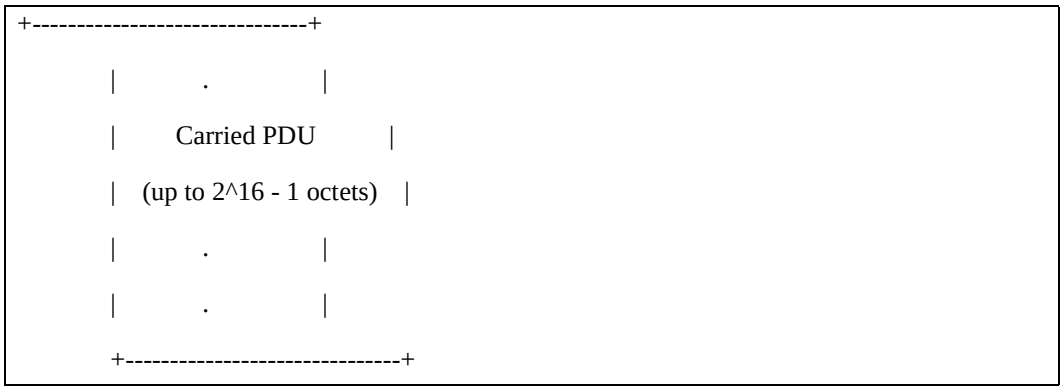
- This application can not be executed more than once. It doesn't deinitialize itself. If user wishes to change configuration and run application again the whole system (vxWorks) must be restarted.
- Code was written for big-endian mode. Current version will not work if XScale switched to little-endian mode.
- currently Mac Learning/Filtering database in ixEthAcc component supports only Ethernet ports. For that reason it couldn't be used in this application for learning Mac addresses from encapsulated Ethernet frames received from Utopia. In the near future ixEthAcc component will support all possible ports (including Utopia), but by this time a very simplified approach is used in this application: only one Mac address is stored per VC (and there is one VC per Phy). It means, that only one Mac address is supported simultaneously per Phy. This is done to keep code as simple as possible.
- This application doesn't initialize any DSL connection. It initializes Utopia interface - if any DSL card is attached to the Richfield board it should be initialized before start of application (e.g. to initialize Adsl card call 'adslUp' function from windshell - adsl module must be separately loaded).
- 2 protocols from RFC 1483 are recognized: The first packet received from ATM will decide the behaviour of the application (ether bridged or routed)

From RFC 1483

- VC Based Multiplexing of Routed Protocols

PDUs of routed protocols shall be carried as such in the Payload of the AAL5 CPCS-PDU. The format of the AAL5 CPCS-PDU Payload field thus becomes:

Figure 83. Payload Format for Routed PDUs



VC Based Multiplexing of Bridged Protocols

PDUs of bridged protocols shall be carried in the Payload of the AAL5 CPCS-PDU.

Figure 84. Payload Format for Bridged Ethernet/802.3 PDUs

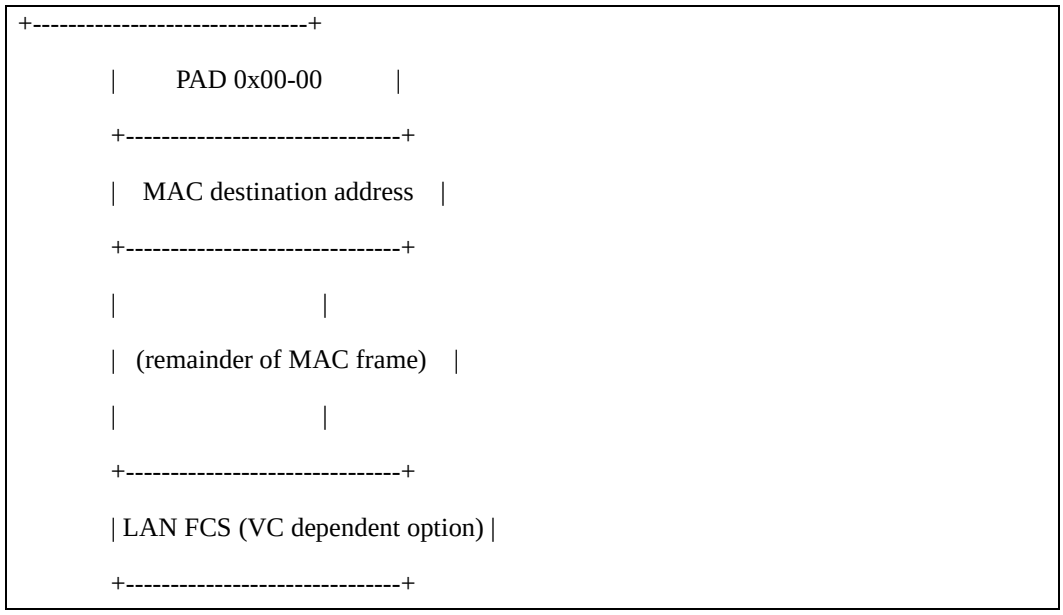


Figure 85. EthAal5 Codelet Data Flow

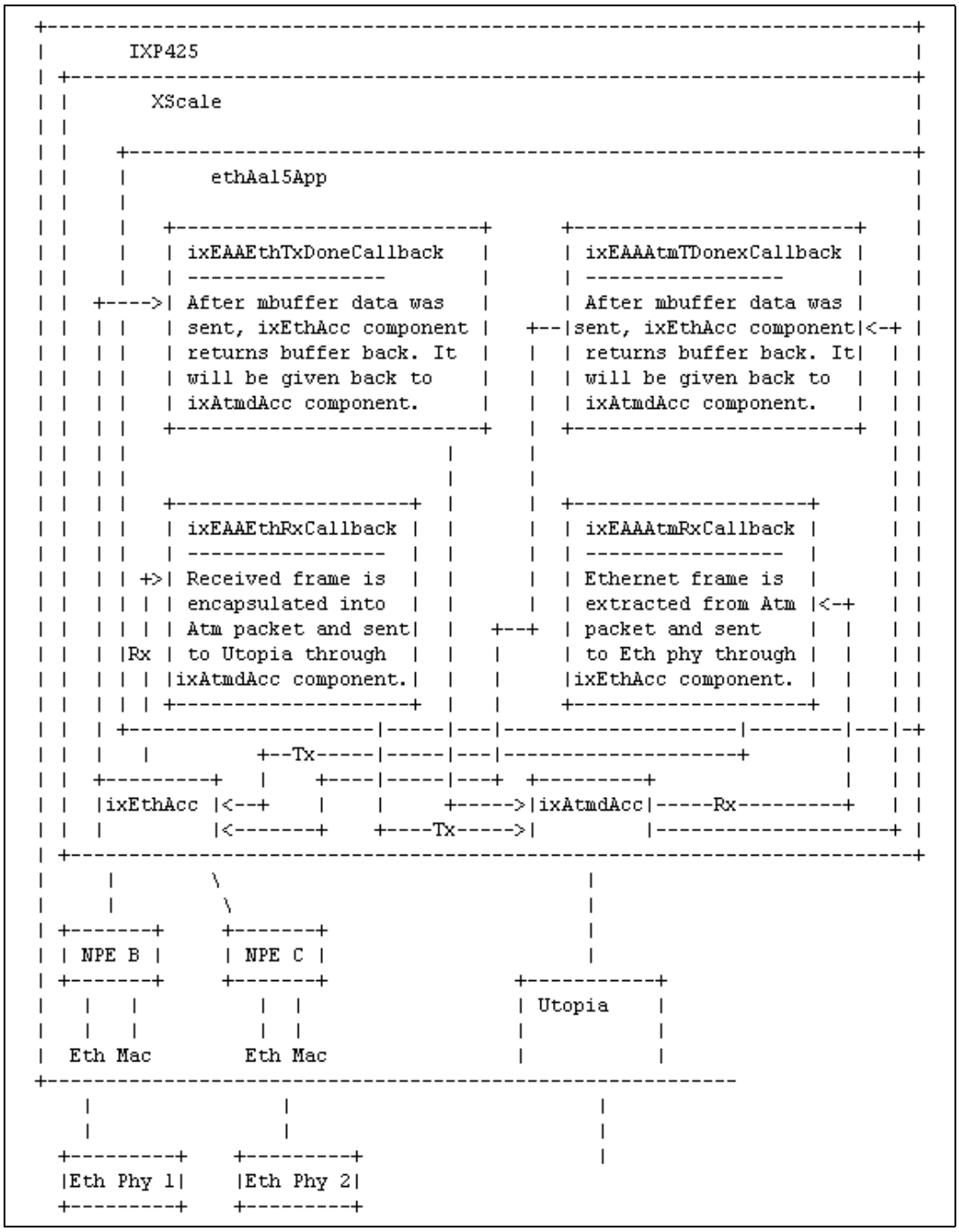
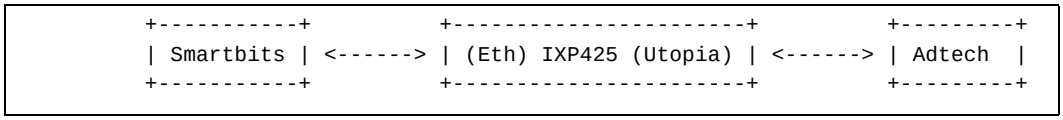


Figure 86. Configuration Example



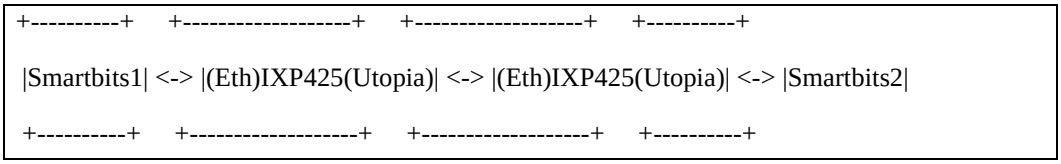
On smartbits set: dst. mac = MAC1 and src mac = MAC2

On Adtech add AAL5 with eth. encapsulation, dst. mac = MAC2, src. MAC=MAC1

Dependently which Phy is used Adtech must use same Vpi/Vci address as assigned to Phy by ethAal5App. By default for Phy 0 it is 1/100 (Vpi/Vci), for Phy 1 2/100 and so on.

Another option is:

Figure 87. Configuration Example Number 2



On smartbits1 set: dst. mac = MAC1 and src mac = MAC2

On smartbits2 set: dst. mac = MAC2 and src mac = MAC1

on IXP425 connected to Smartbits 1 execute in windshell:

ixEAAAddMAC(1, MAC1[0], MAC1[1], ..., MAC1[5]);

We use port 1 for Adsl card, however it may change in the future.

IMPORTANT!!!

If validation board is used (instead of using IXDP425 development board), then uncomment VALIDATION_PLATFORM_USED flag in component.mk

B.4.4 Define Documentation

- #define IX_EAA_MAC1 00
  Define default for MAC1 address for ixEAAAddMac() function.
  Definition at line 374 of file IxEthAal5App.h.
- #define IX_EAA_MAC2 00
  Define default for MAC2 address for ixEAAAddMac() function.
  Definition at line 380 of file IxEthAal5App.h.
- #define IX_EAA_MAC3 00
  Define default for MAC3 address for ixEAAAddMac() function.
  Definition at line 386 of file IxEthAal5App.h.
- #define IX_EAA_MAC4 00
  Define default for MAC4 address for ixEAAAddMac() function.
  Definition at line 392 of file IxEthAal5App.h.
- #define IX_EAA_MAC5 03
  Define default for MAC5 address for ixEAAAddMac() function.
  Definition at line 399 of file IxEthAal5App.h.
- #define IX_EAA_MAC6 01

Define default for MAC6 address for ixEAAAddMac() function.
Definition at line 405 of file IxEthAal5App.h.

- #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
  Definition at line 298 of file IxEthAal5App.h.
- #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.
  Definition at line 283 of file IxEthAal5App.h.
- #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
  Definition at line 311 of file IxEthAal5App.h.
- #define IX_EAA_PORT0 0
  Define default for Port number for ixEAAAddMac() function.
  Definition at line 411 of file IxEthAal5App.h.
- #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.
  Definition at line 321 of file IxEthAal5App.h.

B.4.5 Function Documentation

- ixEthAal5AppCodeletMain (IxEAAModeType *modeType*)
  This is the main function that executes the EthAal5App codelet.
  It first calls IxEAAMain() function which initialize the MAC database, to be an in valid Mac addresses (i.e. contain 0xffs), 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
  After which the main has been called, this ixEthAal5AppCodeletMain() function will add the MAC address using ixEAAAddMAC() function. For a single PHY utopia mode, only one port will be setup a single MAC address. However, if multiple phy is used, 8 ports will be setup and each port is assigned with a unique MAC addresses.
  For single utopia phy. The following is setup using ixEAAAddMAC() function

ixEAAAddMAC(IX_EAA_PORT0,

IX_EAA_MAC1,

IX_EAA_MAC2,

IX_EAA_MAC3,

IX_EAA_MAC4,

IX_EAA_MAC5,

IX_EAA_MAC6);

For multi phy utopia the port number and MAC6 increments using a for-loop

For (port = 0, nextMac = 0; port < IX_EAA_NUM_ATM_PORTS; port++, nextMac++)

{

ixEAAAddMAC(IX_EAA_PORT0 + port,

IX_EAA_MAC1,

IX_EAA_MAC2,

IX_EAA_MAC3,

IX_EAA_MAC4,

IX_EAA_MAC5,

IX_EAA_MAC6 + nextMac);

}

Lastly, ixEthAal5AppCodeletMain() creates a thread which purposed to display the EthAal5App codelet counter every 15secs

B.5 IXP425 Ethernet Access Codelet (IxEthAccCodelet) API

IXP425 Ethernet Access Codelet (IxEthAccCodelet) APIIXP425 Ethernet Access Codelet (IxEthAccCodelet) APIIXP425 Ethernet Access Codelet (IxEthAccCodelet) APIIXP425 Ethernet Access Codelet (IxEthAccCodelet) APIIXP425 Ethernet Access Codelet API.

B.5.1 Defines

- #define IX_ETHACC_CODELET_NPEB_MAC {{0x2, 0x0, 0xa, 0xb, 0xc, 0xd}}
  Hard-encoded MAC address for NPEB.
- #define IX_ETHACC_CODELET_NPEC_MAC {{0x2, 0x0, 0xe, 0xf, 0xa, 0xb}}
  Hard-encoded MAC address for NPEC.
- #define IX_ETHACC_CODELET_RX_MBUF_POOL_SIZE 128
  Size of receive MBuf pool.
- #define IX_ETHACC_CODELET_TX_MBUF_POOL_SIZE 128
  Size of transmit MBuf pool.

- #define IX_ETHACC_CODELET_MAX_PORT IX_ETH_ACC_NUMBER_OF_PORTS
  Number of Ethernet Ports supported for this codelet.
- #define IX_ETHACC_CODELET_MBUF_POOL_SIZE
  Size of MBuf pool.
- #define IX_ETHACC_CODELET_PCK_SIZE IX_ETHACC_RX_MBUF_MIN_SIZE
  Size of MBuf packet.
- #define IX_ETHACC_CODELET_PCK_LEN 1536
  Length of MBuf payload (in bytes).
- #define IX_ETHACC_CODELET_MBUF_DATA_POOL_SIZE
  (IX_ETHACC_CODELET_MBUF_POOL_SIZE * IX_ETHACC_CODELET_PCK_SIZE)
  Size of MBuf data pool.
- #define IX_ETHACC_CODELET_TXGEN_PCK_LEN 60
  Size of packets for TxGenRxSink Operation.
- #define IX_ETHACC_CODELET_TXGEN_PCKS 128
  Number of packets to generate for the TxGenRxSink Operation.
- #define IX_ETHACC_CODELET_RX_FCS_STRIP
  Strip FCS from incoming frames. To undefine, change to #undef.

B.5.2 Functions

- PUBLIC IX_STATUS ixEthAccCodeletMain (IxEthAccCodeletOperation operationType)
  This function is used as a single point of execution for EthAcc codelet.

B.5.3 Detailed Description

IXP425 Ethernet Access Codelet API.

This codelet demonstrates both Ethernet Data and Control plane services and Ethernet Management services.

- A) Ethernet Data and Control plane services:
- Configuring both ports as a receiver sink from an external source (such as Smartbits).
- Configuring Port-1 to automatically transmit frames and receive frames on Port-2. Frames generated and transmitted in Port-1 are loopbacked into Port-2 by using cross cable.
- 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.
- B) Ethernet Management services:
- 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.

Definition

In the context of this codelet, the following definitions are applicable.

Port 1 = ixe0 = Ethernet port associated with NPE-B Ethernet Coprocessor.

Port 2 = ixe1 = Ethernet port associated with NPE-C Ethernet Coprocessor.

Design Constraints

This codelet assumes that the underlying IXP425 Product Line Silicons have two Ethernet NPEs. For silicon with single Ethernet NPE, operation will be only functional in the particular Ethernet port that corresponds to the available Ethernet NPE. Particularly, bridge operation will not work as two Ethernet ports are needed in this operation.

Assumptions

This codelet illustrates the use EthAcc APIs. The operations provided may not be working on the best performance as the target of this codelet is just to show the functionality of APIs. In order to get better performance, #undef IX_ETHACC_CODELET_TXGENRXSINK_VERIFY to disable traffic verification.

Please note that this codelet is not optimized for production quality.

For performance testing, please use the operations below:

- Rx Sink Operation.
- TxGenRxSink Operation.
- Bridge Operation with Ethernet frames sent into either one of the Ethernet Ports.

The operations below need special tuning to optimize them. Tuning can be done by either using a lower traffic(frames/second), reducing the value of IX_ETHACC_CODELET_TXGEN_PCKS or #undef IX_ETHACC_CODELET_TXGENRXSINK_VERIFY.

- Software Loopback Operation.
- PHY Loopback Operation.
- Bridge Operation with Ethernet frames sent into both Ethernet Ports.

VxWorks User Guide

ixEthAccCodeletMain() function is used as a single point of execution for EthAcc Codelet. It allows user to enter selection for different type of supported operations described below:

Usage : >ixEthAccCodeletMain (operationType) Where operationType: 1 = To sink received frames as fast as possible for available ports. 2 = To software loopback received frames to the same port for available ports. 3 = To generate and transmit frames from port 1, remote loopback by using an external cross cable to port 2, and received on port 2 (TxGenRxSink). 4 = To generate frames and perform PHY loopback on the same port for available ports. 5 = To transmit any frame received on one port through the other one (Bridge). 6 = To activate Ethernet MAC learning facility.

Linux User Guide

The idea of using the ixEthAccCodeletMain() as a single point of execution for EthAcc codelet. The operation selected will be executed when user issue 'insmod' in command prompt.

Usage : >insmod csr_codelets_ethAcc.o operationType= Where x: 1 = To sink received frames as fast as possible for available ports. 2 = To software loopback received frames to the same port for available ports. 3 = To generate and transmit frames from port 1, remote loopback by using an external cross cable to port 2, and received on port 2 (TxGenRxSink). 4 = To generate frames and perform PHY loopback on the same port for available ports. 5 = To transmit any frame received on one port through the other one (Bridge). 6 = To activate Ethernet MAC learning facility.

MAC Setup

The default MAC setup will be:

- Promiscuous mode enabled (for learning example)
- Frame Check Sequence appended for all frames generated on the XScale

PHY Setup

This codelet uses two PHYs as defined by IX_ETHACC_CODELET_MAX_PHY The default PHY setup will be:

- 100Mbits,
- full duplex,
- auto-negotiation on.

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.

B.5.4 Define Documentation

- #define IX_ETHACC_CODELET_MAX_PORT IX_ETH_ACC_NUMBER_OF_PORTS
  Number of Ethernet Ports supported for this codelet.
  Definition at line 246 of file IxEthAccCodelet.h.
- #define IX_ETHACC_CODELET_MBUF_DATA_POOL_SIZE (IX_ETHACC_CODELET_MBUF_POOL_SIZE * IX_ETHACC_CODELET_PCK_SIZE)
  Size of MBuf data pool.
  Definition at line 284 of file IxEthAccCodelet.h.
- #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 pool.

Definition at line 255 of file IxEthAccCodelet.h.

- #define IX_ETHACC_CODELET_NPEB_MAC {{0x2, 0x0, 0xa, 0xb, 0xc, 0xd}}
  Hard-encoded MAC address for NPEB.
  Definition at line 210 of file IxEthAccCodelet.h.
- #define IX_ETHACC_CODELET_NPEC_MAC {{0x2, 0x0, 0xe, 0xf, 0xa, 0xb}}
  Hard-encoded MAC address for NPEC.
  Definition at line 219 of file IxEthAccCodelet.h.
- #define IX_ETHACC_CODELET_PCK_LEN 1536
  Length of MBuf payload (in bytes).
  Definition at line 275 of file IxEthAccCodelet.h.
- #define IX_ETHACC_CODELET_PCK_SIZE IX_ETHACC_RX_MBUF_MIN_SIZE
  Size of MBuf packet.
  Definition at line 266 of file IxEthAccCodelet.h.
- #define IX_ETHACC_CODELET_RX_FCS_STRIP
  Strip FCS from incoming frames. To undefine, change to #undef.
- #define IX_ETHACC_CODELET_RX_MBUF_POOL_SIZE 128
  Size of receive MBuf pool.
  Definition at line 228 of file IxEthAccCodelet.h.
- #define IX_ETHACC_CODELET_TX_MBUF_POOL_SIZE 128
  Size of transmit MBuf pool.
  Definition at line 237 of file IxEthAccCodelet.h.
- #define IX_ETHACC_CODELET_TXGEN_PCK_LEN 60
  Size of packets for TxGenRxSink Operation.
  Definition at line 294 of file IxEthAccCodelet.h.
- #define IX_ETHACC_CODELET_TXGEN_PCKS 128
  Number of packets to generate for the TxGenRxSink Operation.
  Definition at line 303 of file IxEthAccCodelet.h.

B.5.5 Function Documentation

- ixEthAccCodeletMain (IxEthAccCodeletOperation *operationType*)
  This function is used as a single point of execution for EthAcc codelet.

Parameters

- *IxEthAccCodeletOperation [in] operationType* - The type of operation to be executed. Refer to the descriptions above.

Returns

- IX_SUCCESS : If operation selected is successfully setup
- IX_FAIL : If operation selected fails to be setup.

B.6 IXP425 Fast Path Access Codelet (IxFpathAccCodelet) API

IXP425 Fast Path Access Codelet (IxFpathAccCodelet) APIIXP425 Fast Path Access Codelet (IxFpathAccCodelet) APIIXP425 Fast Path Access Codelet (IxFpathAccCodelet) APIIXP425 Fast Path Access Codelet (IxFpathAccCodelet) APIIXP425 codelet Fast Path Access component API. This module contains the implementation of the FPATH Access Codelet.

The following top-level scenarios are supported:

- Provide an Ethernet Bridge fast path codelet 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 codelet of the Wan Mac address learning functionality.
- The Utopia port is connected to a piece of test equipment capabale of generating encapsulated ethernet Frames.
- The Ethernet port is connected to a piece of test equipment capabale 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 fastpath codelet 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 codelet of how IP Flow detection and aging can be implemented.
- The Utopia port is connected to a piece of test equipment capabale of generating encapsulated IP packets.
- The Ethernet port is connected to a piece of test equipment capabale 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 codelet 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 traveling 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 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 upto 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.

Sequence on Starting the Fastpath Codelet

1. Reboot the board under test.
2. Connect Adtech Test Equipment to the Utopia port
3. Load the scripts to configure Adtech as per the configuration setting defined in the Fastpath Ethernet bridge Codelet Flow / Fastpath Routed IP Codelet Flow sub section.
4. Start the Adtech Ethernet Analyzer Card.
5. Do one of the following:

- For VxWorks build, Call ixFpAccCodeletMain(0) to start the Ethernet Bridge codelet or ixFpCodeletMain(1) to start the Routed IP codelet.
- For Linux build, insmod ixFpAccCodeletMain type=0 for Ethernet Bridge Application, or insmod ixFpAccCodeletMain type=1 for Routed IP Application.

6. User can define the interval to run the codelet by changing the IX_FPATHACC_CODELET_RUN_IN_SEC as defined in the #define section of this file.
7. Wait until the system initialization has been completed.
8. Start the Adtech Generator Card.
9. After IX_FPATHACC_CODELET_RUN_IN_SEC seconds, Fastpath statistic of the codelet will be displayed on the screen.

Note: In order to run the Fastpath codelet, make sure that the IX_ETH_ACC_FPATH_AWARE flag in IxEthAccFpathDep.h is set to '1' before compiling the image.

Buffer Management

The codelet 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 codelet to invalidating and flushing cached memory when manipulating packets within buffers on the data path.

More specifically the data portion of buffers manipulated by the xscale 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 XScale 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 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 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.

Note: 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 codelet where IP packets are prepended with ethernet frame headers before being submitted to the Ethernet driver.

Connection Database —

The codelet maintains a simple connection database so that 8 fastpath 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 codelets. See IxFpathAccCodeletDefines.h for corresponding #defines 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 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 source code codelet 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 fowards the packet to an encapsulation specific handler in either the router or bridge codelet files.
- show the following IxFpathAcc API :-
    1. ixFpathAccInit()
    2. ixFpathAccBufferSupply()
    3. ixFpathAccBufferPoolInfoGet()
    4. 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.
- **IxFpathAccCodeletRouted.c:** Routed VC codelet.
- Flow learning and aging example.
- shows IxFpathAcc API :-
    ixFpathAccTemplatePairSet()
- **IxFpathAccCodeletBridge.c:** Bridged VC codelet
- Wan Mac learning example.
- shows 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 source.

Fastpath Ethernet Bridge Codelet Flow

The fastpath codelet for Ethernet Bridge initializes the s/w components, A 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 codelet 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.

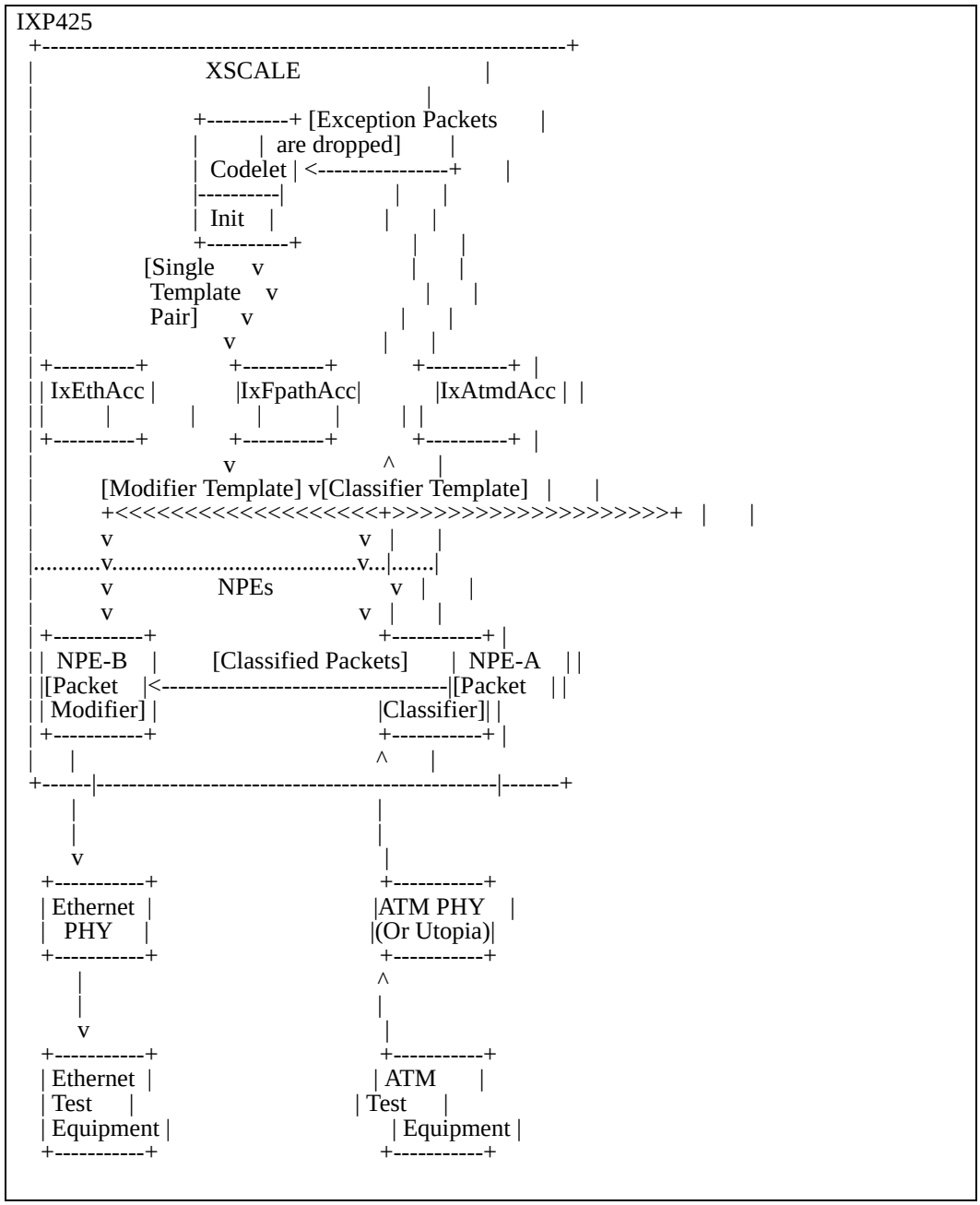
Packets which fail classification are dropped in the codelet and the buffer(s) associated with the packet is recycled to the buffer pool.

Once this codelet started, 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.

Figure 88. Bridge Data Flow



Fastpath Routed IP Codelet Flow

This codelet initialises 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 shown.

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.

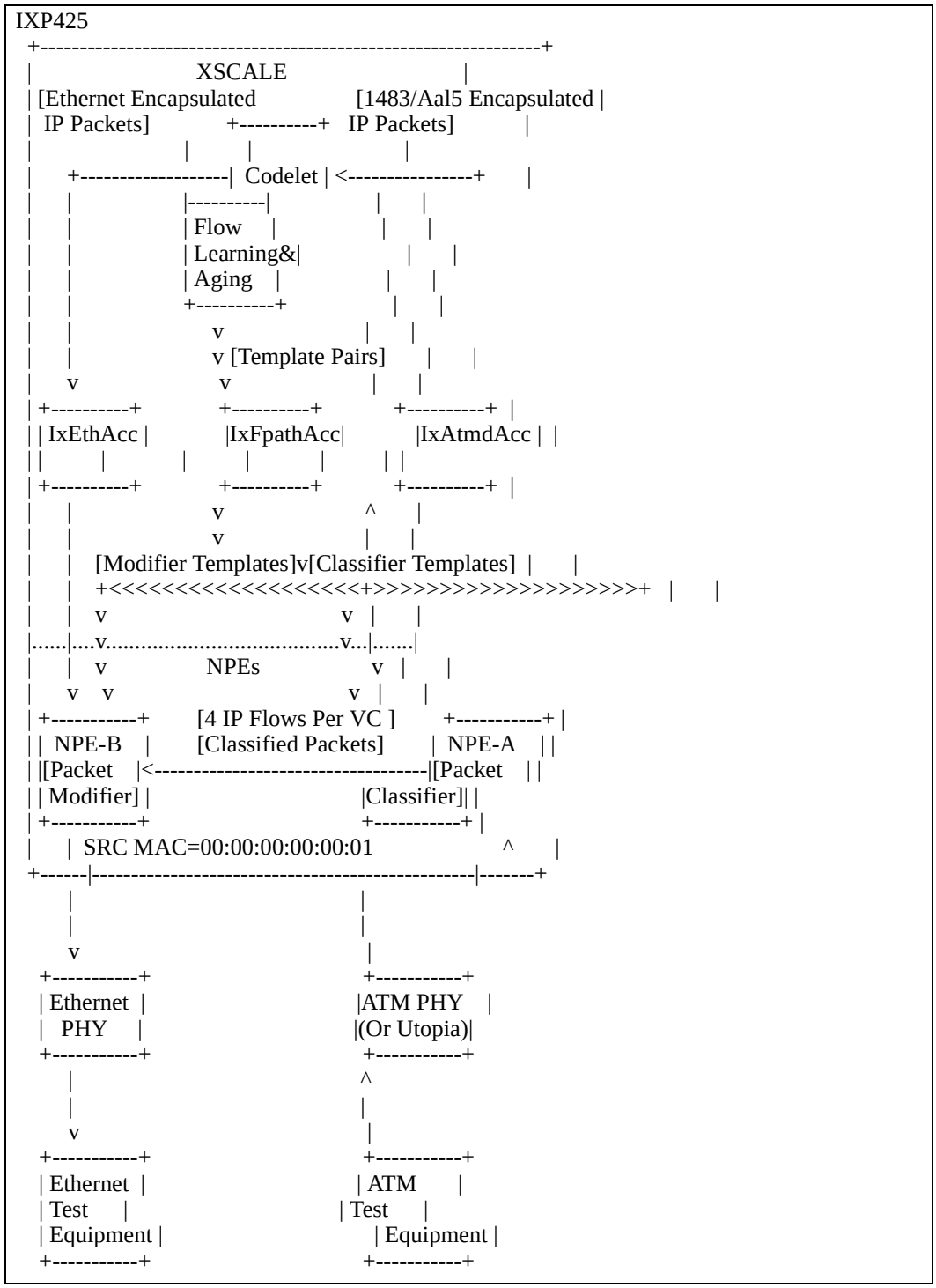
Packets which fail classification continue to the learning function and are subsequently forwarded to the Ethernet driver (IxEthAcc) to a single fixed MAC address.

Once this codelet started, 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 re-encapsulated in Ethernet frames then forwarded to IxEthAcc.
- Destination Mac for all IP packets = 00:00:00:00:00:03

Figure 89. Routed Data Flow



Note: If validation board is used (instead of using IXDP425 development board), then uncomment VALIDATION_PLATFORM_USED flag in component.mk

B.7 IXP425 HSS Access Codelet (IxHssAccCodelet) API

IXP425 HSS Access Codelet (IxHssAccCodelet) APIIXP425 HSS Access Codelet (IxHssAccCodelet) APIIXP425 HSS Access Codelet (IxHssAccCodelet) APIIXP425 HSS Access Codelet (IxHssAccCodelet) APIIXP425 HSS Access Codelet API. The interface for the HSS Access Codelet.

B.7.1 Defines

- #define **IX_HSSACC_CODELET_DURATION_IN_MS** 10000

B.7.2 Functions

- PUBLIC IX_STATUS **ixHssAccCodeletMain** (IxHssAccCodeletOperation operationType, IxHssAccCodeletPortMode portMode, IxHssAccCodeletVerifyMode verifyMode)

B.7.3 Detailed Description

IXP425 HSS Access Codelet API. The interface for the HSS Access Codelet.

This module contains the implementation of the HSS Access Codelet.

The following top-level operation is supported:

- Test Packetised and Channelised Services, with the Codelet acting as data source/sink and HSS as loopback. The Codelet will transmit data and verify that data received is the same as that transmitted. Codelet runs for IX_HSS_CODELET_DURATION_IN_MS ms.

Assumptions

In Channelised service, the codelet transmits traffic continuously. When the codelet runs up to IX_HSS_CODELET_DURATION_IN_MS ms, Tx counter is bigger than Rx counter. This is due to the fact that traffics submitted to NPE (i.e. Tx counter has been increased) are not transmitted out by NPE when HSS service is disabled. These traffics will be dropped and not loopbacked at HSS (Hence, Rx counter not increased).

In Packetised-raw mode service (client 1 and 3), Rx counter will be bigger than Tx counter because in this service, idle packets are received by XScale and causes Rx counter to be bigger than Tx counter. As for packetised-HDLC service, idle packets are handled in HDLC coprocessor and not passed to XScale (Hence, Rx counter not increased).

VxWorks User Guide

ixHssAccCodeletMain() function is used as a single point of execution for HssAcc Codelet.

Usage : >ixHssAccCodeletMain (operationType, portMode, verifyMode) Where operationType: 1 = Packetised Service Only. 2 = Channelised Service Only. 3 = Packetised Service and Channelised Services.

Where portMode: 1 = HSS Port 0 Only. 2 = HSS Port 1 Only. 3 = HSS Port 0 and 1.

Where verifyMode: 1 = codelet verifies traffic received in hardware loopback mode. 2 = codelet does not verify traffic received in hardware loopback mode.

Linux User Guide

The idea of using the ixHssAccCodeletMain() as a single point of execution for HssAcc codelet. The operation selected will be executed when user issue 'insmod' in command prompt.

Usage : >insmod csr_codelets_hssAcc.o operationType=(a) portMode=(b) verifyMode=(c) Where a: 1 = Packetised Service Only. 2 = Channelised Service Only. 3 = Packetised Service and Channelised Services.

Where b: 1 = HSS Port 0 Only. 2 = HSS Port 1 Only. 3 = HSS Port 0 and 1.

Where c: 1 = codelet verifies traffic received in hardware loopback mode. 2 = codelet does not verify traffic received in hardware loopback mode.

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 tranmsit 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 transferral of the data between XScale and NPE is not very appropriate. Instead, circular buffers are used whereby the NPE reads data from a block of SDRAM that the XScale writes to, and writes data to a block of SDRAM that the XScale reads from. On receive, the NPE writes directly into a circular buffer that the XScale 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 XScale 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 XScale 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 XScale 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 XScale to co-ordinate its reading and writing activities to maintain the data flow. The Tx offset lets the XScale know the list offset into the Tx circular pointer list that the NPE will next use to transmit. The Rx offset lets the XScale 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 XScale 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.

B.8 IXP425 PerfProf Access Codelet

IXP425 PerfProf Access CodeletIXP425 PerfProf Access CodeletIXP425 PerfProf Access CodeletIXP425 PerfProf Access Codelet(IxPerfProfAccCodelet) API IXP425 codelet PerfProf Access component API

B.8.1 Defines

- #define PSS_MASK 0x3f000
  Masks out PSS portion of the PMSR register.
- #define **EXPANSION_BUS** 0x1000
- #define **SDRAM_CONTROLLER** 0x2000
- #define **PCI** 0x4000
- #define **QUEUE_MANAGER** 0x8000
- #define **AHB_APB_BRIDGE** 0x10000

B.8.2 Enumerations

- enum IxPerfProfAccCodeletMode { IX_PERFPROF_ACC_CODELET_MODE_HELP = 0,
  IX_PERFPROF_ACC_CODELET_MODE_ALL,
  IX_PERFPROF_ACC_CODELET_MODE_BUS_PMU_NORTH_MODE,
  IX_PERFPROF_ACC_CODELET_MODE_BUS_PMU_SOUTH_MODE,
  IX_PERFPROF_ACC_CODELET_MODE_BUS_PMU_SDRAM_MODE,
  IX_PERFPROF_ACC_CODELET_MODE_XSCALE_PMU_EVENT_SAMPLING,
  IX_PERFPROF_ACC_CODELET_MODE_XSCALE_PMU_TIME_SAMPLING,
  IX_PERFPROF_ACC_CODELET_MODE_XSCALE_PMU_EVENT_COUNTING,
  IX_PERFPROF_ACC_CODELET_MODE_XCYCLE,
  IX_PERFPROF_ACC_CODELET_MODE_BUS_PMU_PMSR_GET }
  Contains selection of mode to be used when calling the main API.

B.8.3 Functions

- PUBLIC void **ixPerfProfAccCodeletMain** (IxPerfProfAccCodeletMode mode, UINT32 param1, UINT32 param2, UINT32 param3, UINT32 param4, UINT32 param5, UINT32 param6, UINT32 param7, UINT32 param8, UINT32 param9)

B.8.4 Detailed Description

(IxPerfProfAccCodelet) API IXP425 codelet PerfProf Access component API

Functionality of the PerfProf Access Codelet

The codelet shall demonstrate how the Performance Profiling utility can be used for profiling purposes.

- The different implementations are demonstrated.
- Help - Lists down how the codelet can be used.
- Demo All - A non user configurable demonstration on how to use the APIs
- Bus Pmu North Mode - Profiling of the north bus activities. Enables user to select events that they wish to monitor.
- Bus Pmu South Mode - Profiling of the south bus activities. Enables user to select events that they wish to monitor.
- Bus Pmu Sdram Mode - Profiling of the sdram bus activities. Enables user to select events that they wish to monitor.
- Bus Pmu PMSR Get Mode - Get the last slave or master to access the bus.
- Xscale PMU Event Sampling - Event Sampling of Xscale PMU. Enables user to select event and sampling rate that they wish to sample.
- Xscale PMU Time Sampling - Time Sampling of Xscale PMU. Enables user to select clock count mode and number of events and rate they wish to sample.
- Xscale PMU Event Counting - Event counting of Xscale PMU. Enables users to select events that they wish to count or monitor.
- Xcycle Measurement - Measurement of cycle idle time. i.e when the cycles are not being used to process anything.

User Guide

Users will be able to start the codelet by calling ixPerfProfAccCodeletMain and passing in up to 10 parameters. The parameters are represented in the following order:

Help Mode *Mode - Select IX_PERFPROF_ACC_CODELET_MODE_HELP Set the rest of the parameters to 0.* ***All functionalities mode*** *Mode - Select IX_PERFPROF_ACC_CODELET_MODE_ALL Set the rest of the parameters to 0.* ***Bus PMU north mode*** *Mode - Select IX_PERFPROF_ACC_CODELET_MODE_BUS_PMU_NORTH_MODE param1 - Select proper PEC1 value from main header file. param2 - Select proper PEC2 value from main header file. param3 - Select proper PEC3 value from main header file. param4 - Select proper PEC4 value from main header file. param5 - Select proper PEC5 value from main header file. param6 - Select proper PEC6 value from main header file. param7 - Select proper PEC7 value from main header file. Set the rest of the parameters to 0.* ***Bus PMU south mode*** *Mode - Select IX_PERFPROF_ACC_CODELET_MODE_BUS_PMU_SOUTH_MODE param1 - Select proper PEC1 value from main header file. param2 - Select proper PEC2 value from main header file. param3 - Select proper PEC3 value from main header file. param4 - Select proper PEC4 value from main header file. param5 - Select proper PEC5 value from main header file. param6 - Select proper PEC6 value from main header file. param7 - Select proper PEC7 value from main header file. Set the rest of the parameters to 0.* ***Bus PMU sdram mode*** *Mode - Select IX_PERFPROF_ACC_CODELET_MODE_BUS_PMU_SDRAM_MODE param1 - Select*

proper PEC1 value from main header file. param2 - Select proper PEC2 value from main header file. param3 - Select proper PEC3 value from main header file. param4 - Select proper PEC4 value from main header file. param5 - Select proper PEC5 value from main header file. param6 - Select proper PEC6 value from main header file. param7 - Select proper PEC7 value from main header file. Set the rest of the parameters to 0. ***Bus PMU PMSR Get*** *Mode - Select IX_PERFPROF_ACC_CODELET_MODE_BUS_PMU_PMSR_GET Set the rest of the parameters to 0.* ***XScale PMU Event Sampling*** *Mode - Select IX_PERFPROF_ACC_CODELET_MODE_DEMO_XSCALE_PMU_EVENT_SAMPLING param1 - Number of events param2 - Event 1 param3 - Sampling rate of Event 1 param4 - Event 2 param5 - Sampling rate of Event 2 param6 - Event 3 param7 - Sampling rate of Event 3 param8 - Event 4 param9 - Sampling rate of Event 4* ***XScale PMU Time Sampling*** *Mode - Select IX_PERFPROF_ACC_CODELET_MODE_DEMO_XSCALE_PMU_TIME_SAMPLING param1 - Sampling rate. param2 - Clock count divider. Set the rest of the parameters to 0.* ***XScale PMU Event Counting*** *Mode - Select IX_PERFPROF_ACC_CODELET_MODE_DEMO_XSCALE_PMU_EVENT_COUNTING param1 - Clock count divider. param2 - Number of events. param3 - Event 1. param4 - Event 2. param5 - Event 3. param6 - Event 4. Set the rest of the parameters to 0.* ***Xcycle Measurement*** *Mode - Select IX_PERFPROF_ACC_CODELET_MODE_XCYCLE param1 - Number of runs required. Set the rest of the parameters to 0.*

VxWorks User Guide

ixPerfProfAccCodeletMain() function is used as a single point of execution for PerfProfAcc Codelet. It allows user to enter selection for different type of supported operations as described above.

Usage : >ixPerfProfAccCodeletMain (mode, param1, param2, param3, param4, param5, param6, param7, param8, param9) Where mode and params are described above.

Linux User Guide

ixPerfProfAccCodeletMain() function is used as a single point of execution for PerfProfAcc Codelet. It allows user to enter selection for different type of supported operations as described above.

*Usage : >insmod csr_codelets_perfProfAcc.o \ mode= \ param1= \ param2= \ param3= \ param4= \ param5= \ param6= \ param7= \ param8= \ param9= *

Where Parameter X are as described above.

B.8.5 Define Documentation

- #define PSS_MASK 0x3f000

  Masks out PSS portion of the PMSR register.

  Definition at line 220 of file IxPerfProfAccCodelet.h.

B.8.6 Enumeration Type Documentation

- enum IxPerfProfAccCodeletMode

  Contains selection of mode to be used when calling the main API.

Enumeration Values

- ***IX_PERFPROF_ACC_CODELET_MODE_HELP*** Select help mode.
- ***IX_PERFPROF_ACC_CODELET_MODE_ALL*** Select all mode.
- ***IX_PERFPROF_ACC_CODELET_MODE_BUS_PMU_NORTH_MODE*** Select north bus pmu mode.
- ***IX_PERFPROF_ACC_CODELET_MODE_BUS_PMU_SOUTH_MODE*** Select south bus pmu mode.
- ***IX_PERFPROF_ACC_CODELET_MODE_BUS_PMU_SDRAM_MODE*** Select sdram mode.
- ***IX_PERFPROF_ACC_CODELET_MODE_XSCALE_PMU_EVENT_SAMPLING*** Select xscale pmu event sampling mode.
- ***IX_PERFPROF_ACC_CODELET_MODE_XSCALE_PMU_TIME_SAMPLING*** Select xscale pmu time sampling mode.
- ***IX_PERFPROF_ACC_CODELET_MODE_XSCALE_PMU_EVENT_COUNTING*** Select xscale pmu event counting mode.
- ***IX_PERFPROF_ACC_CODELET_MODE_XCYCLE*** Select xcycle mode.
- ***IX_PERFPROF_ACC_CODELET_MODE_BUS_PMU_PMSR_GET*** Select bus pmu pmsr get mode.

  Definition at line 279 of file IxPerfProfAccCodelet.h.

B.9 IXP425 Timers (IxTimersCodelet) API

IXP425 Timers (IxTimersCodelet) APIIXP425 Timers (IxTimersCodelet) APIIXP425 Timers (IxTimersCodelet) APIIXP425 Timers (IxTimersCodelet) APIIXP425 Timer component API.

B.9.1 Typedefs

- typedef void(* IxTimerIsr)(void *arg)

  Timer callback prototype.

B.9.2 Enumerations

- enum IxTimerId { IX_TIMER_1, IX_TIMER_2, IX_TIMER_WDOG, IX_TIMER_TS, IX_TIMER_PMU, IX_TIMER_MAX }

  Hardware timers.

B.9.3 Functions

- PUBLIC IX_STATUS ixTimerInit (BOOL recoverFromLostStatus)

  Initialise the Timer API.
- PUBLIC IX_STATUS ixTimerBind (IxTimerId timer, IxTimerIsr isr, void *arg)

  Initialise a Timer.

- PUBLIC IX_STATUS ixTimerEnable (IxTimerId timer, UINT32 downCounter, BOOL oneShot)
  Enable a Timer.
- PUBLIC IX_STATUS ixTimerDisable (IxTimerId timer)
  Disable a Timer.
- PUBLIC IX_STATUS ixTimerDownCounterGet (IxTimerId timer, UINT32 *downCounter)
  Get the remaining time of a timer.
- PUBLIC IX_STATUS ixTimerUnbind (IxTimerId timer)
  Unregister a Timer.
- PUBLIC void ixTimerShow (void)
  Show internal counters.
- PUBLIC IX_STATUS ixTimerMemMap (void)
  Maps the timer address space.
- PUBLIC void ixTimerMemUnmap (void)
  Unmaps the timer address space.

B.9.4 Detailed Description

IXP425 Timer component API.

These utilities provide support for enabling, triggering and disabling timers

B.9.5 Typedef Documentation

- IxTimerIsr
  Timer callback prototype.
  Definition at line 98 of file IxTimersCodelet.h.

B.9.6 Enumeration Type Documentation

- enum IxTimerId
  Hardware timers.

Enumeration Values

- ***IX_TIMER_1*** General Purpose Timer 1.
- ***IX_TIMER_2*** General Purpose Timer 2.
- ***IX_TIMER_WDOG*** Watchdog Timer.
- ***IX_TIMER_TS*** Timestamp Timer.
- ***IX_TIMER_PMU*** XScale core PMU timer.
- ***IX_TIMER_MAX*** Delimiter for error checking.
  Definition at line 81 of file IxTimersCodelet.h.

B.9.7 Function Documentation

- ixTimerBind (IxTimerId *timer*, IxTimerIsr *isr*, void * *arg*)
  Initialise a Timer.
  This function is called to initialise one of the timers

Parameters

- *timer* (in) the timer to initialise
- *isr* (in) the callback to register
- *arg* (in) the parameter to be passed to the callback

Returns

- IX_SUCCESS, the timer successfully initialised
- IX_FAIL, failed to initialize the timer
- ixTimerDisable (IxTimerId *timer*)
  Disable a Timer.
  This function is called to disable a timer

Parameters

- *timer* (in) the timer to disable

Returns

- IX_SUCCESS, the timer successfully disabled
- IX_FAIL, failed to disable the timer
  Note: This function can be used from an interrupt context
- ixTimerDownCounterGet (IxTimerId *timer*, UINT32 * *downCounter*)
  Get the remaining time of a timer.
  This function is called to get the remaining time before the next interrupt triggers
  If this function is used after a timer expires, the result is a negative value. Its unsigned representation is a number close to 0xfffff.....
  Note: Watchdog timer stops when expiring. Then the counter retrived by this function is always 0.

Parameters

- *timer* (in) the timer to query
- *downCounter* (out) the timer time left

Returns

- IX_SUCCESS, the timer successfully queried
- IX_FAIL, failed to query the timer

Note: This function can be used from an interrupt context

- ixTimerEnable (IxTimerId *timer*, UINT32 *downCounter*, BOOL *oneShot*)
  Enable a Timer.

This function is called to enable a timer

Parameters

- *timer* (in) the timer to enable
- *downCounter* (in) the number of cycles between interrupts. downCounter should be a multiple of 4 for *IX_TIMER_1* and *IX_TIMER_2* timers. This counter is in pClock cycles (peripheral cycles) except for the *IX_TIMER_PMU* where the unit is in xClock cycles (processor frequency)
- *oneShot* (in) the type of timer to trigger. When set to TRUE, the interrupt will fire once when the downCounter reaches 0. When set to FALSE, the interrupts will fire every `downCounter` bus cycles. FALSE is not supported for *IX_TIMER_WDOG* and *IX_TIMER_TS* timers.

Returns

- IX_SUCCESS, the timer successfully enabled
- IX_FAIL, failed to enable the timer

  Note: This function can be used from an interrupt context
- ixTimerInit (BOOL *recoverFromLostStatus*)

  Initialise the Timer API.

  This function must be called before and other IxTimer function. It sets up internal data structures.

Parameters

- *recoverFromLostStatus* (in) set to true to enable Lost Status workaround

Returns

- IX_SUCCESS, the IxTimer API successfully initialised
- IX_FAIL, failed to initialize the IxTimer API
- ixTimerMemMap (void)

  Maps the timer address space.

  This function is called in order to map the physical address space for the timer to virtual adress space

Parameters

- *none*

Returns

- IX_SUCCESS, the timer physical address space was mapped to virtual address space
- IX_FAIL, the timer physical address space failed to be mapped to virtual address space
- ixTimerMemUnmap (void)

  Unmaps the timer address space.

  This function is called in order to unmap the physical address space for the timer

Parameters

- *none*

Returns

- none
- ixTimerShow (void)
  Show internal counters.
  This function is called to show the internal counters icremented when Lost Status workaround is needed.

Parameters

- *none*

Returns

- void
- ixTimerUnbind (IxTimerId *timer*)
  Unregister a Timer.
  This function is called to unregister a timer

Parameters

- *timer* (in) the timer to unregister

Returns

- IX_SUCCESS, the timer successfully removed
- IX_FAIL, failed to remove the timer

B.10 IXP425 USB RNDIS Codelet (IxUSBRNDIS) API

IXP425 USB RNDIS Codelet (IxUSBRNDIS) APIIXP425 USB RNDIS Codelet (IxUSBRNDIS) APIIXP425 USB RNDIS Codelet (IxUSBRNDIS) APIIXP425 USB RNDIS Codelet (IxUSBRNDIS) APIIXP425 codelet USB RNDIS API.

B.10.1 Modules

- IXP425 USB RNDIS End Driver Codelet (IxUSBRNDIS) API
  IXP425 codelet for RNDIS End API.
- IXP425 USB RNDIS Vendor Codelet (IxUSBRNDIS)
  IXP425 codelet for RNDIS Vendor information.

B.10.2 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)
  Function prototype for layer initialization.
- PUBLIC IX_STATUS ixUSBRNDISInit (void)
  Function prototype for initializing RNDIS.
- PUBLIC void ixUSBRNDISUnload (void)
  Function prototype for releasing the I/O memory and disconnecting the interrupt.
- PUBLIC IX_USB_MBLK * ixUSBRNDISCreateMBuf (UINT8 *buffer, UINT32 len)
  Function prototype for creating MBufs.
- PUBLIC IX_STATUS ixUSBRNDISSendDataPacket (RNDIS_BUF *packet)
  Function prototype for sending data packet. It is the hook for the RNDIS END.
- void **ixUSBRNDISIpHdrDump** (const char *const mData)
- const char * **ixUSBRNDISIpProtoStrGet** (const UINT8 ipProto)
- const char * **ixUSBRNDISEthernetTypeStrGet** (const UINT16 etherType)
- void **ixUSBRNDISEthernetHdrDump** (const char *const mData, BOOL *nonIpHdrDetected)

B.10.3 Detailed Description

IXP425 codelet USB RNDIS API.

How to use the USB RNDIS codelet:

- build a loadable object and load it into vxWorks
- start the codelet by typing ixUSBRNDISStart

You should see the "usb" network interface in the output generated by the ifShow command.

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.

Notes:

- 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 IXP425 into a network.

Warning: this codelet is for demonstration purposes only, it should not be considered a fully working application.

B.10.4 Function Documentation

- PUBLIC IX_USB_MBLK * ixUSBRNDISCreateMBuf (UINT8 * *buffer*, UINT32 *len*)
  Function prototype for creating MBufs.

Parameters

- *UINT8* *buffer - Pointer to a buffer
- *UINT32* len - Length of buffer

Returns

- IX_USB_MBLK - Successfully create MBUF
- NULL - Failed to create MBUF
- PUBLIC IX_STATUS ixUSBRNDISInit (void)
  Function prototype for initializing RNDIS.

Parameters

- *None*

Returns

- IX_SUCCESS - Successfully initalized RNDIS
- IX_FAIL - Failed to initialize RNDIS
- PUBLIC IX_STATUS ixUSBRNDISLayerInit (void * *pDrvCtrl*)
  Function prototype for layer initialization.

Parameters

- *pDrvCtrl* - Pointer to the device to be initialized

Returns

- IX_SUCCESS - Successfully initalized
- IX_FAIL - Failed to initialize
- PUBLIC IX_STATUS ixUSBRNDISProcessDataPacket (IX_USB_MBLK *)
  Function prototype for processing data packet.

Parameters

- *IX_USB_MBLK* - data packet to be processed

Returns

- IX_SUCCESS - Data packet successfully processed
- IX_FAIL - Failed to process data packet
- PUBLIC IX_STATUS ixUSBRNDISProcessEncapsulatedCommand (IX_USB_MBLK *)
  Function prototype for processing encapsulation command.

Parameters

- *IX_USB_MBLK* - memory buffer to be encapsulated

Returns

- IX_SUCCESS - Succesfully processing encapsulation command
- IX_FAIL - Failed to process encapsulation command for some internal reason
- PUBLIC IX_STATUS ixUSBRNDISSendDataPacket (RNDIS_BUF * *packet*)
  Function prototype for sending data packet. It is the hook for the RNDIS END.

Parameters

- *RNDIS* *packet - Pointer to a data packet to be transmitted

Returns

- IX_SUCCESS - Successfully send a data packet
- IX_FAIL - Failed to send a data packet
- PUBLIC IX_STATUS ixUSBRNDISSignalEncapsulatedCommand (void)
  Function prototype for signalling encapsulation command.

Parameters

- *None*

Returns

- IX_SUCCESS - Succesfully signalling encapsulation command
- IX_FAIL - Failed to signal encapsulation command for some internal reason
- PUBLIC void ixUSBRNDISUnload (void)
  Function prototype for releasing the I/O memory and disconnecting the interrupt.

Parameters

- *None*

Returns

- None

B.11 IXP425 USB RNDIS End Driver Codelet (IxUSBRNDIS) API

IXP425 USB RNDIS End Driver Codelet (IxUSBRNDIS) APIIXP425 USB RNDIS End Driver Codelet (IxUSBRNDIS) APIIXP425 USB RNDIS End Driver Codelet (IxUSBRNDIS) APIIXP425 USB RNDIS End Driver Codelet (IxUSBRNDIS) APIIXP425 codelet for RNDIS End API.

B.11.1 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.

B.11.2 Functions

- PUBLIC STATUS ixUSBRNDISStart (void)
  Starts the RNDIS component.
- PUBLIC void rndisInt (void *pDrvCtrl, int packet)
  RNDIS END hook for receiving packets.

B.11.3 Detailed Description

IXP425 codelet for RNDIS End API.

B.11.4 Function Documentation

- PUBLIC STATUS ixUSBRNDISStart (void)
  Starts the RNDIS component.

Parameters

None

Returns

- OK - Succesfully start RNDIS component
- ERROR - Failed to start RNDIS component

- PUBLIC void rndisInt (void * *pDrvCtrl*, int *packet*)
  RNDIS END hook for receiving packets.

Parameters

- *pDrvCtrl* - pointer to END device
- *packet* - received packet

Returns

None

B.12 IXP425 USB RNDIS Vendor Codelet (IxUSBRNDIS)

IXP425 USB RNDIS Vendor Codelet (IxUSBRNDIS)IXP425 USB RNDIS Vendor Codelet (IxUSBRNDIS)IXP425 USB RNDIS Vendor Codelet (IxUSBRNDIS)IXP425 USB RNDIS Vendor Codelet (IxUSBRNDIS)IXP425 codelet for RNDIS Vendor information.

B.12.1 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.

B.12.2 Detailed Description

IXP425 codelet for RNDIS Vendor information.

B.13 IXP425 Crypto Access Codelet

IXP425 Crypto Access CodeletIXP425 Crypto Access CodeletIXP425 Crypto Access CodeletIXP425 Crypto Access Codelet(IxCryptoAccCodelet) API IXP425 Crypto Access Codelet API

B.13.1 Defines

- #define **INLINE** __inline__
- #define IX_CRYPTOACC_CODELET_NPEC_IMAGE_ID
  IX_NPEDL_NPEIMAGE_NPEC_CRYPTO_AES_ETH
  The build ID for NPE C image, default is IX_NPEDL_NPEIMAGE_NPEC_CRYPTO_AES_ETH.
- #define IX_CRYPTOACC_CODELET_CIPHER_ALGO
  (IX_CRYPTO_ACC_CIPHER_DES)
  Cipher algorithm.
- #define IX_CRYPTOACC_CODELET_CIPHER_MODE
  (IX_CRYPTO_ACC_MODE_CBC)
  Mode of operation for the cipher algorithm.
- #define IX_CRYPTOACC_CODELET_CIPHER_KEY_LEN
  (IX_CRYPTO_ACC_DES_KEY_64)
  Permitted key length according to the cipher algorithm.

- #define IX_CRYPTOACC_CODELET_CIPHER_BLOCK_LEN (IX_CRYPTO_ACC_DES_BLOCK_64)
  Cipher block length for the cipher algorithm.
- #define IX_CRYPTOACC_CODELET_CIPHER_IV_LEN (IX_CRYPTO_ACC_DES_IV_64)
  Initialisation vector length for the selected mode of operation and cipher algorithm.
- #define IX_CRYPTOACC_CODELET_AUTH_ALGO (IX_CRYPTO_ACC_AUTH_SHA1)
  Authentication algorithm.
- #define IX_CRYPTOACC_CODELET_AUTH_KEY_LEN (IX_CRYPTO_ACC_SHA1_KEY_160)
  Key length permitted according to the authentication algorithm selected above.
- #define IX_CRYPTOACC_CODELET_AUTH_DIGEST_LEN (IX_CRYPTO_ACC_SHA1_DIGEST_160)
  Message digest length for the authentication algorithm selecetd above.
- #define PERFORMANCE_WINDOW_SIZE 20
  Number of crypto packets need to be sent in for performance benchmarking (sampling size), 1 unit represents 1000 of packets.
- #define IX_CRYPTOACC_CODELET_CRYPTO_MBUF_POOL_SIZE 20
  Size of MBuf pool, ie number of buffers to circulate.
- #define IX_CRYPTOACC_CODELET_BATCH_LEN 1000
  Number of packets per sampling for performance benchmarking.
- #define IX_CRYPTOACC_CODELET_QMR_DISPATCHER_PRIORITY 150
  Recommended priority of queue manager dispatch loop.
- #define IX_CRYPTOACC_CODELET_XSCALE_TICK 66
  The XScale tick is 66 MHz.
- #define IX_CRYPTOACC_CODELET_REGISTER_WAIT_TIME 500
  The codelet register wait time (in ms) in a for loop.
- #define IX_CRYPTOACC_CODELET_PERFORM_WAIT_TIME 500
  The codelet perform wait time (in ms) in a for loop.
- #define IX_CRYPTOACC_CODELET_MAX_TIME_WAIT 200
  Maximum time (in ms) to wait before exiting the program.
- #define IX_CRYPTOACC_CODELET_MIN_CRYPTO_PKT_SIZE 64
  The minimum packet size.
- #define IX_CRYPTOACC_CODELET_MAX_CRYPTO_PKT_SIZE 65456
  The maximum packet size.
- #define IX_CRYPTOACC_CODELET_NPEB_BUILD_ID 1
  NPE B Build ID.
- #define IX_CRYPTOACC_CODELET_QMGR_DISPATCHER_MODE (FALSE)
  QMgr Dispatcher Mode, whether in interrupt (TRUE) or poll (FALSE) mode.

B.13.2 Enumerations

- enum IxCryptAccCodeletOperation { IX_CRYPTOACC_CODELET_ENCRYPT = 1, IX_CRYPTOACC_CODELET_AUTH, IX_CRYPTOACC_CODELET_ENCRYPT_AUTH }

  The codelet mode of operation.

B.13.3 Functions

- IX_STATUS ixCryptoAccCodeletMain (IxCryptAccCodeletOperation operationType, UINT32 packetLen)

  This is the entry point function to the codelet to choose the operation for the codelet and packet length to be used. This is the main function of the codelet where crypto contexts registration and crypto perform services are done. Based on the selected operation and selected packet length, packets are sent to cryptoAcc for processing.

B.13.4 Detailed Description

(IxCryptoAccCodelet) API IXP425 Crypto Access Codelet API

The codelet demonstrates how the Security Hardware Accelerator API can be used for cryptographic purposes.

Three different cryptographic operations are demonstrated :

- Encryption and decryption. Default is DES-CBC.
- Authentication calculation and check. Default is SHA1.
- Combined service of encryption/authentication calculation and authentication check/ decryption. Default is DES-CBC mode with SHA1.

Here is the list of functionalities of the codelet and they are in the order in which the codelet must follow in order to utilise the Security Hardware Accelerator, i.e. Initialise the related components, followed by crypto context registration, send in crypto services request using the registered crypto context, unregister crypto context, etc.

- The codelet demonstrates how the Network Processor Engine (NPE) is initialised, how the NPE image is downloaded to the NPE, and and how the NPE is started.
- The codelet demonstrates how the Queue Manager is initialised to polling or interrupt mode.
- The codelet shows how to set up the callback functions and how to register crypto contexts with cryptoAcc access component.
- The codelet demonstrates how the registration should be done for the three different operations using the register API.
- The codelet demonstrates how the perform API can be used after successful registration.
- The codelet demonstrates the use of the unregister API in the event of re-starting the above mentioned operation.
- The codelet measures the performance of the Security Hardware Accelerator for each of the three operations stated above.

For both Linux and VxWorks, the type of cipher algorithm and authentication algorithm can be set at USER SETTING portion. The key length, block length and IV length must be set according to the algorithm standard. Please refer to IxCryptoAcc.h for permitted setting for each algorithm. Any changes to the setting below will require the codes to be recompiled.

Default Cryptographic Setting for Performance Benchmarking

- Cipher Algorithm : DES
- Cipher Mode of Operation : CBC
- Authentication Algorithm : SHA1

Default Performance Sampling Size

- Packets per Sampling : 1,000
- Number of Samples : 20
- Total Packets : 20,000

Default mbuf Pool Setting

- Number of mbufs in pool : 20
- Min of mbufs needed in pool : 2 (for crypto context registration purpose)

User may choose to run different operation with different packet length. The performance rate of selected operation with selected packet length will be displayed after throughput rate has been captured for a predefined performance sampling size.

Notes for Performance Benchmarking:

- Packets are generated in XScale with random payload. No explicit verification of payload will be carried out in the codelet.
- Encrypted packets are sent to NPE again for decryption and same applies to authentication operation. Packet payload is verified implicitly through this feedback system.
- Throughput rate is captured from the point where packet is sent to NPE for processing to the point where NPE completes the encryption and/or authentication operation and notifies XScale through callback.
- No external packet generator or benchmarking devices involved in this codelet. Throughput rate is captured using XScale timestamp.
- Time taken for codelet to complete the operation depends on packet length selected. Larger packet will take more time to complete.

VxWorks User Guide

ixCryptoAccCodeletMain() function is used as the entry point of execution for cryptoAcc Codelet.

It allows user to enter selection for different operations described below with different packet length.

Usage : >ixCryptoAccCodeletMain (operationType, packetLen) Where operationType: 1 = To encrypt and decrypt packets using selected cipher algorithm. 2 = To authenticate packets using selected authentication algorithm. 3 = To encrypt/authenticate and authenticate/decrypt packets using selected cipher algorithm and selected authentication

algorithm. Where packetLen: Packet length ranges from 64 bytes to 65456 bytes, if cipher algorithm is DES/3DES, packet length must be multiple of 8-byte (cipher block length); while AES algorithm must have packet length that is multiple of 16-byte in size.

Linux User Guide

The idea of using the ixCryptoAccCodeletMain() as the entry point of execution for cryptoAcc codelet.

The selected operation will be executed when user issues 'insmod' at command prompt.

Usage : >insmod csr_codelets_cryptoAcc.o operationType= packetLen= Where x: 1 = To encrypt and decrypt packets using selected cipher algorithm. 2 = To authenticate packets using selected authentication algorithm. 3 = To encrypt/authenticate and authenticate/decrypt packets using selected cipher algorithm and selected authentication algorithm. Where y: Packet length ranges from 64 bytes to 65456 bytes, if cipher algorithm is DES/3DES, packet length must be multiple of 8-byte (cipher block length); while AES algorithm must have packet length that is multiple of 16-byte in size.

B.13.5 Define Documentation

- #define IX_CRYPTOACC_CODELET_AUTH_ALGO (IX_CRYPTO_ACC_AUTH_SHA1)

  Authentication algorithm.

  Definition at line 297 of file IxCryptoAccCodelet.h.

- #define IX_CRYPTOACC_CODELET_AUTH_DIGEST_LEN (IX_CRYPTO_ACC_SHA1_DIGEST_160)

  Message digest length for the authentication algorithm selecetd above.

  Definition at line 318 of file IxCryptoAccCodelet.h.

- #define IX_CRYPTOACC_CODELET_AUTH_KEY_LEN (IX_CRYPTO_ACC_SHA1_KEY_160)

  Key length permitted according to the authentication algorithm selected above.

  Definition at line 308 of file IxCryptoAccCodelet.h.

- #define IX_CRYPTOACC_CODELET_BATCH_LEN 1000

  Number of packets per sampling for performance benchmarking.

  Definition at line 374 of file IxCryptoAccCodelet.h.

- #define IX_CRYPTOACC_CODELET_CIPHER_ALGO (IX_CRYPTO_ACC_CIPHER_DES)

  Cipher algorithm.

  Definition at line 246 of file IxCryptoAccCodelet.h.

- #define IX_CRYPTOACC_CODELET_CIPHER_BLOCK_LEN (IX_CRYPTO_ACC_DES_BLOCK_64)

  Cipher block length for the cipher algorithm.

  Definition at line 276 of file IxCryptoAccCodelet.h.

- #define IX_CRYPTOACC_CODELET_CIPHER_IV_LEN (IX_CRYPTO_ACC_DES_IV_64)

  Initialisation vector length for the selected mode of operation and cipher algorithm.

  Definition at line 287 of file IxCryptoAccCodelet.h.

- #define IX_CRYPTOACC_CODELET_CIPHER_KEY_LEN (IX_CRYPTO_ACC_DES_KEY_64)

  Permitted key length according to the cipher algorithm.

  Definition at line 266 of file IxCryptoAccCodelet.h.

- #define IX_CRYPTOACC_CODELET_CIPHER_MODE (IX_CRYPTO_ACC_MODE_CBC)

  Mode of operation for the cipher algorithm.

  Definition at line 256 of file IxCryptoAccCodelet.h.

- #define IX_CRYPTOACC_CODELET_CRYPTO_MBUF_POOL_SIZE 20

  Size of MBuf pool, ie number of buffers to circulate.

  Definition at line 339 of file IxCryptoAccCodelet.h.

- #define IX_CRYPTOACC_CODELET_MAX_CRYPTO_PKT_SIZE 65456

  The maximum packet size.

  Definition at line 444 of file IxCryptoAccCodelet.h.

- #define IX_CRYPTOACC_CODELET_MAX_TIME_WAIT 200

  Maximum time (in ms) to wait before exiting the program.

  Definition at line 424 of file IxCryptoAccCodelet.h.

- #define IX_CRYPTOACC_CODELET_MIN_CRYPTO_PKT_SIZE 64

  The minimum packet size.

  Definition at line 434 of file IxCryptoAccCodelet.h.

- #define IX_CRYPTOACC_CODELET_NPEB_BUILD_ID 1

  NPE B Build ID.

  Definition at line 454 of file IxCryptoAccCodelet.h.

- #define IX_CRYPTOACC_CODELET_NPEC_IMAGE_ID IX_NPEDL_NPEIMAGE_NPEC_CRYPTO_AES_ETH

  The build ID for NPE C image, default is IX_NPEDL_NPEIMAGE_NPEC_CRYPTO_AES_ETH.

  Note: Please refer to IxNpeDl.h for NPE image definition

  Definition at line 230 of file IxCryptoAccCodelet.h.

- #define IX_CRYPTOACC_CODELET_PERFORM_WAIT_TIME 500

  The codelet perform wait time (in ms) in a for loop.

  Definition at line 414 of file IxCryptoAccCodelet.h.

- #define IX_CRYPTOACC_CODELET_QMGR_DISPATCHER_MODE (FALSE)

  QMgr Dispatcher Mode, whether in interrupt (TRUE) or poll (FALSE) mode.

  Notes:

- QMgr dispatcher can be run in both poll mode and interrupt mode in vxWorks platform. When QMgr dispatcher is running in interrupt mode, the display of packet rate for Ethernet software loopback on the windshell is very slow if the crypto packet length chosen is very small. This is due to the time taken by NPE to service crypto packet is very fast and QMgr callbacks with higher priority are triggered frequently and blocked the task of displaying.
- In linux platform, QMgr Dispatcher is NOT advisible to run in poll mode. This is due to the task scheduling in poll mode is not fair enough between multiple threads that cause the performance rate to drop tremendously. Thus the performance rate displayed in Linux platform

using poll mode is not accurate. Performance rate is more accurate when QMgr dispatcher is running in interrupt mode.

Definition at line 484 of file IxCryptoAccCodelet.h.

- #define IX_CRYPTOACC_CODELET_QMR_DISPATCHER_PRIORITY 150

  Recommended priority of queue manager dispatch loop.

  Definition at line 384 of file IxCryptoAccCodelet.h.

- #define IX_CRYPTOACC_CODELET_REGISTER_WAIT_TIME 500

  The codelet register wait time (in ms) in a for loop.

  Definition at line 404 of file IxCryptoAccCodelet.h.

- #define IX_CRYPTOACC_CODELET_XSCALE_TICK 66

  The XScale tick is 66 MHz.

  Definition at line 394 of file IxCryptoAccCodelet.h.

- #define PERFORMANCE_WINDOW_SIZE 20

  Number of crypto packets need to be sent in for performance benchmarking (sampling size), 1 unit represents 1000 of packets.

  Definition at line 329 of file IxCryptoAccCodelet.h.

B.13.6 Enumeration Type Documentation

enum IxCryptAccCodeletOperation

The codelet mode of operation.

Enumeration Values

- ***IX_CRYPTOACC_CODELET_ENCRYPT*** Encryption and decryption.
- ***IX_CRYPTOACC_CODELET_AUTH*** Authentication calculation and check.
- ***IX_CRYPTOACC_CODELET_ENCRYPT_AUTH*** Combined service of encryption/ authentication calculation and authentication check/decryption.

  Definition at line 354 of file IxCryptoAccCodelet.h.

B.13.7 Function Documentation

- ixCryptoAccCodeletMain (IxCryptAccCodeletOperation *operationType*, UINT32 *packetLen*)

  This is the entry point function to the codelet to choose the operation for the codelet and packet length to be used. This is the main function of the codelet where crypto contexts registration and crypto perform services are done. Based on the selected operation and selected packet length, packets are sent to cryptoAcc for processing.

Parameters

- *IxCryptAccCodeletOperation [in] operationType* - operation for codelet. 1: Encrypt/decrypt operation 2: Authentication operation 3: Encrypt/decrypt and authentication operation
- *UINT32 [in] packetLen* - packet length

Returns

- IX_STATUS

- IX_SUCCESS - codelet runs successfully
- IX_FAIL - codelet fails

This page intentionally left blank.

API and Codelet Data Structures C

This appendix documents the data structures of the APIs and codelets for the Intel® IXP400 Software v.1.3's access-layer APIs.

This reference chapter was generated automatically using the most recent source code available when this document was generated, and is useful for reference purposes. Be advised that the software may have been subsequently updated and the user should use the source code for the most accurate reference information.

C.1 Data Structure Index

Data Structure Index 677
Aal5Statistics Struct Reference 682
Data Fields 682
Detailed Description 682
ActiveIpFlow Struct Reference 682
Data Fields 682
Detailed Description 683
ChannelisedStats Struct Reference 683
Data Fields 683
Detailed Description 683
GeneralStats Struct Reference 683
Data Fields 684
Detailed Description 684
ix_ossl_thread_main_info_t Struct Reference 684
Data Fields 684
Detailed Description 684
ix_ossl_time_t Struct Reference 685
Data Fields 685
Detailed Description 685
IxAtmCodeletStats Struct Reference 685
Data Fields 685
Detailed Description 686
IxAtmdAccUtopiaConfig Struct Reference 686
Data Fields 686
Detailed Description 687
IxAtmdAccUtopiaConfig::UtRxConfig_ Struct Reference 688
Data Fields 688
Detailed Description 689
Field Documentation 689
IxAtmdAccUtopiaConfig::UtRxDefineIdle_ Struct Reference 691
Data Fields 691
Detailed Description 691
IxAtmdAccUtopiaConfig::UtRxEnableFields_ Struct Reference 692
Data Fields 692

Detailed Description 693
Field Documentation 693
IxAtmdAccUtopiaConfig::UtRxStatsConfig_ Struct Reference 697
Data Fields 697
Detailed Description 697
Field Documentation 697
IxAtmdAccUtopiaConfig::UtRxTransTable0_ Struct Reference 698
Data Fields 698
Detailed Description 698
IxAtmdAccUtopiaConfig::UtRxTransTable1_ Struct Reference 699
Data Fields 699
Detailed Description 699
IxAtmdAccUtopiaConfig::UtRxTransTable2_ Struct Reference 699
Data Fields 700
Detailed Description 700
IxAtmdAccUtopiaConfig::UtRxTransTable3_ Struct Reference 700
Data Fields 700
Detailed Description 701
IxAtmdAccUtopiaConfig::UtRxTransTable4_ Struct Reference 701
Data Fields 701
Detailed Description 702
IxAtmdAccUtopiaConfig::UtRxTransTable5_ Struct Reference 702
Data Fields 702
Detailed Description 702
IxAtmdAccUtopiaConfig::UtSysConfig_ Struct Reference 702
Data Fields 703
Detailed Description 703
Field Documentation 703
IxAtmdAccUtopiaConfig::UtTxConfig_ Struct Reference 704
Data Fields 704
Detailed Description 705
Field Documentation 705
IxAtmdAccUtopiaConfig::UtTxDefineIdle_ Struct Reference 707
Data Fields 707
Detailed Description 707
IxAtmdAccUtopiaConfig::UtTxEnableFields_ Struct Reference 708
Data Fields 708
Detailed Description 709
Field Documentation 709
IxAtmdAccUtopiaConfig::UtTxStatsConfig_ Struct Reference 711
Data Fields 711
Detailed Description 711
Field Documentation 712
IxAtmdAccUtopiaConfig::UtTxTransTable0_ Struct Reference 712
Data Fields 712
Detailed Description 713
IxAtmdAccUtopiaConfig::UtTxTransTable1_ Struct Reference 713
Data Fields 713
Detailed Description 714
IxAtmdAccUtopiaConfig::UtTxTransTable2_ Struct Reference 714

Data Fields 714
Detailed Description 714
IxAtmdAccUtopiaConfig::UtTxTransTable3_ Struct Reference 715
Data Fields 715
Detailed Description 715
IxAtmdAccUtopiaConfig::UtTxTransTable4_ Struct Reference 715
Data Fields 716
Detailed Description 716
IxAtmdAccUtopiaConfig::UtTxTransTable5_ Struct Reference 716
Data Fields 716
Detailed Description 717
IxAtmdAccUtopiaStatus Struct Reference 717
Data Fields 717
Detailed Description 717
IxAtmdAccUtopiaStatus::UtRxCellConditionStatus_ Struct Reference 718
Data Fields 718
Detailed Description 719
IxAtmdAccUtopiaStatus::UtTxCellConditionStatus_ Struct Reference 719
Data Fields 719
Detailed Description 719
IxAtmmPortCfg Struct Reference 720
Data Fields 720
Detailed Description 720
IxAtmmVc Struct Reference 720
Data Fields 720
Detailed Description 721
Field Documentation 721
IxAtmScheduleTable Struct Reference 721
Data Fields 721
Detailed Description 721
Field Documentation 722
IxAtmScheduleTableEntry Struct Reference 722
Data Fields 722
Detailed Description 722
Field Documentation 723
IxAtmTrafficDescriptor Struct Reference 723
Data Fields 723
Detailed Description 724
IxCryptoAccAuthCtx Struct Reference 724
Data Fields 724
Detailed Description 725
Field Documentation 725
IxCryptoAccCipherCtx Struct Reference 725
Data Fields 725
Detailed Description 726
Field Documentation 726
IxCryptoAccCtx Struct Reference 727
Data Fields 727
Detailed Description 727
Field Documentation 728

IxEthAccMacAddr Struct Reference 728
Data Fields 728
Detailed Description 728
IxEthDBMacAddr Struct Reference 728
Data Fields 728
Detailed Description 729
IxEthDBPortDefinition Struct Reference 729
Data Fields 729
Detailed Description 729
IxEthEthObjStats Struct Reference 729
Data Fields 729
Detailed Description 730
IxFpCodeletFastPathConnection Struct Reference 730
Data Fields 730
Detailed Description 731
IxFpCodeletTemplate Struct Reference 731
Data Fields 731
Detailed Description 732
IxHssAccCodeletStats Struct Reference 732
Data Fields 732
Detailed Description 732
IxHssAccConfigParams Struct Reference 732
Data Fields 732
Detailed Description 733
IxHssAccPktHdlcFraming Struct Reference 733
Data Fields 733
Detailed Description 733
IxHssAccPortConfig Struct Reference 734
Data Fields 734
Detailed Description 735
IxMbufPool Struct Reference 735
Data Fields 735
Detailed Description 736
IxNpeA_AtmVcFp Struct Reference 736
Data Fields 736
Detailed Description 736
IxNpeA_NpePacketDescriptor Struct Reference 736
Data Fields 737
Detailed Description 737
Field Documentation 737
IxNpeA_RxAtmVc Struct Reference 738
Data Fields 738
Detailed Description 738
IxNpeA_TxAtmVc Struct Reference 739
Data Fields 739
Detailed Description 739
IxNpeDlImageId Struct Reference 739
Data Fields 740
Detailed Description 740
IxNpeMhMessage Struct Reference 740

Data Fields 740
Detailed Description 740
IxOamITU610Cell Struct Reference 741
Data Fields 741
Detailed Description 741
IxOamITU610GenericPayload Struct Reference 741
Data Fields 741
Detailed Description 741
IxOamITU610LbPayload Struct Reference 742
Data Fields 742
Detailed Description 742
IxOamITU610Payload Union Reference 742
Data Fields 742
Detailed Description 742
IxPerfProfAccBusPmuResults Struct Reference 743
Data Fields 743
Detailed Description 743
IxPerfProfAccXcycleResults Struct Reference 743
Data Fields 743
Detailed Description 743
IxPerfProfAccXscalePmuEvtCnt Struct Reference 744
Data Fields 744
Detailed Description 744
IxPerfProfAccXscalePmuResults Struct Reference 744
Data Fields 744
Detailed Description 745
IxPerfProfAccXscalePmuSamplePcProfile Struct Reference 745
Data Fields 745
Detailed Description 745
IxQMgrQInlinedReadWriteInfo Struct Reference 746
Data Fields 746
Detailed Description 746
ixUARTDev Struct Reference 747
Data Fields 747
Detailed Description 747
ixUARTStats Struct Reference 747
Data Fields 747
Detailed Description 748
LearningIpFlow Struct Reference 748
Data Fields 748
Detailed Description 748
PacketisedStats Struct Reference 748
Data Fields 749
Detailed Description 749
USBDevice Struct Reference 750
Data Fields 750
Detailed Description 750
USBDeviceCounters Struct Reference 750
Data Fields 750
Detailed Description 751

USBSetupPacket Struct Reference 751
Data Fields 751
Detailed Description 752

C.2 Aal5Statistics Struct Reference

Aal5StatisticsAal5StatisticsAal5StatisticsAal5StatisticsTypes For Gathering Statistics.

C.2.1 Data Fields

- unsigned int **totalPduCount**
- unsigned int **validPduCount**
- unsigned int **errorCount**

C.2.2 Detailed Description

Types For Gathering Statistics.

Definition at line 377 of file IxFpathAccCodeletDefines_p.h.

The documentation for this struct was generated from the following files:

- IxFpathAccCodeletDefines_p.h
- IxFpathAccCodeletMain_p.h

C.3 ActiveIpFlow Struct Reference

ActiveIpFlowActiveIpFlowActiveIpFlowActiveIpFlowStruct to hold IP flow information while active.

C.3.1 Data Fields

- BOOL aged
  Flow may be superceded.
- UINT32 ipAddress
  Destination IP address.
- UINT8 macAddress [6]
  Corresponding Mac Address.
- int age
  In units of Counting Intervals.

C.3.2 Detailed Description

Struct to hold IP flow information while active.

Definition at line 426 of file IxFpathAccCodeletDefines_p.h.

The documentation for this struct was generated from the following files:

- IxFpathAccCodeletDefines_p.h
- IxFpathAccCodeletMain_p.h

C.4 ChannelisedStats Struct Reference

ChannelisedStatsChannelisedStatsChannelisedStatsChannelisedStatsingroup IxHssAccCodeletCom

C.4.1 Data Fields

- UINT32 **txSamples**
- UINT32 **txBytes**
- UINT32 **rxSamples**
- UINT32 **rxBytes**
- UINT32 **rxIdles**
- UINT32 **rxVerifyFails**
- UINT32 **connectFails**
- UINT32 **portEnableFails**
- UINT32 **portDisableFails**
- UINT32 **disconnectFails**

C.4.2 Detailed Description

ingroup IxHssAccCodeletCom

brief Type definition structure for channelised statistics

Definition at line 93 of file IxHssAccCodeletCom.h.

The documentation for this struct was generated from the following file:

- IxHssAccCodeletCom.h

C.5 GeneralStats Struct Reference

GeneralStatsGeneralStatsGeneralStatsGeneralStatsingroup IxHssAccCodeletCom

C.5.1 Data Fields

- UINT32 **portInitFails**
- UINT32 **errorRetrievalFails**
- UINT32 **txFrmSyncErrors**
- UINT32 **rxFrmSyncErrors**
- UINT32 **txOverRunErrors**
- UINT32 **rxOverRunErrors**
- UINT32 **chanSwTxErrors**
- UINT32 **chanSwRxErrors**
- UINT32 **pktSwTxErrors**
- UINT32 **pktSwRxErrors**
- UINT32 **unrecognisedErrors**

C.5.2 Detailed Description

ingroup IxHssAccCodeletCom

brief Type definition structure for general statistics

Definition at line 69 of file IxHssAccCodeletCom.h.

The documentation for this struct was generated from the following file:

- IxHssAccCodeletCom.h

C.6 ix_ossl_thread_main_info_t Struct Reference

ix_ossl_thread_main_info_tix_ossl_thread_main_info_tix_ossl_thread_main_info_tix_ossl_thread_main_info_tThis type defines thread main info.

C.6.1 Data Fields

- ix_ossl_thread_entry_point_t threadMain
  pointer to thread entry point function
- void * threadArg
  void pointer to a user defined thread entry point argument

C.6.2 Detailed Description

This type defines 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.

Definition at line 259 of file ix_ossl.h.

The documentation for this struct was generated from the following file:

- ix_ossl.h

C.7 ix_ossl_time_t Struct Reference

ix_ossl_time_tix_ossl_time_tix_ossl_time_tix_ossl_time_tThis type defines OSSL time.

C.7.1 Data Fields

- unsigned long tv_sec
  seconds

- unsigned long tv_nsec
  nanoseconds [1, 999,999,999]

C.7.2 Detailed 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

Definition at line 221 of file ix_ossl.h.

The documentation for this struct was generated from the following file:

- ix_ossl.h

C.8 IxAtmCodeletStats Struct Reference

IxAtmCodeletStatsIxAtmCodeletStatsIxAtmCodeletStatsIxAtmCodeletStatsCodelet statistics.

C.8.1 Data Fields

- UINT32 **txPdus**
- UINT32 **txBytes**
- UINT32 **rxPdus**
- UINT32 **rxBytes**

- UINT32 **txDonePdus**
- UINT32 **rxFreeBuffers**
- UINT32 **txPdusSubmitFail**
- UINT32 **txPdusSubmitBusy**
- UINT32 **rxPdusInvalid**

C.8.2 Detailed Description

Codelet statistics.

Definition at line 147 of file IxAtmCodelet_p.h.

The documentation for this struct was generated from the following file:

- IxAtmCodelet_p.h

C.9 IxAtmdAccUtopiaConfig Struct Reference

IxAtmdAccUtopiaConfigIxAtmdAccUtopiaConfigIxAtmdAccUtopiaConfigIxAtmdAccUtopiaConfigUtopia configuration.

C.9.1 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.

C.9.2 Detailed Description

Utopia configuration.

This structure is used to set the Utopia parameters

- contains the values of Utopia registers, to be set during initialisation
- 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.

Definition at line 930 of file IxAtmdAccCtrl.h.

The documentation for this struct was generated from the following file:

- IxAtmdAccCtrl.h

C.10 IxAtmdAccUtopiaConfig::UtRxConfig_ Struct Reference

IxAtmdAccUtopiaConfig::UtRxConfig_IxAtmdAccUtopiaConfig::UtRxConfig_IxAtmdAccUtopiaConfig::UtRxConfig_IxAtmdAccUtopiaConfig::UtRxConfig_Utopia Rx config Register.

C.10.1 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

C.10.2 Detailed Description

Utopia Rx config Register.

Definition at line 1267 of file IxAtmdAccCtrl.h.

C.10.3 Field Documentation

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.

Definition at line 1329 of file IxAtmdAccCtrl.h.

unsigned int IxAtmdAccUtopiaConfig::UtRxConfig_::rxCellSize

[22:16] Receive cell size

Configures the receive cell size. Values between 52-64 are valid

Definition at line 1312 of file IxAtmdAccCtrl.h.

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.

  Definition at line 1299 of file IxAtmdAccCtrl.h.

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

  Definition at line 1315 of file IxAtmdAccCtrl.h.

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

Definition at line 1294 of file IxAtmdAccCtrl.h.

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.

Definition at line 1304 of file IxAtmdAccCtrl.h.

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

Definition at line 1270 of file IxAtmdAccCtrl.h.

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

Definition at line 1275 of file IxAtmdAccCtrl.h.

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

Definition at line 1280 of file IxAtmdAccCtrl.h.

unsigned int IxAtmdAccUtopiaConfig::UtRxConfig_::rxParity

[28] Utopia Receive Parity Checking enable

- 1 - Parity checking enabled
- 0 - Parity checking disabled

Definition at line 1286 of file IxAtmdAccCtrl.h.

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

  Definition at line 1321 of file IxAtmdAccCtrl.h.

The documentation for this struct was generated from the following file:

- IxAtmdAccCtrl.h

C.11 IxAtmdAccUtopiaConfig::UtRxDefineIdle_ Struct Reference

IxAtmdAccUtopiaConfig::UtRxDefineIdle_IxAtmdAccUtopiaConfig::UtRxDefineIdle_IxAtmdAccUtopiaConfig::UtRxDefineIdle_IxAtmdAccUtopiaConfig::UtRxDefineIdle_Utopia Rx idle cells config Register.

C.11.1 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.

C.11.2 Detailed Description

Utopia Rx idle cells config Register.

Definition at line 1369 of file IxAtmdAccCtrl.h.

The documentation for this struct was generated from the following file:

- IxAtmdAccCtrl.h

C.12 IxAtmdAccUtopiaConfig::UtRxEnableFields_ Struct Reference

IxAtmdAccUtopiaConfig::UtRxEnableFields_IxAtmdAccUtopiaConfig::UtRxEnableFields_IxAtmdAccUtopiaConfig::UtRxEnableFields_IxAtmdAccUtopiaConfig::UtRxEnableFields_Utopia Rx enable Register.

C.12.1 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

C.12.2 Detailed Description

Utopia Rx enable Register.

Definition at line 1389 of file IxAtmdAccCtrl.h.

C.12.3 Field Documentation

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.

Definition at line 1402 of file IxAtmdAccCtrl.h.

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.

Definition at line 1392 of file IxAtmdAccCtrl.h.

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.

Definition at line 1397 of file IxAtmdAccCtrl.h.

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

Definition at line 1434 of file IxAtmdAccCtrl.h.

unsigned int IxAtmdAccUtopiaConfig::UtRxEnableFields_::discardParErr

[22] Discard cells containing parity errors

- 1 - Discard cells with parity errors
- 0 - Cells with parity errors are passed

Definition at line 1438 of file IxAtmdAccCtrl.h.

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.

Definition at line 1446 of file IxAtmdAccCtrl.h.

unsigned int IxAtmdAccUtopiaConfig::UtRxEnableFields_::enbIdleCellCnt

[18] Enable Receive Idle Cell Count

- 1 - Enable count of Idle cells received.
- 0 - No count is maintained.

Definition at line 1454 of file IxAtmdAccCtrl.h.

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.

Definition at line 1462 of file IxAtmdAccCtrl.h.

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.

Definition at line 1458 of file IxAtmdAccCtrl.h.

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

Definition at line 1473 of file IxAtmdAccCtrl.h.

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

Definition at line 1478 of file IxAtmdAccCtrl.h.

unsigned int IxAtmdAccUtopiaConfig::UtRxEnableFields_::invalidSizeOvrInt

[9] Enable CBI Receive Status Condition if the InvalidSizeCount register overflows

- 1 - CBI Receive Status Condition asserted.
- ¸0 - No CBI Receive Status asserted

Definition at line 1483 of file IxAtmdAccCtrl.h.

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.

Definition at line 1407 of file IxAtmdAccCtrl.h.

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.

Definition at line 1494 of file IxAtmdAccCtrl.h.

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.

  Definition at line 1468 of file IxAtmdAccCtrl.h.

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

  Definition at line 1488 of file IxAtmdAccCtrl.h.

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.

  Definition at line 1429 of file IxAtmdAccCtrl.h.

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.

  Definition at line 1413 of file IxAtmdAccCtrl.h.

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.

  Definition at line 1418 of file IxAtmdAccCtrl.h.

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.

  Definition at line 1424 of file IxAtmdAccCtrl.h.

The documentation for this struct was generated from the following file:

- IxAtmdAccCtrl.h

C.13 IxAtmdAccUtopiaConfig::UtRxStatsConfig_ Struct Reference

IxAtmdAccUtopiaConfig::UtRxStatsConfig_IxAtmdAccUtopiaConfig::UtRxStatsConfig_IxAtmdAccUtopiaConfig::UtRxStatsConfig_IxAtmdAccUtopiaConfig::UtRxStatsConfig_Utopia Rx stats config Register.

C.13.1 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

C.13.2 Detailed Description

Utopia Rx stats config Register.

Definition at line 1345 of file IxAtmdAccCtrl.h.

C.13.3 Field Documentation

unsigned int IxAtmdAccUtopiaConfig::UtRxStatsConfig_::clp

[0] CLP [0] or PHY Address [0]

Note: if VCStatsRxCLP is set to 0 the CLP field is ignored in test

Note: if VCStatsRxEnb is set to 0 only the PHY port address is used for statistics gathering..

Definition at line 1358 of file IxAtmdAccCtrl.h.

unsigned int IxAtmdAccUtopiaConfig::UtRxStatsConfig_::pti

[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

Note: if VCStatsRxEnb is set to 0 only the PHY port address is used for statistics gathering..

Definition at line 1353 of file IxAtmdAccCtrl.h.

The documentation for this struct was generated from the following file:

- IxAtmdAccCtrl.h

C.14 IxAtmdAccUtopiaConfig::UtRxTransTable0_ Struct Reference

IxAtmdAccUtopiaConfig::UtRxTransTable0_IxAtmdAccUtopiaConfig::UtRxTransTable0_IxAtmdAccUtopiaConfig::UtRxTransTable0_IxAtmdAccUtopiaConfig::UtRxTransTable0_Utopia Rx translation table Register.

C.14.1 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

C.14.2 Detailed Description

Utopia Rx translation table Register.

Definition at line 1505 of file IxAtmdAccCtrl.h.

The documentation for this struct was generated from the following file:

- IxAtmdAccCtrl.h

C.15 IxAtmdAccUtopiaConfig::UtRxTransTable1_ Struct Reference

IxAtmdAccUtopiaConfig::UtRxTransTable1_IxAtmdAccUtopiaConfig::UtRxTransTable1_IxAtmdAccUtopiaConfig::UtRxTransTable1_IxAtmdAccUtopiaConfig::UtRxTransTable1_Utopia Rx translation table Register.

C.15.1 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

C.15.2 Detailed Description

Utopia Rx translation table Register.

Definition at line 1532 of file IxAtmdAccCtrl.h.

The documentation for this struct was generated from the following file:

- IxAtmdAccCtrl.h

C.16 IxAtmdAccUtopiaConfig::UtRxTransTable2_ Struct Reference

IxAtmdAccUtopiaConfig::UtRxTransTable2_IxAtmdAccUtopiaConfig::UtRxTransTable2_IxAtmdAccUtopiaConfig::UtRxTransTable2_IxAtmdAccUtopiaConfig::UtRxTransTable2_Utopia Rx translation table Register.

C.16.1 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
- Uunsigned 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

C.16.2 Detailed Description

Utopia Rx translation table Register.

Definition at line 1559 of file IxAtmdAccCtrl.h.

The documentation for this struct was generated from the following file:

- IxAtmdAccCtrl.h

C.17 IxAtmdAccUtopiaConfig::UtRxTransTable3_ Struct Reference

IxAtmdAccUtopiaConfig::UtRxTransTable3_IxAtmdAccUtopiaConfig::UtRxTransTable3_IxAtmdAccUtopiaConfig::UtRxTransTable3_IxAtmdAccUtopiaConfig::UtRxTransTable3_Utopia Rx translation table Register.

C.17.1 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

C.17.2 Detailed Description

Utopia Rx translation table Register.

Definition at line 1584 of file IxAtmdAccCtrl.h.

The documentation for this struct was generated from the following file:

- IxAtmdAccCtrl.h

C.18 IxAtmdAccUtopiaConfig::UtRxTransTable4_ Struct Reference

IxAtmdAccUtopiaConfig::UtRxTransTable4_IxAtmdAccUtopiaConfig::UtRxTransTable4_IxAtmdAccUtopiaConfig::UtRxTransTable4_IxAtmdAccUtopiaConfig::UtRxTransTable4_Utopia Rx translation table Register.

C.18.1 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

C.18.2 Detailed Description

Utopia Rx translation table Register.

Definition at line 1609 of file IxAtmdAccCtrl.h.

The documentation for this struct was generated from the following file:

- IxAtmdAccCtrl.h

C.19 IxAtmdAccUtopiaConfig::UtRxTransTable5_ Struct Reference

IxAtmdAccUtopiaConfig::UtRxTransTable5_IxAtmdAccUtopiaConfig::UtRxTransTable5_IxAtmdAccUtopiaConfig::UtRxTransTable5_IxAtmdAccUtopiaConfig::UtRxTransTable5_Utopia Rx translation table Register.

C.19.1 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

C.19.2 Detailed Description

Utopia Rx translation table Register.

Definition at line 1634 of file IxAtmdAccCtrl.h.

The documentation for this struct was generated from the following file:

- IxAtmdAccCtrl.h

C.20 IxAtmdAccUtopiaConfig::UtSysConfig_ Struct Reference

IxAtmdAccUtopiaConfig::UtSysConfig_IxAtmdAccUtopiaConfig::UtSysConfig_IxAtmdAccUtopiaConfig::UtSysConfig_IxAtmdAccUtopiaConfig::UtSysConfig_NPE setup Register.

C.20.1 Data Fields

- unsigned int reserved_1:2
  [31-30] These bits are always 0
- unsigned int txEnbFSM:1
  [29] Enables the operation ofthe Utopia Transmit FSM
- 1 - FSM enabled
- 0 - FSM inactive
- unsigned int rxEnbFSM:1
  [28] Enables the operation ofthe Utopia Revieve 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

C.20.2 Detailed Description

NPE setup Register.

Definition at line 1648 of file IxAtmdAccCtrl.h.

C.20.3 Field Documentation

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
  Definition at line 1660 of file IxAtmdAccCtrl.h.

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
  Definition at line 1680 of file IxAtmdAccCtrl.h.

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
  Definition at line 1666 of file IxAtmdAccCtrl.h.

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
  Definition at line 1673 of file IxAtmdAccCtrl.h.

The documentation for this struct was generated from the following file:

- IxAtmdAccCtrl.h

C.21 IxAtmdAccUtopiaConfig::UtTxConfig_ Struct Reference

IxAtmdAccUtopiaConfig::UtTxConfig_IxAtmdAccUtopiaConfig::UtTxConfig_IxAtmdAccUtopiaConfig::UtTxConfig_IxAtmdAccUtopiaConfig::UtTxConfig_Utopia Tx Config Register.

C.21.1 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

C.21.2 Detailed Description

Utopia Tx Config Register.

Definition at line 937 of file IxAtmdAccCtrl.h.

C.21.3 Field Documentation

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.

Definition at line 984 of file IxAtmdAccCtrl.h.

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.

Definition at line 980 of file IxAtmdAccCtrl.h.

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.

Definition at line 972 of file IxAtmdAccCtrl.h.

unsigned int IxAtmdAccUtopiaConfig::UtTxConfig_::txEvenParity

[26] Utopia Transmit Parity Mode

- 1 - Even Parity Generated
- 0 - Odd Parity Generated.

Definition at line 963 of file IxAtmdAccCtrl.h.

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.

Definition at line 967 of file IxAtmdAccCtrl.h.

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

Definition at line 941 of file IxAtmdAccCtrl.h.

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

Definition at line 947 of file IxAtmdAccCtrl.h.

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

Definition at line 952 of file IxAtmdAccCtrl.h.

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.
  Definition at line 958 of file IxAtmdAccCtrl.h.

The documentation for this struct was generated from the following file:

- IxAtmdAccCtrl.h

C.22 IxAtmdAccUtopiaConfig::UtTxDefineIdle_ Struct Reference

IxAtmdAccUtopiaConfig::UtTxDefineIdle_IxAtmdAccUtopiaConfig::UtTxDefineIdle_IxAtmdAccUtopiaConfig::UtTxDefineIdle_IxAtmdAccUtopiaConfig::UtTxDefineIdle_Utopia Tx idle cells Register.

C.22.1 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.

C.22.2 Detailed Description

Utopia Tx idle cells Register.

Definition at line 1027 of file IxAtmdAccCtrl.h.

The documentation for this struct was generated from the following file:

- IxAtmdAccCtrl.h

C.23 IxAtmdAccUtopiaConfig::UtTxEnableFields_ Struct Reference

IxAtmdAccUtopiaConfig::UtTxEnableFields_IxAtmdAccUtopiaConfig::UtTxEnableFields_IxAtmdAccUtopiaConfig::UtTxEnableFields_IxAtmdAccUtopiaConfig::UtTxEnableFields_Utopia Tx ienable fields Register.

C.23.1 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

C.23.2 Detailed Description

Utopia Tx ienable fields Register.

Definition at line 1049 of file IxAtmdAccCtrl.h.

C.23.3 Field Documentation

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.

Definition at line 1062 of file IxAtmdAccCtrl.h.

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.

Definition at line 1052 of file IxAtmdAccCtrl.h.

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.

Definition at line 1057 of file IxAtmdAccCtrl.h.

unsigned int IxAtmdAccUtopiaConfig::UtTxEnableFields_::enbIdleCellCnt

[17] Enable Transmit Idle Cell Count

- 1 - Enable count of Idle cells transmitted.

- 0 - No count is maintained.

Definition at line 1112 of file IxAtmdAccCtrl.h.

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.

Definition at line 1067 of file IxAtmdAccCtrl.h.

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

Definition at line 1103 of file IxAtmdAccCtrl.h.

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

Definition at line 1098 of file IxAtmdAccCtrl.h.

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.

Definition at line 1091 of file IxAtmdAccCtrl.h.

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

Definition at line 1073 of file IxAtmdAccCtrl.h.

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.

Definition at line 1079 of file IxAtmdAccCtrl.h.

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.

Definition at line 1086 of file IxAtmdAccCtrl.h.

The documentation for this struct was generated from the following file:

- IxAtmdAccCtrl.h

C.24 IxAtmdAccUtopiaConfig::UtTxStatsConfig_ Struct Reference

IxAtmdAccUtopiaConfig::UtTxStatsConfig_IxAtmdAccUtopiaConfig::UtTxStatsConfig_IxAtmdAccUtopiaConfig::UtTxStatsConfig_IxAtmdAccUtopiaConfig::UtTxStatsConfig_Utopia Tx stats Register.

C.24.1 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

C.24.2 Detailed Description

Utopia Tx stats Register.

Definition at line 1001 of file IxAtmdAccCtrl.h.

C.24.3 Field Documentation

unsigned int IxAtmdAccUtopiaConfig::UtTxStatsConfig_::clp

[0] ATM CLP or PHY Address [0]

Note: if VCStatsTxCLP is set to 0 the CLP field is ignored in test

Note: if VCStatsTxEnb is set to 0 only the transmit PHY port address as defined by this register is used for ATM statistics [4:0].

Definition at line 1014 of file IxAtmdAccCtrl.h.

unsigned int IxAtmdAccUtopiaConfig::UtTxStatsConfig_::pti

[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

Note: if VCStatsTxEnb is set to 0 only the transmit PHY port address as defined by this register is used for ATM statistics [4:0].

Definition at line 1009 of file IxAtmdAccCtrl.h.

The documentation for this struct was generated from the following file:

- IxAtmdAccCtrl.h

C.25 IxAtmdAccUtopiaConfig::UtTxTransTable0_ Struct Reference

IxAtmdAccUtopiaConfig::UtTxTransTable0_IxAtmdAccUtopiaConfig::UtTxTransTable0_IxAtmdAccUtopiaConfig::UtTxTransTable0_IxAtmdAccUtopiaConfig::UtTxTransTable0_Utopia Tx translation table Register.

C.25.1 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

C.25.2 Detailed Description

Utopia Tx translation table Register.

Definition at line 1128 of file IxAtmdAccCtrl.h.

The documentation for this struct was generated from the following file:

- IxAtmdAccCtrl.h

C.26 IxAtmdAccUtopiaConfig::UtTxTransTable1_ Struct Reference

IxAtmdAccUtopiaConfig::UtTxTransTable1_IxAtmdAccUtopiaConfig::UtTxTransTable1_IxAtmdAccUtopiaConfig::UtTxTransTable1_IxAtmdAccUtopiaConfig::UtTxTransTable1_Utopia Tx translation table Register.

C.26.1 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

C.26.2 Detailed Description

Utopia Tx translation table Register.

Definition at line 1153 of file IxAtmdAccCtrl.h.

The documentation for this struct was generated from the following file:

- IxAtmdAccCtrl.h

C.27 IxAtmdAccUtopiaConfig::UtTxTransTable2_ Struct Reference

IxAtmdAccUtopiaConfig::UtTxTransTable2_IxAtmdAccUtopiaConfig::UtTxTransTable2_IxAtmdAccUtopiaConfig::UtTxTransTable2_IxAtmdAccUtopiaConfig::UtTxTransTable2_Utopia Tx translation table Register.

C.27.1 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

C.27.2 Detailed Description

Utopia Tx translation table Register.

Definition at line 1178 of file IxAtmdAccCtrl.h.

The documentation for this struct was generated from the following file:

- IxAtmdAccCtrl.h

C.28 IxAtmdAccUtopiaConfig::UtTxTransTable3_ Struct Reference

IxAtmdAccUtopiaConfig::UtTxTransTable3_IxAtmdAccUtopiaConfig::UtTxTransTable3_IxAtmdAccUtopiaConfig::UtTxTransTable3_IxAtmdAccUtopiaConfig::UtTxTransTable3_Utopia Tx translation table Register.

C.28.1 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

C.28.2 Detailed Description

Utopia Tx translation table Register.

Definition at line 1203 of file IxAtmdAccCtrl.h.

The documentation for this struct was generated from the following file:

- IxAtmdAccCtrl.h

C.29 IxAtmdAccUtopiaConfig::UtTxTransTable4_ Struct Reference

IxAtmdAccUtopiaConfig::UtTxTransTable4_IxAtmdAccUtopiaConfig::UtTxTransTable4_IxAtmdAccUtopiaConfig::UtTxTransTable4_IxAtmdAccUtopiaConfig::UtTxTransTable4_Utopia Tx translation table Register.

C.29.1 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

C.29.2 Detailed Description

Utopia Tx translation table Register.

Definition at line 1228 of file IxAtmdAccCtrl.h.

The documentation for this struct was generated from the following file:

- IxAtmdAccCtrl.h

C.30 IxAtmdAccUtopiaConfig::UtTxTransTable5_ Struct Reference

IxAtmdAccUtopiaConfig::UtTxTransTable5_IxAtmdAccUtopiaConfig::UtTxTransTable5_IxAtmdAccUtopiaConfig::UtTxTransTable5_IxAtmdAccUtopiaConfig::UtTxTransTable5_Utopia Tx translation table Register.

C.30.1 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

C.30.2 Detailed Description

Utopia Tx translation table Register.

Definition at line 1253 of file IxAtmdAccCtrl.h.

The documentation for this struct was generated from the following file:

- IxAtmdAccCtrl.h

C.31 IxAtmdAccUtopiaStatus Struct Reference

IxAtmdAccUtopiaStatusIxAtmdAccUtopiaStatusIxAtmdAccUtopiaStatusIxAtmdAccUtopiaStatus Utopia status.

C.31.1 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.

C.31.2 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.

Definition at line 1704 of file IxAtmdAccCtrl.h.

The documentation for this struct was generated from the following file:

- IxAtmdAccCtrl.h

C.32 IxAtmdAccUtopiaStatus::UtRxCellConditionStatus_ Struct Reference

IxAtmdAccUtopiaStatus::UtRxCellConditionStatus_IxAtmdAccUtopiaStatus::UtRxCellCondition Status_IxAtmdAccUtopiaStatus::UtRxCellConditionStatus_IxAtmdAccUtopiaStatus::UtRxCellC onditionStatus_Utopia Rx Status Register.

C.32.1 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

C.32.2 Detailed Description

Utopia Rx Status Register.

Definition at line 1763 of file IxAtmdAccCtrl.h.

The documentation for this struct was generated from the following file:

- IxAtmdAccCtrl.h

C.33 IxAtmdAccUtopiaStatus::UtTxCellConditionStatus_ Struct Reference

IxAtmdAccUtopiaStatus::UtTxCellConditionStatus_IxAtmdAccUtopiaStatus::UtTxCellCondition Status_IxAtmdAccUtopiaStatus::UtTxCellConditionStatus_IxAtmdAccUtopiaStatus::UtTxCellCo nditionStatus_Utopia Tx Status Register.

C.33.1 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

C.33.2 Detailed Description

Utopia Tx Status Register.

Definition at line 1716 of file IxAtmdAccCtrl.h.

The documentation for this struct was generated from the following file:

- IxAtmdAccCtrl.h

C.34 IxAtmmPortCfg Struct Reference

IxAtmmPortCfgIxAtmmPortCfgIxAtmmPortCfgIxAtmmPortCfgStructure contains port-specific information required to initialize IxAtmm, and specifically, the IXP425 UTOPIA Level-2 device.

C.34.1 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

C.34.2 Detailed Description

Structure contains port-specific information required to initialize IxAtmm, and specifically, the IXP425 UTOPIA Level-2 device.

Definition at line 184 of file IxAtmm.h.

The documentation for this struct was generated from the following file:

- IxAtmm.h

C.35 IxAtmmVc Struct Reference

IxAtmmVcIxAtmmVcIxAtmmVcIxAtmmVcThis structure describes the required attributes of a virtual connection.

C.35.1 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.

C.35.2 Detailed Description

This structure describes the required attributes of a virtual connection.

Definition at line 159 of file IxAtmm.h.

C.35.3 Field Documentation

IxAtmTrafficDescriptor IxAtmmVc::trafficDesc

Traffic descriptor of this virtual connection.

This structure is defined by the IXP425 ATM Transmit Scheduler (IxAtmSch) API component.

Definition at line 166 of file IxAtmm.h.

The documentation for this struct was generated from the following file:

- IxAtmm.h

C.36 IxAtmScheduleTable Struct Reference

IxAtmScheduleTableIxAtmScheduleTableIxAtmScheduleTableIxAtmScheduleTableThis 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.

C.36.1 Data Fields

- unsigned tableSize
  Number of entries.
- unsigned totalCellSlots
  Number of cells.
- IxAtmScheduleTableEntry * table
  Pointer to schedule entries.

C.36.2 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.

Definition at line 378 of file IxAtmTypes.h.

C.36.3 Field Documentation

IxAtmScheduleTableEntry* IxAtmScheduleTable::table

Pointer to schedule entries.

Pointer to an array containing tableSize entries

Definition at line 391 of file IxAtmTypes.h.

unsigned IxAtmScheduleTable::tableSize

Number of entries.

Indicates the total number of entries in the table.

Definition at line 380 of file IxAtmTypes.h.

unsigned IxAtmScheduleTable::totalCellSlots

Number of cells.

Indicates the total number of ATM cells which are scheduled by all the entries in the table.

Definition at line 385 of file IxAtmTypes.h.

The documentation for this struct was generated from the following file:

- IxAtmTypes.h

C.37 IxAtmScheduleTableEntry Struct Reference

IxAtmScheduleTableEntryIxAtmScheduleTableEntryIxAtmScheduleTableEntryIxAtmScheduleTableEntryATM Schedule Table entry.

C.37.1 Data Fields

- IxAtmConnId connId
  connection Id
- unsigned int numberOfCells
  number of cells to transmit

C.37.2 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

- the number of cells to be transmitted (numberOfCells)
- the 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.

Definition at line 346 of file IxAtmTypes.h.

C.37.3 Field Documentation

IxAtmConnId IxAtmScheduleTableEntry::connId

connection Id

Identifier of VC from which cells are to be transmitted. When this valus 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.

Definition at line 348 of file IxAtmTypes.h.

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. OverSchduling results in the transmission of idle cells.

Definition at line 356 of file IxAtmTypes.h.

The documentation for this struct was generated from the following file:

- IxAtmTypes.h

C.38 IxAtmTrafficDescriptor Struct Reference

IxAtmTrafficDescriptorIxAtmTrafficDescriptorIxAtmTrafficDescriptorIxAtmTrafficDescriptorStructure describing an ATM traffic contract for a Virtual Connection (VC).

C.38.1 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.

C.38.2 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.

These parameters are defined by the ATM forum working group (http://www.atmforum.com).

Typical values for a voice channel 64 Kbit/s

- atmService *IX_ATM_RTVBR*
- pcr 400 (include IP overhead, and AAL5 trailer)
- cdvt 5000000 (5 ms)
- scr = pcr

Typical values for a data channel 800 Kbit/s

- atmService *IX_ATM_UBR*
- pcr 1962 (include IP overhead, and AAL5 trailer)
- cdvt 5000000 (5 ms)

Definition at line 284 of file IxAtmTypes.h.

The documentation for this struct was generated from the following file:

- IxAtmTypes.h

C.39 IxCryptoAccAuthCtx Struct Reference

IxCryptoAccAuthCtxIxCryptoAccAuthCtxIxCryptoAccAuthCtxIxCryptoAccAuthCtxStructure storing authentication configuration parameters required to perform security functionality.

C.39.1 Data Fields

- IxCryptoAccAuthAlgo authAlgo
  authentication algorithm - MD5, SHA1
- UINT32 authDigestLen
  Digest length in bytes.
- UINT32 authKeyLen
  Hash key length in bytes.

- union {
- UINT8 authKey [IX_CRYPTO_ACC_MAX_AUTH_KEY_LENGTH]
- UINT8 sha1Key [IX_CRYPTO_ACC_MAX_AUTH_KEY_LENGTH]
- UINT8 md5Key [IX_CRYPTO_ACC_MAX_AUTH_KEY_LENGTH]
- } key

  Hash key.

C.39.2 Detailed Description

Structure storing authentication configuration parameters required to perform security functionality.

Structure is used to specify the authentication context required for using the IXP425 Security (IxCryptoAcc) API interface.

Definition at line 403 of file IxCryptoAcc.h.

C.39.3 Field Documentation

UINT8 IxCryptoAccAuthCtx::authKey[IX_CRYPTO_ACC_MAX_AUTH_KEY_LENGTH]

default hash key array

Definition at line 412 of file IxCryptoAcc.h.

UINT8 IxCryptoAccAuthCtx::md5Key[IX_CRYPTO_ACC_MAX_AUTH_KEY_LENGTH]

MD5 key.

Definition at line 416 of file IxCryptoAcc.h.

UINT8 IxCryptoAccAuthCtx::sha1Key[IX_CRYPTO_ACC_MAX_AUTH_KEY_LENGTH]

SHA1 key.

Definition at line 415 of file IxCryptoAcc.h.

The documentation for this struct was generated from the following file:

- IxCryptoAcc.h

C.40 IxCryptoAccCipherCtx Struct Reference

IxCryptoAccCipherCtxIxCryptoAccCipherCtxIxCryptoAccCipherCtxIxCryptoAccCipherCtxStructure storing cipher configuration parameters required to perform security functionality.

C.40.1 Data Fields

- IxCryptoAccCipherAlgo cipherAlgo

Cipher Algorithm - DES, 3DES, AES.

- IxCryptoAccCipherMode cipherMode
  Cipher mode - ECB, CBC, CTR.
- UINT32 cipherKeyLen
  Cipher key length in bytes.
- union {
- UINT8 cipherKey [IX_CRYPTO_ACC_MAX_CIPHER_KEY_LENGTH]
- UINT8 desKey [IX_CRYPTO_ACC_DES_KEY_64]
- UINT8 tripleDesKey [IX_CRYPTO_ACC_3DES_KEY_192]
- UINT8 aesKey128 [IX_CRYPTO_ACC_AES_KEY_128]
- UINT8 aesKey192 [IX_CRYPTO_ACC_AES_KEY_192]
- UINT8 aesKey256 [IX_CRYPTO_ACC_AES_KEY_256]
- } key
  Cipher key.
- UINT32 cipherBlockLen
  Cipher block length in bytes.
- UINT32 cipherInitialVectorLen
  Length of IV in bytes.
- UINT32 securityParameterIndex
  SPI is used in generating counter block for AES CTR mode.

C.40.2 Detailed Description

Structure storing cipher configuration parameters required to perform security functionality.

Structure is used to specify the cipher context required for using the IXP425 Security (IxCryptoAcc) API interface.

Definition at line 365 of file IxCryptoAcc.h.

C.40.3 Field Documentation

UINT8 IxCryptoAccCipherCtx::aesKey128[IX_CRYPTO_ACC_AES_KEY_128]

AES-128 key.

Definition at line 379 of file IxCryptoAcc.h.

UINT8 IxCryptoAccCipherCtx::aesKey192[IX_CRYPTO_ACC_AES_KEY_192]

AES-192 key.

Definition at line 380 of file IxCryptoAcc.h.

UINT8 IxCryptoAccCipherCtx::aesKey256[IX_CRYPTO_ACC_AES_KEY_256]

AES-256 key.

Definition at line 381 of file IxCryptoAcc.h.

UINT32 IxCryptoAccCipherCtx::cipherBlockLen

Cipher block length in bytes.

(DES/3DES - 8 bytes, AES - 16 bytes)

Definition at line 383 of file IxCryptoAcc.h.

UINT8 IxCryptoAccCipherCtx::cipherKey[IX_CRYPTO_ACC_MAX_CIPHER_KEY_LENGTH]

default key array

Definition at line 374 of file IxCryptoAcc.h.

UINT8 IxCryptoAccCipherCtx::desKey[IX_CRYPTO_ACC_DES_KEY_64]

DES key.

Definition at line 377 of file IxCryptoAcc.h.

UINT8 IxCryptoAccCipherCtx::tripleDesKey[IX_CRYPTO_ACC_3DES_KEY_192]

3DES key

Definition at line 378 of file IxCryptoAcc.h.

The documentation for this struct was generated from the following file:

- IxCryptoAcc.h

C.41 IxCryptoAccCtx Struct Reference

IxCryptoAccCtxIxCryptoAccCtxIxCryptoAccCtxIxCryptoAccCtxStructure storing configuration parameters required to perform security functionality.

C.41.1 Data Fields

- IxCryptoAccOperation operation
  Types of operation.
- IxCryptoAccCipherCtx cipherCtx
  Cipher context.
- IxCryptoAccAuthCtx authCtx
  Authentication context.
- BOOL useDifferentSrcAndDestMbufs
  If TRUE, data is read from srcMbuf, result is written to destMbuf (non in-place operation).

C.41.2 Detailed Description

Structure storing configuration parameters required to perform security functionality.

Structure is used to specify the crypto context (hardware accelerator context) required for using the IXP425 Security (IxCryptoAcc) API interface.

Definition at line 430 of file IxCryptoAcc.h.

C.41.3 Field Documentation

BOOL IxCryptoAccCtx::useDifferentSrcAndDestMbufs

If TRUE, data is read from srcMbuf, result is written to destMbuf (non in-place operation).

If FALSE data is read from srcMbuf, and written back to srcMbuf (in-place operation). Default is FALSE. Note that only the crypted/authenticated data that is copied, not the entire source mbuf.

Definition at line 435 of file IxCryptoAcc.h.

The documentation for this struct was generated from the following file:

- IxCryptoAcc.h

C.42 IxEthAccMacAddr Struct Reference

IxEthAccMacAddrIxEthAccMacAddrIxEthAccMacAddrIxEthAccMacAddrThe IEEE 802.3 Ethernet MAC address structure.

C.42.1 Data Fields

- UINT8 macAddress [IX_IEEE803_MAC_ADDRESS_SIZE]
  MAC address.

C.42.2 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.

Definition at line 110 of file IxEthAcc.h.

The documentation for this struct was generated from the following file:

- IxEthAcc.h

C.43 IxEthDBMacAddr Struct Reference

IxEthDBMacAddrIxEthDBMacAddrIxEthDBMacAddrIxEthDBMacAddrThe IEEE 802.3 Ethernet MAC address structure.

C.43.1 Data Fields

- UINT8 **macAddress** [IX_IEEE803_MAC_ADDRESS_SIZE]

C.43.2 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.

Definition at line 103 of file IxEthDB.h.

The documentation for this struct was generated from the following file:

- IxEthDB.h

C.44 IxEthDBPortDefinition Struct Reference

IxEthDBPortDefinitionIxEthDBPortDefinitionIxEthDBPortDefinitionIxEthDBPortDefinitionPort Definition - a structure contains the Port type and capabilities.

C.44.1 Data Fields

- IxEthDBPortType **type**
- IxEthDBPortCapability **capabilities**

C.44.2 Detailed Description

Port Definition - a structure contains the Port type and capabilities.

Definition at line 74 of file IxEthDBPortDefs.h.

The documentation for this struct was generated from the following file:

- IxEthDBPortDefs.h

C.45 IxEthEthObjStats Struct Reference

IxEthEthObjStatsIxEthEthObjStatsIxEthEthObjStatsIxEthEthObjStatsThis 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.

C.45.1 Data Fields

- UINT32 dot3StatsAlignmentErrors
  link error count
- UINT32 dot3StatsFCSErrors
  link error count

- UINT32 dot3StatsFrameTooLongs
  link error count
- UINT32 dot3StatsInternalMacReceiveErrors
  link error count
- UINT32 LearnedEntryDiscards
  NPE error count.
- UINT32 dot3StatsSingleCollisionFrames
  link error count
- UINT32 dot3StatsMultipleCollisionFrames
  link error count
- UINT32 dot3StatsDeferredTransmissions
  link error count
- UINT32 dot3StatsLateCollisions
  link error count
- UINT32 dot3StatsExcessiveCollsions
  link error count
- UINT32 dot3StatsInternalMacTransmitErrors
  link error count
- UINT32 dot3StatsCarrierSenseErrors
  link error count

C.45.2 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.

Definition at line 1331 of file IxEthAcc.h.

The documentation for this struct was generated from the following file:

- IxEthAcc.h

C.46 IxFpCodeletFastPathConnection Struct Reference

IxFpCodeletFastPathConnectionIxFpCodeletFastPathConnectionIxFpCodeletFastPathConnectionIxFpCodeletFastPathConnectionMain structure to hold connection information.

C.46.1 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**

C.46.2 Detailed Description

Main structure to hold connection information.

Definition at line 435 of file IxFpathAccCodeletDefines_p.h.

The documentation for this struct was generated from the following files:

- IxFpathAccCodeletDefines_p.h
- IxFpathAccCodeletMain_p.h

C.47 IxFpCodeletTemplate Struct Reference

IxFpCodeletTemplateIxFpCodeletTemplateIxFpCodeletTemplateIxFpCodeletTemplateThis type describes a classifier or modifier template.

C.47.1 Data Fields

- UINT8 * data
  pointer to the template data
- unsigned int len
  the length of the template data
- UINT8 * data
  pointer to the template data

C.47.2 Detailed Description

This type describes a classifier or modifier template.

Definition at line 367 of file IxFpathAccCodeletDefines_p.h.

The documentation for this struct was generated from the following files:

- IxFpathAccCodeletDefines_p.h
- IxFpathAccCodeletTemplates_p.h

C.48 IxHssAccCodeletStats Struct Reference

IxHssAccCodeletStatsIxHssAccCodeletStatsIxHssAccCodeletStatsIxHssAccCodeletStatsingroup IxHssAccCodeletCom

C.48.1 Data Fields

- GeneralStats **gen**
- ChannelisedStats **chan**
- PacketisedStats **pkt** [IX_HSSACC_HDLC_PORT_MAX]

C.48.2 Detailed Description

ingroup IxHssAccCodeletCom

brief Type definition structure for HSS Access Codelet statistics

Definition at line 145 of file IxHssAccCodeletCom.h.

The documentation for this struct was generated from the following file:

- IxHssAccCodeletCom.h

C.49 IxHssAccConfigParams Struct Reference

IxHssAccConfigParamsIxHssAccConfigParamsIxHssAccConfigParamsIxHssAccConfigParamsStructure containing HSS configuration parameters.

C.49.1 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 packetizedIdlePattern
  The data to be transmitted on packetised service when there is no client data to tx.
- IxHssAccClkSpeed clkSpeed
  The HSS clock speed.

C.49.2 Detailed Description

Structure containing HSS configuration parameters.

Definition at line 536 of file IxHssAcc.h.

The documentation for this struct was generated from the following file:

- IxHssAcc.h

C.50 IxHssAccPktHdlcFraming Struct Reference

IxHssAccPktHdlcFramingIxHssAccPktHdlcFramingIxHssAccPktHdlcFramingIxHssAccPktHdlcFramingThis structure contains information required by the NPE to configure the HDLC co-processor.

C.50.1 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.

C.50.2 Detailed Description

This structure contains information required by the NPE to configure the HDLC co-processor.

Definition at line 560 of file IxHssAcc.h.

The documentation for this struct was generated from the following file:

- IxHssAcc.h

C.51 IxHssAccPortConfig Struct Reference

IxHssAccPortConfigIxHssAccPortConfigIxHssAccPortConfigIxHssAccPortConfigStructure containing HSS port configuration parameters.

C.51.1 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)

C.51.2 Detailed Description

Structure containing HSS port configuration parameters.

Note: All of these are used for TX. Only some are specific to RX.

Definition at line 497 of file IxHssAcc.h.

The documentation for this struct was generated from the following file:

- IxHssAcc.h

C.52 IxMbufPool Struct Reference

IxMbufPoolIxMbufPoolIxMbufPoolIxMbufPoolImplementation of buffer pool structure for use with non-VxWorks OS.

C.52.1 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.
- IxMbufPoolAllocationType **poolAllocType**

C.52.2 Detailed Description

Implementation of buffer pool structure for use with non-VxWorks OS.

Definition at line 269 of file IxOsBuffPoolMgt.h.

The documentation for this struct was generated from the following file:

- IxOsBuffPoolMgt.h

C.53 IxNpeA_AtmVcFp Struct Reference

IxNpeA_AtmVcFpIxNpeA_AtmVcFpIxNpeA_AtmVcFpIxNpeA_AtmVcFpAtm Descriptor structure used for fpath.

C.53.1 Data Fields

- IxNpeA_RxAtmVc npe
  Npe A.
- UINT32 masterMbufLen
  Mbuf length.
- IX_MBUF * pRootMbuf
  Pointer to root Mbuf.

C.53.2 Detailed Description

Atm Descriptor structure used for fpath.

Definition at line 1145 of file IxNpeA.h.

The documentation for this struct was generated from the following file:

- IxNpeA.h

C.54 IxNpeA_NpePacketDescriptor Struct Reference

IxNpeA_NpePacketDescriptorIxNpeA_NpePacketDescriptorIxNpeA_NpePacketDescriptorIxNpeA_NpePacketDescriptorHSS Packetized NpePacket Descriptor Structure.

C.54.1 Data Fields

- UINT8 status
  Status of the packet passed to the client.
- UINT8 errorCount
  Number of errors.
- UINT8 chainCount
  Mbuf chain count e.g.
- UINT8 rsvdByte0
  Reserved byte to make the descriptor word align.
- UINT16 packetLength
  Packet Length.
- UINT16 rsvdShort0
  Reserved short to make the descriptor a word align.
- IX_MBUF * pRootMbuf
  Pointer to Root mbuf.
- IX_MBUF * pNextMbuf
  Pointer to next mbuf.
- UINT8 * pMbufData
  Pointer to the current mbuf->data.
- UINT32 mbufLength
  Current mbuf length.

C.54.2 Detailed Description

HSS Packetized NpePacket Descriptor Structure.

Definition at line 1176 of file IxNpeA.h.

C.54.3 Field Documentation

UINT8 IxNpeA_NpePacketDescriptor::chainCount

Mbuf chain count e.g.
0 - No mbuf chain
Definition at line 1180 of file IxNpeA.h.

The documentation for this struct was generated from the following file:

- IxNpeA.h

C.55 IxNpeA_RxAtmVc Struct Reference

IxNpeA_RxAtmVcIxNpeA_RxAtmVcIxNpeA_RxAtmVcIxNpeA_RxAtmVcRx Descriptor definition.

C.55.1 Data Fields

- UINT32 rxBitField
  Recieved bit field.
- UINT32 atmCellHeader
  ATM Cell Header.
- UINT32 pSlowPathDesc
  Slow path descriptor.
- UINT16 currMbufLen
  Mbuf Length.
- UINT8 fpIndex
  Shared with timeLimit.
- UINT8 fpMatchIndex
  Fast path Match Index.
- unsigned char * pFpMbufData
  Pointer to fast path mbuf data.
- IX_MBUF * pCurrMbuf
  Pointer to current mbuf.
- unsigned char * pCurrMbufData
  Pointer to current mbuf->data.
- IX_MBUF * pNextMbuf
  Pointer to next mbuf.
- UINT32 totalLen
  Total Length.
- UINT32 aal5CrcResidue
  AAL5 CRC Residue.

C.55.2 Detailed Description

Rx Descriptor definition.

Definition at line 1125 of file IxNpeA.h.

The documentation for this struct was generated from the following file:

- IxNpeA.h

C.56 IxNpeA_TxAtmVc Struct Reference

IxNpeA_TxAtmVcIxNpeA_TxAtmVcIxNpeA_TxAtmVcIxNpeA_TxAtmVcTx Descriptor definition.

C.56.1 Data Fields

- UINT8 port
  Tx Port number.
- UINT8 aalType
  AAL Type.
- UINT16 currMbufLen
  mbuf length
- UINT32 atmCellHeader
  ATM cell header.
- IX_MBUF * pCurrMbuf
  pointer to mbuf
- unsigned char * pCurrMbufData
  Pointer to mbuf->dat.
- IX_MBUF * pNextMbuf
  Pointer to next mbuf.
- UINT32 totalLen
  Total Length.
- UINT32 aal5CrcResidue
  AAL5 CRC Residue.

C.56.2 Detailed Description

Tx Descriptor definition.

Definition at line 1103 of file IxNpeA.h.

The documentation for this struct was generated from the following file:

- IxNpeA.h

C.57 IxNpeDlImageId Struct Reference

IxNpeDlImageIdIxNpeDlImageIdIxNpeDlImageIdIxNpeDlImageIdImage Id to identify each image contained in an image library.

C.57.1 Data Fields

- IxNpeDlNpeId npeId
  NPE ID.
- IxNpeDlFunctionalityId functionalityId
  Build ID indicates functionality of image.
- IxNpeDlMajor major
  Major Release Number.
- IxNpeDlMinor minor
  Minor Revision Number.

C.57.2 Detailed Description

Image Id to identify each image contained in an image library.

Note: **THIS struct HAS BEEN DEPRECATED AND SHOULD NOT BE USED.** It will be removed in a future release. See ixNpeDlNpeInitAndStart for more information.

Definition at line 589 of file IxNpeDl.h.

The documentation for this struct was generated from the following file:

- IxNpeDl.h

C.58 IxNpeMhMessage Struct Reference

IxNpeMhMessageIxNpeMhMessageIxNpeMhMessageIxNpeMhMessageThe 2-word message structure to send to and receive from the NPEs.

C.58.1 Data Fields

- UINT32 data [2]
  the actual data of the message

C.58.2 Detailed Description

The 2-word message structure to send to and receive from the NPEs.

Definition at line 102 of file IxNpeMh.h.

The documentation for this struct was generated from the following file:

- IxNpeMh.h

C.59 IxOamITU610Cell Struct Reference

IxOamITU610CellIxOamITU610CellIxOamITU610CellIxOamITU610CellOAM ITU610 Cell.

C.59.1 Data Fields

- atmCellHeader **header**
- IxOamITU610Payload **payload**

C.59.2 Detailed Description

OAM ITU610 Cell.

Definition at line 233 of file IxAtmCodelet_p.h.

The documentation for this struct was generated from the following file:

- IxAtmCodelet_p.h

C.60 IxOamITU610GenericPayload Struct Reference

IxOamITU610GenericPayloadIxOamITU610GenericPayloadIxOamITU610GenericPayloadIxOamITU610GenericPayloadGeneric payload isn't a real payload but is used for checking which payload type a received OAM cell is.

C.60.1 Data Fields

- UINT8 **oamTypeAndFunction**
- UINT8 **reserved** [IX_OAM_ITU610_GENERIC_PAYLOAD_RESERVED_BYTES_LEN]
- UINT8 **reservedAndCrc10** [IX_OAM_ITU610_RESERVED_AND_CRC10_LEN]

C.60.2 Detailed Description

Generic payload isn't a real payload but is used for checking which payload type a received OAM cell is.

Definition at line 213 of file IxAtmCodelet_p.h.

The documentation for this struct was generated from the following file:

- IxAtmCodelet_p.h

C.61 IxOamITU610LbPayload Struct Reference

IxOamITU610LbPayloadIxOamITU610LbPayloadIxOamITU610LbPayloadIxOamITU610LbPayloadOam cells payload typedefs.

C.61.1 Data Fields

- UINT8 **oamTypeAndFunction**
- UINT8 **loopbackIndication**
- UINT8 **correlationTag** [IX_OAM_ITU610_LB_CORRELATION_TAG_LEN]
- UINT8 **llid** [IX_OAM_ITU610_LOCATION_ID_LEN]
- UINT8 **sourceId** [IX_OAM_ITU610_LOCATION_ID_LEN]
- UINT8 **reserved** [IX_OAM_ITU610_LB_RESERVED_BYTES_LEN]
- UINT8 **reservedAndCrc10** [IX_OAM_ITU610_RESERVED_AND_CRC10_LEN]

C.61.2 Detailed Description

Oam cells payload typedefs.

Definition at line 198 of file IxAtmCodelet_p.h.

The documentation for this struct was generated from the following file:

- IxAtmCodelet_p.h

C.62 IxOamITU610Payload Union Reference

IxOamITU610PayloadIxOamITU610PayloadIxOamITU610PayloadIxOamITU610PayloadOAM ITU610 Payload.

C.62.1 Data Fields

- IxOamITU610LbPayload **lbPayload**
- IxOamITU610GenericPayload **genericPayload**

C.62.2 Detailed Description

OAM ITU610 Payload.

Definition at line 223 of file IxAtmCodelet_p.h.

The documentation for this union was generated from the following file:

- IxAtmCodelet_p.h

C.63 IxPerfProfAccBusPmuResults Struct Reference

IxPerfProfAccBusPmuResultsIxPerfProfAccBusPmuResultsIxPerfProfAccBusPmuResultsIxPerfProfAccBusPmuResultsResults obtained from running the Bus Pmu component. The results are obtained when the get functions is called.

C.63.1 Data Fields

- UINT32 statsToGetLower27Bit [IX_PERFPROF_ACC_BUS_PMU_MAX_PECS]
  Lower 27 Bit of counter value.
- UINT32 statsToGetUpper32Bit [IX_PERFPROF_ACC_BUS_PMU_MAX_PECS]
  Upper 32 Bit of counter value.

C.63.2 Detailed Description

Results obtained from running the Bus Pmu component. The results are obtained when the get functions is called.

Definition at line 155 of file IxPerfProfAcc.h.

The documentation for this struct was generated from the following file:

- IxPerfProfAcc.h

C.64 IxPerfProfAccXcycleResults Struct Reference

IxPerfProfAccXcycleResultsIxPerfProfAccXcycleResultsIxPerfProfAccXcycleResultsIxPerfProfAccXcycleResultsResults obtained from Xcycle run.

C.64.1 Data Fields

- float maxIdlePercentage
  maximum percentage of Idle cycles
- float minIdlePercentage
  minimum percentage of Idle cycles
- float aveIdlePercentage
  average percentage of Idle cycles
- UINT32 totalMeasurements
  total number of measurement made

C.64.2 Detailed Description

Results obtained from Xcycle run.

Definition at line 141 of file IxPerfProfAcc.h.

The documentation for this struct was generated from the following file:

- IxPerfProfAcc.h

C.65 IxPerfProfAccXscalePmuEvtCnt Struct Reference

IxPerfProfAccXscalePmuEvtCntIxPerfProfAccXscalePmuEvtCntIxPerfProfAccXscalePmuEvtCntIxPerfProfAccXscalePmuEvtCntcontains results of a counter

C.65.1 Data Fields

- UINT32 lower32BitsEventCount
  lower 32bits value of the event counter
- UINT32 upper32BitsEventCount
  upper 32bits value of the event counter

C.65.2 Detailed Description

contains results of a counter

Structure contains the results of a counter, which are split into the lower and upper 32 bits of the final count

Definition at line 111 of file IxPerfProfAcc.h.

The documentation for this struct was generated from the following file:

- IxPerfProfAcc.h

C.66 IxPerfProfAccXscalePmuResults Struct Reference

IxPerfProfAccXscalePmuResultsIxPerfProfAccXscalePmuResultsIxPerfProfAccXscalePmuResultsIxPerfProfAccXscalePmuResultscontains results of counters and their overflow

C.66.1 Data Fields

- UINT32 clk_value
  current value of clock counter
- UINT32 clk_samples
  number of clock counter overflows
- UINT32 event1_value
  current value of event 1 counter
- UINT32 event1_samples
  number of event 1 counter overflows

- UINT32 event2_value
  current value of event 2 counter
- UINT32 event2_samples
  number of event 2 counter overflows
- UINT32 event3_value
  current value of event 3 counter
- UINT32 event3_samples
  number of event 3 counter overflows
- UINT32 event4_value
  current value of event 4 counter
- UINT32 event4_samples
  number of event 4 counter overflows

C.66.2 Detailed Description

contains results of counters and their overflow

Structure contains all values of counters and associated overflows. The specific event and clock counters are determined by the user

Definition at line 123 of file IxPerfProfAcc.h.

The documentation for this struct was generated from the following file:

- IxPerfProfAcc.h

C.67 IxPerfProfAccXscalePmuSamplePcProfile Struct Reference

IxPerfProfAccXscalePmuSamplePcProfileIxPerfProfAccXscalePmuSamplePcProfileIxPerfProfAccXscalePmuSamplePcProfileIxPerfProfAccXscalePmuSamplePcProfilecontains summary of samples taken

C.67.1 Data Fields

- UINT32 programCounter
  the program counter value of the sample
- UINT32 freq
  the frequency of the occurence of the sample

C.67.2 Detailed Description

contains summary of samples taken

Structure contains all details of each program counter value - frequency that PC occurs

Definition at line 99 of file IxPerfProfAcc.h.

The documentation for this struct was generated from the following file:

- IxPerfProfAcc.h

C.68 IxQMgrQInlinedReadWriteInfo Struct Reference

IxQMgrQInlinedReadWriteInfoIxQMgrQInlinedReadWriteInfoIxQMgrQInlinedReadWriteInfoIx QMgrQInlinedReadWriteInfoInternal structure to facilitate inlining functions in IxQMgr.h.

C.68.1 Data Fields

- UINT32 qOflowStatBitMask
  overflow status mask
- UINT32 qWriteCount
  queue write count
- volatile UINT32 * qAccRegAddr
  access register
- volatile UINT32 * qUOStatRegAddr
  status register
- volatile UINT32 * qConfigRegAddr
  config register
- UINT32 qEntrySizeInWords
  queue entry size in words
- UINT32 qSizeInEntries
  queue size in entries
- UINT32 qUflowStatBitMask
  underflow status mask
- UINT32 qReadCount
  queue read count

C.68.2 Detailed Description

Internal structure to facilitate inlining functions in IxQMgr.h.

Definition at line 971 of file IxQMgr.h.

The documentation for this struct was generated from the following file:

- IxQMgr.h

C.69 ixUARTDev Struct Reference

ixUARTDevixUARTDevixUARTDevixUARTDevDevice descriptor for the UART.

C.69.1 Data Fields

- UINT8 * addr
  device base address
- ixUARTMode mode
  interrupt, polled or loopback
- int baudRate
  baud rate
- int freq
  UART clock frequency.
- int options
  hardware options
- int fifoSize
  FIFO xmit size.
- ixUARTStats stats
  device statistics

C.69.2 Detailed Description

Device descriptor for the UART.

Definition at line 335 of file IxUART.h.

The documentation for this struct was generated from the following file:

- IxUART.h

C.70 ixUARTStats Struct Reference

ixUARTStatsixUARTStatsixUARTStatsixUARTStatsStatistics for the UART.

C.70.1 Data Fields

- UINT32 **rxCount**
- UINT32 **txCount**
- UINT32 **overrunErr**
- UINT32 **parityErr**
- UINT32 **framingErr**

- UINT32 **breakErr**

C.70.2 Detailed Description

Statistics for the UART.

Definition at line 321 of file IxUART.h.

The documentation for this struct was generated from the following file:

- IxUART.h

C.71 LearningIpFlow Struct Reference

LearningIpFlowLearningIpFlowLearningIpFlowLearningIpFlowStruct to hold IP flow information while learning.

C.71.1 Data Fields

- BOOL inUse
  Learning is taking place.
- UINT32 ipAddress
  Destination IP address.
- UINT8 macAddress [6]
  Corresponding Mac Address.
- unsigned int counter
  Current Packet Count.

C.71.2 Detailed Description

Struct to hold IP flow information while learning.

Definition at line 417 of file IxFpathAccCodeletDefines_p.h.

The documentation for this struct was generated from the following files:

- IxFpathAccCodeletDefines_p.h
- IxFpathAccCodeletMain_p.h

C.72 PacketisedStats Struct Reference

PacketisedStatsPacketisedStatsPacketisedStatsPacketisedStatsingroup IxHssAccCodeletCom

C.72.1 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**

C.72.2 Detailed Description

ingroup IxHssAccCodeletCom

brief Type definition structure for Packetised statistics

Definition at line 112 of file IxHssAccCodeletCom.h.

The documentation for this struct was generated from the following file:

- IxHssAccCodeletCom.h

C.73 USBDevice Struct Reference

USBDeviceUSBDeviceUSBDeviceUSBDeviceUSBDevice.

C.73.1 Data Fields

- UINT32 baseIOAddress
  base I/O device address
- UINT32 interruptLevel
  device IRQ
- UINT32 lastError
  detailed error of last function call
- UINT32 deviceIndex
  USB device index.
- UINT32 flags
  initialization flags
- UINT8 deviceContext [USB_CONTEXT_SIZE]
  used by the driver to identify the device

C.73.2 Detailed Description

USBDevice.

Definition at line 67 of file usbtypes.h.

The documentation for this struct was generated from the following file:

- usbtypes.h

C.74 USBDeviceCounters Struct Reference

USBDeviceCountersUSBDeviceCountersUSBDeviceCountersUSBDeviceCountersfile usbprivatetypes.h

C.74.1 Data Fields

- UINT32 **frames**
- UINT32 **irqCount**
- UINT32 **Rx**
- UINT32 **Tx**
- UINT32 **DRx**
- UINT32 **DTx**

- UINT32 **bytesRx**
- UINT32 **bytesTx**
- UINT32 **setup**

C.74.2 Detailed Description

file usbprivatetypes.h

author Intel Corporation date 30-OCT-2001

This file contains the private USB Driver data types

-- Intel Copyright Notice --

INTEL CONFIDENTIAL

Copyright 2001-2003 Intel Corporation All Rights Reserved.

The source code contained or described herein and all documents related to the source code ("Material") are owned by Intel Corporation or its suppliers or licensors. Title to the Material remains with Intel Corporation or its suppliers and licensors. The Material contains trade secrets and proprietary and confidential information of Intel or its suppliers and licensors. The Material is protected by worldwide copyright and trade secret laws and treaty provisions. No part of the Material may be used, copied, reproduced, modified, published, uploaded, posted, transmitted, distributed, or disclosed in any way without Intel's prior express written permission.

No license under any patent, copyright, trade secret or other intellectual property right is granted to or conferred upon you by disclosure or delivery of the Materials, either expressly, by implication, inducement, estoppel or otherwise. Any license under such intellectual property rights must be express and approved by Intel in writing.

For further details, please see the file README.TXT distributed with this software. -- End Intel Copyright Notice --

Definition at line 46 of file usbprivatetypes.h.

The documentation for this struct was generated from the following file:

- usbprivatetypes.h

C.75 USBSetupPacket Struct Reference

USBSetupPacketUSBSetupPacketUSBSetupPacketUSBSetupPacketStandard USB Setup packet components, see the USB Specification 1.1.

C.75.1 Data Fields

- UCHAR **bmRequestType**
- UCHAR **bRequest**
- UINT16 **wValue**

- UINT16 **wIndex**
- UINT16 **wLength**

C.75.2 Detailed Description

Standard USB Setup packet components, see the USB Specification 1.1.

Definition at line 53 of file usbstd.h.

The documentation for this struct was generated from the following file:
usbstd.h