Intel® IXP400 Software

Programmer's Guide

June 2004

Document Number: 252539-005a

Contents

Figures

Tables

Revision History

Date	Revision	Description
June 2004	005a	Updated branding and document-title information.
December 2003	005	Updated manual for IXP400 Software Version 1.4. Removed API documentation (now in a seperate reference).
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 v1.4 release.

1.1 Versions Supported by this Document

This programmer's guide is intended to be used in conjunction with the Intel® IXP400 Software v1.4 release. For subsequent versions after software release 1.4, 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:

1. Open the file `ixp400_xscale_sw/src/include/IxVersionId.h`.
2. Check the value of IX_VERSION_ID.

1.2 Hardware Supported by this Release

The Intel® IXP400 Software v1.4 release supports the following processors:

- All members of the Intel® IXP42X Product Line of Network Processors
- All variants of the Intel® IXC1100 Control Plane Processor

1.3 Intended Audience

This document describes the software release 1.4 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 and IXC1100 control plane processors.

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 memory buffer management.
Chapters 3 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.

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 through 3 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 and IxQmgr. For that reason, developers also should review those chapters.
3. Review the codelet descriptions in Chapter 21 and their respective source code for those codelets that exercise the features of interest.
4. Refer to the API source code and source code documentation as necessary.

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 Intel® IXP42X product line and IXC1100 control plane processors 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; DSLAMs (Digital Subscriber Line Access Multiplexers) for multi-dwelling units (MxU); wireless access points; industrial control systems; and networked printers.

The IXP42X product line and IXC1100 control plane processors deliver 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 and IXC1100 control plane processors provide 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 and IXC1100 control plane processors have a unique distributed processing architecture that features the performance of the Intel XScale® Core and three Network Processor Engines (NPEs). The combination of the four high-performance processors provides tremendous processing power and enables wire-speed performance at both the LAN and WAN ports. The three NPEs are designed to offload many computationally intensive data plane operations from the Intel XScale core. This provides ample “processing headroom” on the Intel XScale core for developers to add differentiating product features. Software development is made easier by the extensive Intel XScale® technology tools environment that includes compilers, debuggers, operating systems, models, emulators, support services from third party vendors, and fully documented evaluation 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 with the Intel® 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			
SHA-1 / MD-5	X	X			
WEP (ARC4 with WEP-ICV)	X	X	X	X	X
Multi-Channel HDLC	8		8		
Commercial Temperature	X	X	X	X	X
Extended Temperature	X				X

1.6 Related Documents

Users of this document should always refer to the associated **Software Release Notes** for the specific release. Additional 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 Specification Update	273795
Intel® IXP42X Product Line of Network Processors and IXC1100 Control Plane Processor Developer's Manual	252480
Intel® IXP4XX Product Line and IXC1100 Control Plane Processors 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
ARC4	Applied RC4
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
CAS	Channel Associated Signaling
CBC	Cipher Block Chaining
CBR	Constant Bit Rate
CCD	Cryptographic Context Database
CCM	Counter mode encryption with CBC-MAC authentication
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 Intel® IXP42X Product Line
IXA	Internet Exchange Architecture
IXP	Internet Exchange Processor
LAN	Local Area Network
LSB	Least Significant Bit
MAC	Media Access Control
MBS	Maximum Burst Size
MCR	Minimum Cell Rate
MD5	Message Digest 5
MFS	Maximum Frame Size
MIB	Management Information Base
MII	Media-Independent Interface
MLPPP	Multi-Link Point-to-Point Protocol
MPHY	Multi PHY
MSB	Most Significant Bit
MVIP	Multi-Vendor Integration Protocol
MxU	Multi-dwelling Unit
NAK	Not-Acknowledge Packet
NAPT	Network Address Port Translation
NAT	Network Address Translation
NE	Nearly Empty
NF	Nearly Full
NOTE	Not Empty
NOTF	Not Full
NOTNE	Not Nearly Empty
NOTNF	Not Nearly Full

NPE	Network Processing Engine
OC3	Optical Carrier - 3
OF	Overflow
OFB	Output FeedBack
OS	Operating System
PBX	Private Branch Exchange
PCI	Peripheral Component Interface
PCR	Peak Cell Rate
PDU	Protocol Data Unit
PHY	Physical Layer Interface
PID	Packet Identifier
PRE	Preamble Packet
QMgr	Queue Manager
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
WEP	Wired Equivalent Privacy
Xcycle	Idle-Cycle Counter Utilities
xDSL	Any Digital Subscriber Line
XOR	Exclusive OR

Software Architecture Overview 2

2.1 High-Level Overview

The primary design principles of the Intel® IXP400 Software v1.4 architecture are to enable the Intel® IXP42X Product Line and Intel® IXC1100 Control Plane Processor hardware in a manner which allows maximum flexibility. Intel® IXP400 Software v1.4 consists of a collection of software components specific to the IXP42X product line and IXC1100 control plane processors and their reference boards.

This section discusses the software architecture of this product, as shown in "Intel® IXP400 Software v1.4 Architecture Block Diagram" on page 26

The **NPE microcode** consists of one or more loadable and executable NPE instruction files that implement the NPE functionality behind the software release 1.4 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 1.4 library at build time in the form of a single header file. The library includes a NPE downloader component that provides NPE code version selection and downloading services. A variety of NPE microcode images are provided, enabling different combinations of services.

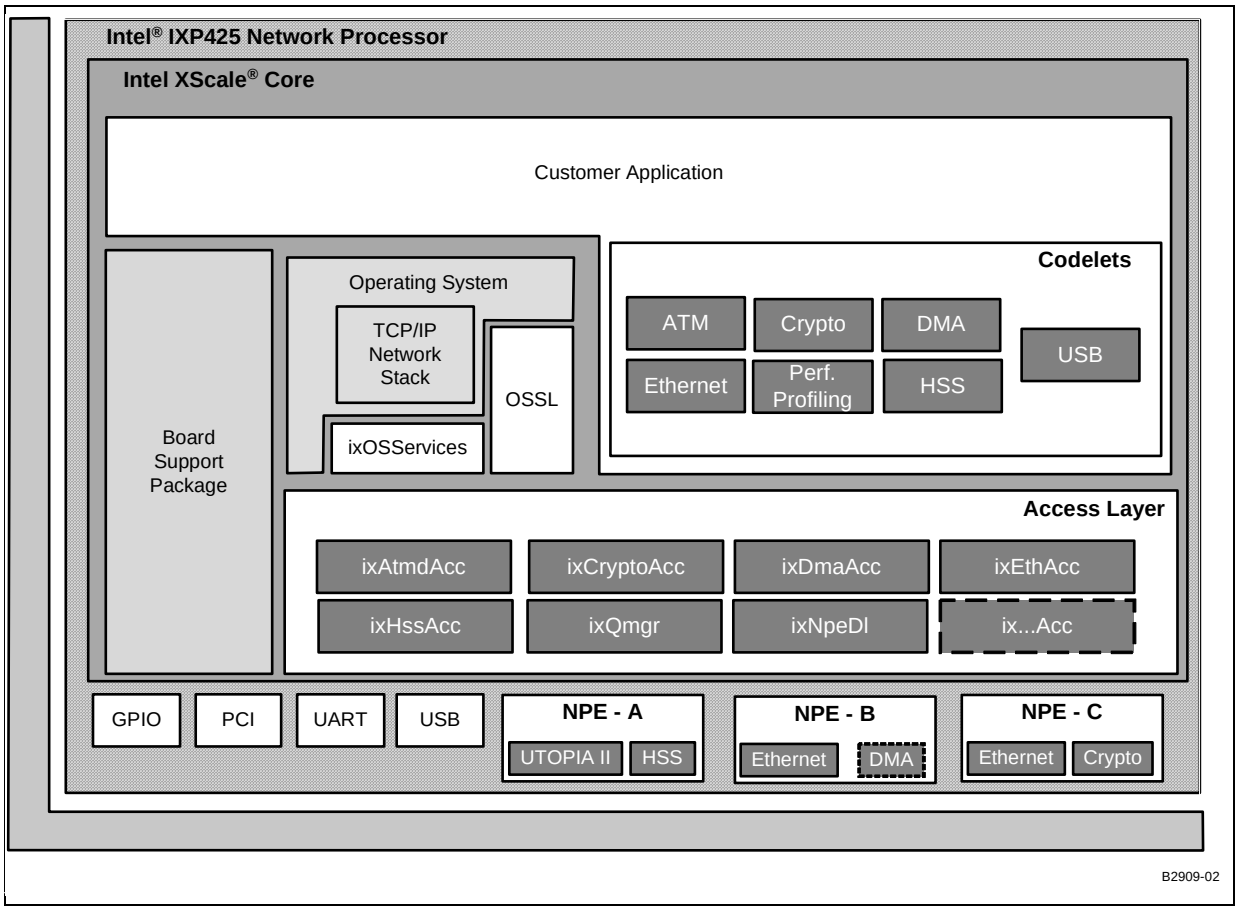
The **Access Layer** provides a software interface which gives customer code access to the underlying capabilities of the IXP42X product line and IXC1100 control plane processors. This layer is made up of a set of software components (Access Layer Components), which clients can use to configure, control and communicate with the hardware. Specifically, most access layer components provide an API interface to specific NPE-hosted hardware capabilities, such as AAL 0 and AAL 5 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: AAL 5 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 v1.4 Architecture Block Diagram



2.2 Deliverable Model

Intel® IXP400 Software v1.4 consists of these elements:

- Intel® IXP400 Software v1.4 library
- Complete documentation and source code for library components
- NPE microcode images
- Example codelets

Note: 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 1.4 library. The NPE images are provided in the form of a single header file incorporated within the software release package, and those NPE images are assumed compatible for that specific release.

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 1.4 supports VxWorks* and the standard Linux* 2.4 kernel. MontaVista* software will provide the support for Linux. Support for other operating systems may be available. For further information, visit the following Internet site:

http://developer.intel.com/design/network/products/npfamily/ixp425.htm

The software release 1.4's software library is OS-independent in that all components are written in ANSI-C with no direct calls 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, and 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 IXC1100 control plane processors 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, Wind River's PLATFORM for Network Equipment* 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, use the following Web URL:

http://www.doxygen.org.

The Intel® IXP400 Software compressed file contains the HTML source code documentation at `ixp400_xscale_sw\doc\index.html`. This output is suitable for *online* browsing. For a *printable* reference, see the Adobe* Portable Document Format (PDF) file, contained in the compressed software-download file.

2.6 Release Directory Structure

The software release 1.4 includes the following directory structure:

```
\---ixp400_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
        |   +---hssAcc
        |   +---perfProfAcc
        |   +---timers
        |   \---usb
        |       +---drivers
        |       \---include
        +---cryptoAcc
        +---dmaAcc
        +---ethAcc
        |   \---include
        +---ethDB
        |   \---include
```

```
+---ethMii
+---featureCtrl
+---flashUpgrade (flash upgrade utility for WindRiver vxWorks)
+---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 1.4 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.

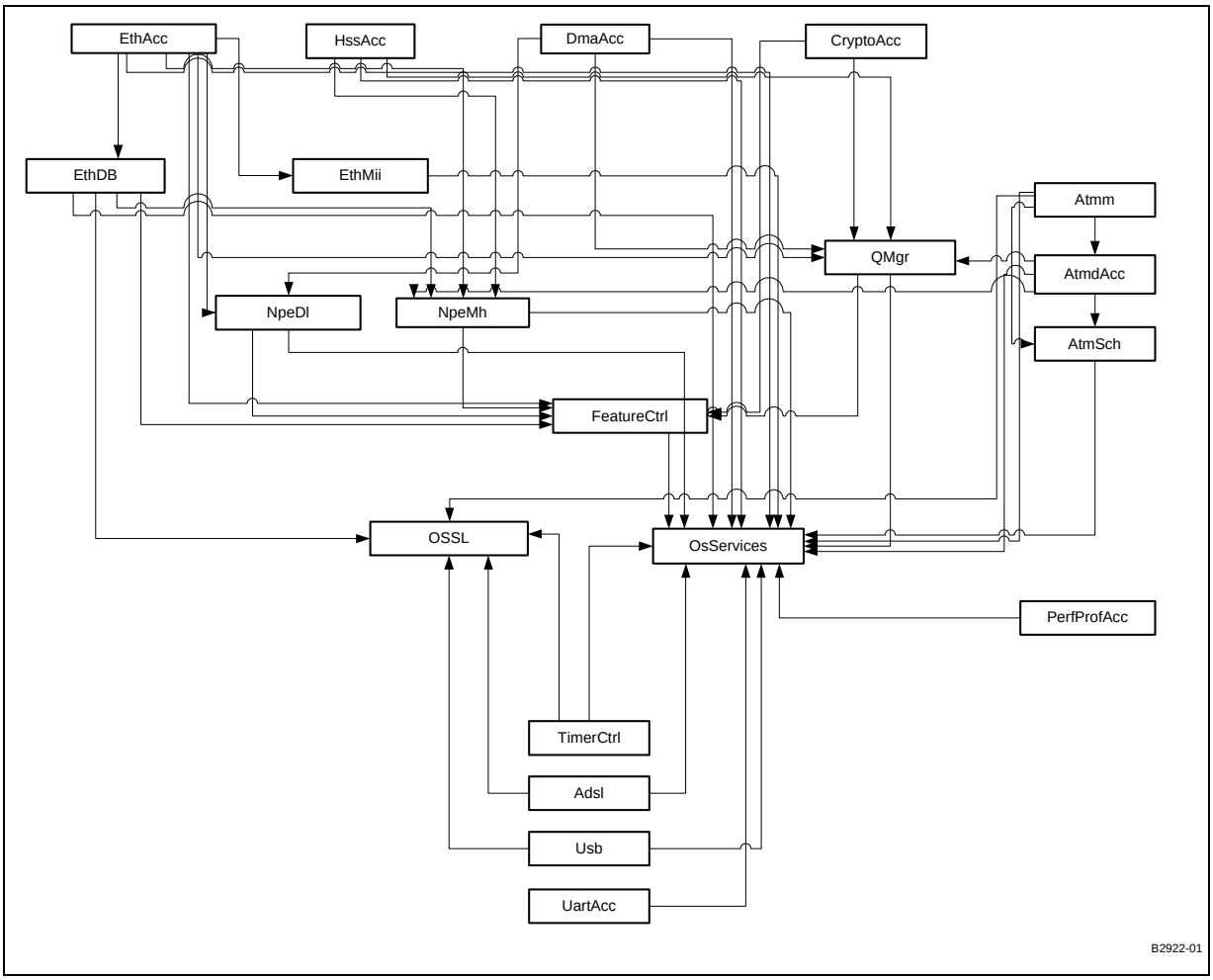
2.9 Statistics and MIBs

The software release 1.4 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 Global Dependency Chart

Figure 2 shows the inter-dependencies for the major APIs discussed in this document.

Figure 2. Global Dependencies



Buffer Management 3

This chapter describes the data buffer system used in Intel® IXP400 Software v1.4, and includes definitions of the IXP400 software internal memory buffers, cache management strategies, and other pertinent information.

3.1 Overview

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

In the IXP400 software, the access-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 of the buffers must be agreed by all components so that they can efficiently flow in the system. The IXP400 software uses two internal buffer formats for all network data:

- mBuf
- raw buffer

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

mBuf

The mBuf format is originally defined in BSD TCP/IP code distribution. VxWorks from Wind River Systems implements mBuf in a specific way — the buffer always being "external." This simplifies the buffer management without sacrificing functionality and flexibility.

Although the mBuf format used is heavily related to VxWorks, it is used in the IXP400 software architecture as one of the two OS-independent buffer formats. The mBuf format is typically used for packet-switched network data. The access-layer components that use this format are IxAtmdAcc, IxEthAcc, IxHssAcc (packetized services), IxCryptoAcc and IxUSB.

Linux utilizes memory structures called skbuffs. The IXP400 software allocates empty IX_MBUF structures (the name for the IXP400 internal mBufs) and sets the data payload pointer to the skbuff payload pointer. A private field inside IX_MBUF is used to save the actual skbuff pointer. In this manner, the OS buffers are not freed directly by the IXP400 software.

The IXP400 software mbuf to skbuff mapping is a “zero-copy” implementation. There is no copy/performance penalty in using Linux skbuffs. Other proprietary buffer schemes could also be implemented with the IXP400 software using the mbuf to skbuff implementation as an example.

Raw Buffers

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 IxHssAcc channelized service uses raw buffers.

3.2 Standard Internal mBuf Definition

The IXP400 software provides a thin abstraction layer to create a standard buffer format to be used throughout the access-layer code, in order to swap buffers of packetised data between the upper software layers and the NPEs.

This abstraction layer is contained in the global header file “IxOsBuffMgt.h” in the IXP400 software. This file contains the definition of the buffer format, and provides the facilities to map this buffer format to native OS buffer implementations. It is important to note that the buffer format used by the IXP400 software is hard-coded in the NPE microcode. It is not possible to change it without rewriting the NPE microcode — a capability not provided in software release 1.4.

The IX_MBUF format is shown below.

Table 2. Internal IX_MBUF Field Format

	0	1	2	3
0	ix_next (IX_MBUF_NEXT_BUFFER_IN_PKT_PTR)			
4	ix_nextPacket (IX_MBUF_NEXT_PKT_IN_CHAIN_PTR)			
8	ix_data (IX_MBUF_MDATA)			
12	ix_len (IX_MBUF_MLEN)			
16	ix_type (IX_MBUF_TYPE)	ix_flags (IX_MBUF_FLAGS)	ix_reserved	
20	ix_pktHeader.ix_pool (IX_MBUF_NET_POOL)			
24	ix_pktHeader.len (IX_MBUF_PKT_LEN)			
28	priv			

A set of macros are provided for the IXP400 software to access each of the fields in the buffer structure. Each macro takes a single parameter – a pointer to the buffer itself. Each macro returns the value stored in the field. More detail on the field, their usage, and the macros are detailed in the table below.

Table 3. IX_MBUF Field Details

Field / MACRO	Purpose	Used by Access-Layer?
IX_MBUF_NEXT_BUFFER_IN_PKT_PTR Parameter type: *IX_MBUF ** Return type: *IX_MBUF ** Description: Returns a 32-bit pointer to the next buffer in the packet	32-bit pointer to the next buffer in a chain (linked list) of buffers. NULL entry marks end of chain.	Yes, where buffer chaining is supported.
IX_MBUF_NEXT_PKT_IN_CHAIN_PTR Parameter type: *IX_MBUF ** Return type: *IX_MBUF ** Description: Returns a 32-bit pointer to the first buffer in the next packet in the packet chain	32-bit pointer to the next packet in a chain (linked list) of packets. NULL entry marks end of chain. Each packet in the chain may consist of a chain of buffers.	No. Packet chaining is not supported by IXP400 Software. **NOTE**: IXP400 Software components may use this field for internal purposes. **This field should be set to NULL when submitted to any access-layer component.**
IX_MBUF_MDATA Parameter type: *IX_MBUF ** Return type: *char ** Description: Returns a pointer to the first byte of the buffer data	32-bit pointer to the data section of a buffer. The data section typically contains the payload of a network buffer.	Yes.
IX_MBUF_MLEN Parameter type: *IX_MBUF ** Return type: *int* Description: Returns the number of octets of valid data in the data section of the buffer	Lengths (octets) of valid data in the data section of the buffer.	Yes.
IX_MBUF_TYPE Parameter type: *IX_MBUF ** Return type: *unsigned char* Description: Returns the type field of the buffer	Buffer type	Yes, by some components.
IX_MBUF_FLAGS Parameter type: *IX_MBUF ** Return type: *unsigned char* Description: Returns the flags field of the buffer	Buffer flags.	Yes, by some components.
Reserved	Reserved field, used to preserve 32-bit word alignment.	No.

Table 3. IX_MBUF Field Details

Field / MACRO	Purpose	Used by Access-Layer?
IX_MBUF_NET_POOL Parameter type: *IX_MBUF ** Return type: *unsigned int* Description: Returns a 32-bit pointer to the parent pool of the buffer	32-bit pointer to the parent pool of the buffer	Yes, by some components.
IX_MBUF_PKT_LEN Parameter type: *IX_MBUF ** Return type: *unsigned int* Description: Returns the length of the packet (typically stored in the first buffer of the packet only)	Total length (octets) of the data sections of all buffers in a chain of buffers (packet). Typically set only in the first buffer in the chain (packet).	Yes, where buffer chaining is supported.
Priv	Reserved field, used to pad the structure to 32 bytes (cache-line size in the Intel XScale core's MMU is 32 bytes).	No.

3.3 Mapping to OS Native Buffer Types

3.3.1 VxWorks* M_BLK Buffer

The buffer format used by the NPEs and IXP400 software was originally designed to correlate with the native buffer structure in the VxWorks OS — the *M_BLK* structure — which is also based on the BSD-standard *mbuf*. For this reason, when compiled for VxWorks, the IX_MBUF buffer format is implemented directly as an M_BLK buffer. Only the subset of fields listed above are used by the IXP400 software, and the macros listed above are used by the IXP400 software to access the correct fields within the M_BLK structure.

The M_BLK structure is defined in the global VxWorks header file "netBufLib.h"

Note that the M_BLK structure contains many fields which are not used by the IXP400 software. These fields are simply ignored and are not modified by the IXP400 software.

M_BLK buffers support 2 levels of buffer chaining:

- *buffer chaining* — Each buffer can be chained together to form a packet. This is achieved using the ***IX_MBUF_NEXT_BUFFER_IN_PKT_PTR*** equivalent field in the M_BLK. This is supported and required by the IXP400 software.
- *packet chaining* — Each packet can consists of a chain of 1 or more buffers. Packets can also be chained together (to form a chain of chains, so to speak). **This is not supported by the** IXP400 software**.** The ***IX_MBUF_NEXT_PKT_IN_CHAIN_PTR*** equivalent field of the M_BLK buffer structure is used for this purpose. Most IXP400 software components will ignore this field. Some however, such as IxEthAcc, will reuse this field internally for a different purpose.

The following table shows the field mapping between the IX_MBUF and the M_BLK buffer structures:

Table 4. IX_MBUF to M_BLK Mapping

IX_MBUF	M_BLK
IX_MBUF_NEXT_BUFFER_IN_PKT_PTR	mBlkHdr.mNext
IX_MBUF_NEXT_PKT_IN_CHAIN_PTR	mBlkHdr.mNextPkt
IX_MBUF_MDATA	mBlkHdr.mData
IX_MBUF_MLEN	mBlkHdr.mLen
IX_MBUF_TYPE	mBlkHdr.mType
IX_MBUF_FLAGS	mBlkHdr.mFlags
ix_reserved	mBlkHdr.reserved
IX_MBUF_NET_POOL	mBlkPktHdr.rcvif
IX_MBUF_PKT_LEN	mBlkPktHdr.len
priv	pClBlk

3.3.2 Linux* skbuff Buffer

The buffer format native to the Linux OS is the “skbuff” buffer structure. This is significantly different from the IX_MBUF buffer format used by the IXP400 software. A simple utility function to map the essential fields of an skbuff to an IX_MBUF structure is provided by *mbuf_swap_skb()*, located in IxOsBuffMgt.h.

It works on the following principles:

- Each IX_MBUF is mapped to an skbuff (1:1 mapping)
- The **priv** field of the IX_MBUF structure is used to store a pointer to the corresponding skbuff
- The **IX_MBUF_MDATA** pointer field of the IX_MBUF structure will be set to point to the **data** field of the corresponding skbuff.
- The **IX_MBUF_MLEN** and **IX_MBUF_PKT_LEN** fields of the IX_MBUF structure will be set to the length of the skbuff data section (the **len** field in the skbuff structure).

The prototype for this function is listed below.

Table 5. Prototype for IX_MBUF/skbuff Conversion Function

```
static inline struct sk_buff *
mbuf_swap_skb(IX_MBUF *mbuf, struct sk_buff *skb);

Parameters:
mbuf: valid pointer to a buffer of type IX_MBUF.
The following fields of this buffer structure will get
overwritten:
 • priv
 • ix_len
 • ix_pktHeader.len
 • ix_data

skb: NULL pointer, or valid pointer to a buffer of type
skbuff.
If NULL, the function will assume that an skbuff has already
been attached to the IX_MBUF so it will simply return mbuf-
>priv.

Otherwise, the function will assume that skb is a pointer to a
valid skbuff which should be attached to the mbuf.  In this
case, it will make the following assignments:
 • mbuf->priv = skb
 • mbuf->ix_data = skb->data
 • mbuf->ix_len = skb->len
 • mbuf->ix_pktHeader.len = skb->len
```

The suggested usage model of this function is:

- Allocate a pool of IX_MBUF buffer headers. Do not allocate data sections for these buffers.
- When passing a buffer from higher-level software (e.g. OS network stack) to the IXP400 software, attach the skbuff to an IX_MBUF using the *mbuf_swap_skb()* function and submit the IX_MBUF.
- When receiving an IX_MBUF passed from IXP400 software to higher-level software, use the *mbuf_swap_skb()* function to retrieve a pointer to the skbuff that was attached to the IX_MBUF, and use that skbuff with the OS network stack to process the data.

The Linux NPE Ethernet driver (“ixp425_eth.c”), which is included in the IXP400 software distribution in a patch file named “ixp425LinuxDrivers.patch”, contains an example of this suggested usage model.

3.4 Caching Strategy

The general strategy is the IXP400 software architecture is that the software (include Intel XScale® Core-based code and NPE microcode) only concerns itself with the parts of a buffer which it modifies. For all other parts of the buffer, the user (higher-level software) is entirely responsible.

mBuf buffers typically contain a header section and a data section. The header section contains fields which may be used and modified by the IXP400 software (including NPE microcode). Examples of such fields are:

- pointer to the data section of the mbuf
- length of the data section of the mbuf
- pointer to the next mbuf in a chain of mbufs
- buffer type field
- buffer flags field

As a general rule, IXP400 software concerns itself only with mbuf headers, and assumes that the user (i.e. higher-level software) will take care of the data section of mbufs.

The use of cached memory for mbufs is strongly encouraged, as it will result in a performance gain as the buffer data is accessed many times up through the higher layers of the Operating System's network stack. However, use of cached memory has some implications which need to be considered when used for buffers passed through the IXP400 software Access-Layer.

The code that executes on Intel XScale core accesses the buffer memory via the cache in the Intel XScale core MMU. However, the NPEs bypass the cache and access this memory directly. This has different implications for buffers transmitted from the Intel XScale core to NPE (Tx path), and for buffers received from NPE to the Intel XScale core (Rx path).

3.4.1 Tx Path

If a buffer in cached memory has been altered by the Intel XScale core code, the change will exist in the cached copy of the mbuf, but may not be written to memory yet. In order to ensure that the memory is up-to-date, the portion of cache containing the altered data must be *flushed*.

The cache flushing strategy uses the following general guidelines:

- **The "user" is responsible for flushing the data section of the mbuf**. Only those portions of the data section which have been altered by the the Intel XScale core code need to be flushed. This must be done **before** submitting an mbuf to the IXP400 software for transmission via the component APIs (e.g. ixEthAccPortTxFrameSubmit().
- **The** IXP400 software **is responsible for flushing the header section of the mbuf.** This must be done before submitting an mbuf to the NPE. Communication to the NPEs is generally performed by Access-Layer components sending mBuf headers to IxQMgr queues.

Since flushing portions of the cache is an expensive operation in terms of CPU cycles, it is not advisable to simply flush both the header **and** data sections of each mBuf. To minimize the performance impact of cache-flushing, the IXP400 software only flushes that which it modifies (the mbuf header) and leaves the flushing of the data section as the responsibility of the user. The user can minimize the performance impact by flushing only what it needs to.

Tx Cache Flushing Example

In the case of an Ethernet bridging system, only the user knows that it is not necessary to flush any part of the packet payload. In a routing environment, the stack knows that only the beginning of the mbuf may need to be flushed (for example, if the TTL field of the IP header is changed). Additionally, with the VxWorks OS, mbufs can be from cached memory or uncached memory. Only the user knows which ones need to be flushed or invalidated and which ones do not.

When the NPE has transmitted the data in a buffer, it will return the buffer back to the Intel XScale core. In most cases, the cache copy is still valid because the NPE will not modify the contents of the buffer on transmission. Therefore, as a general rule, IXP400 Software does not invalidate the cached copy of mbufs used for transmission after they are returned by the NPE.

3.4.2 Rx Path

If a buffer has been altered by an NPE, the change will exist in memory but the copy of the buffer in the Intel XScale core cache may not be up-to-date. We need to ensure that the cached copy is up-to-date by invalidating the portion of cache which contains the copy of the altered buffer data.

The strategy for dealing with data receive by the NPEs uses the following general guidelines:

- The “user” is responsible for invalidating the data section of the mbuf. Again, only the user knows which portions of the data section it needs to access. Some IXP400 software require the user to submit free mbufs that are to be used to hold received data (for example, ixEthAccPortRxFreeReplenish()). It is strongly recommended that the cache location holding the data portion of the free mbufs be invalidated **before** submitting them via the API.
- The IXP400 software is responsible for flushing and invalidating the header section of the mbuf. The IXP400 software may modify the header of the mbuf before passing it to the NPE, hence the need to flush and then invalidate the header section of the mbuf. This should be done before submitting an mbuf to the NPE for reception (via IxQMgr queues).

Note: In some cases, the Access-Layer will flush the header section of the mbuf before submitting the mbuf to the NPE and will invalidate the header section after receiving it back from the NPE with data. This approach is also acceptable - however, the approach listed above is considered more efficient and more robust.

As in the flushing operations listed in the previous section, invalidating portions of the cache is an expensive operation in terms of CPU cycles. To minimize the performance impact of cache-invalidating, the IXP400 software only invalidates that which it modifies (the mbuf header) and leaves the invalidating of the data section as the responsibility of the user. The user can minimize the performance impact by invalidating only what is necessary. When recycling mbufs, only the user knows what was the previous use of the mbuf and the parts of payload which may require to be invalidated.

3.4.3 Caching Strategy Summary

Before the NPE reads the contents of a buffer in cached memory, ensure that the memory is up-to-date by flushing cached copies of any parts of the buffer memory modified by the Intel XScale core.

After the NPE modifies the contents of a buffer in cached memory, ensure that the Intel XScale core’s MMU cache is up-to-date by invalidating cached copies of any parts of the buffer memory which the Intel XScale core will need to read.

OS-independent macros are provided for both flushing (`IX_ACC_DATA_CACHE_FLUSH`) and invalidating (`IX_ACC_DATA_CACHE_INVALIDATE`). For more information, refer to the header file `ixp400_xscale_sw\src\include\IxOsCacheMMU.h`.

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

This chapter describes the Intel® IXP400 Software v1.4'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 1.4 supports AAL 5, AAL 0, and OAM. This component provides an abstraction to the IXP42X product line and IXC1100 control plane processors' 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 1.4, 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 and IXC1100 control plane processors' 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.

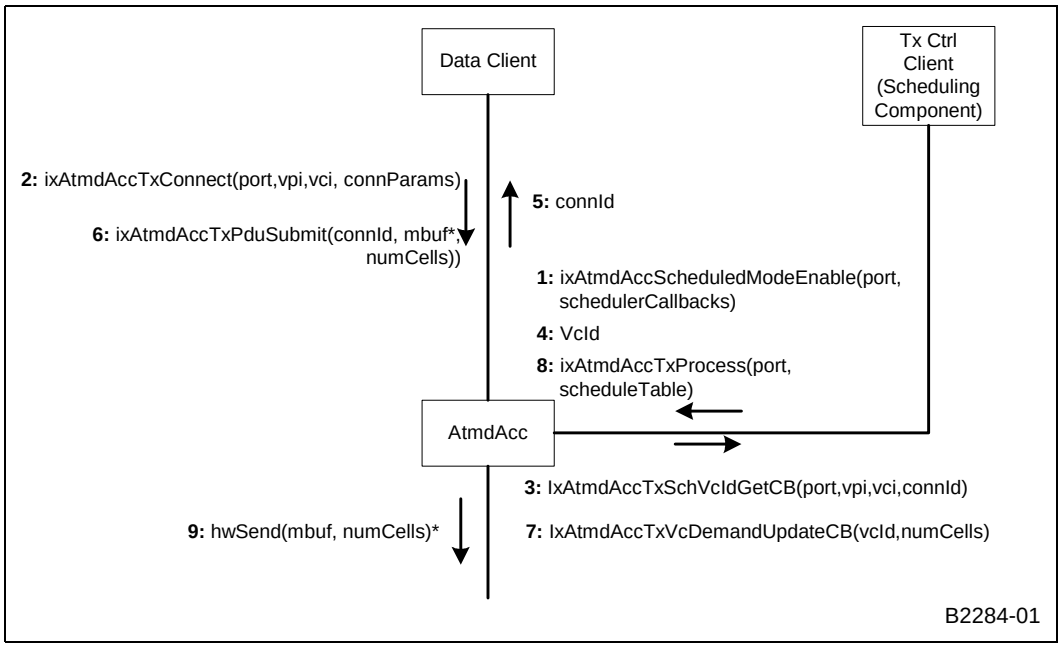
In the case of OAM, a PDU containing OAM cells for any port, VPI, or VCI must be submitted for transmission on the dedicated OAM-VC for that port. This is true regardless of whether an AAL-5/AAL-0-48/AAL-0-52 transmit service connection exists for the given VPI or VCI. The dedicated OAM-VC will be scheduled 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 3 shows VC connection and buffer transmission for a scheduled port.

Figure 3. 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.

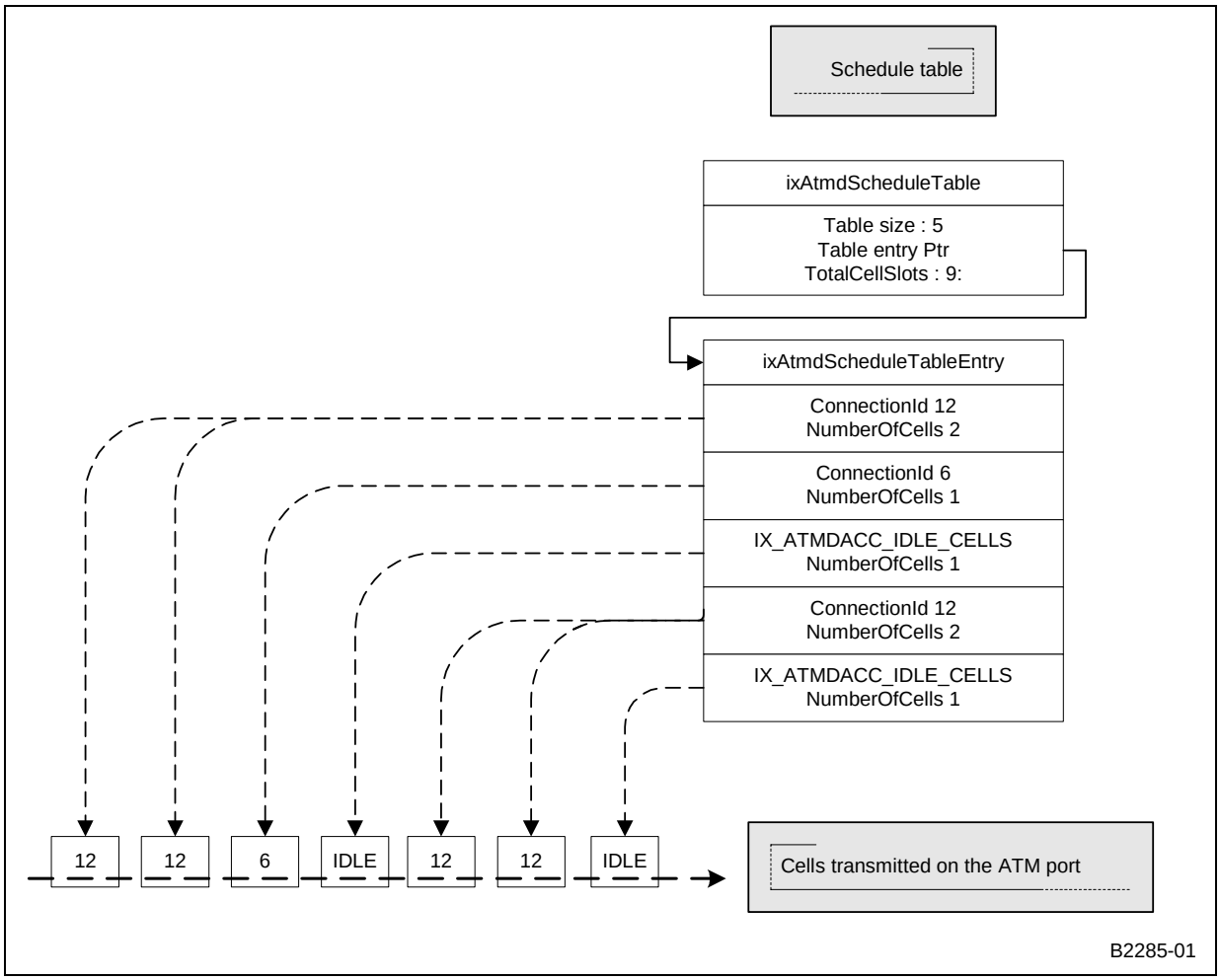
4.3.1.1 Schedule Table Description

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

Figure 4. 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

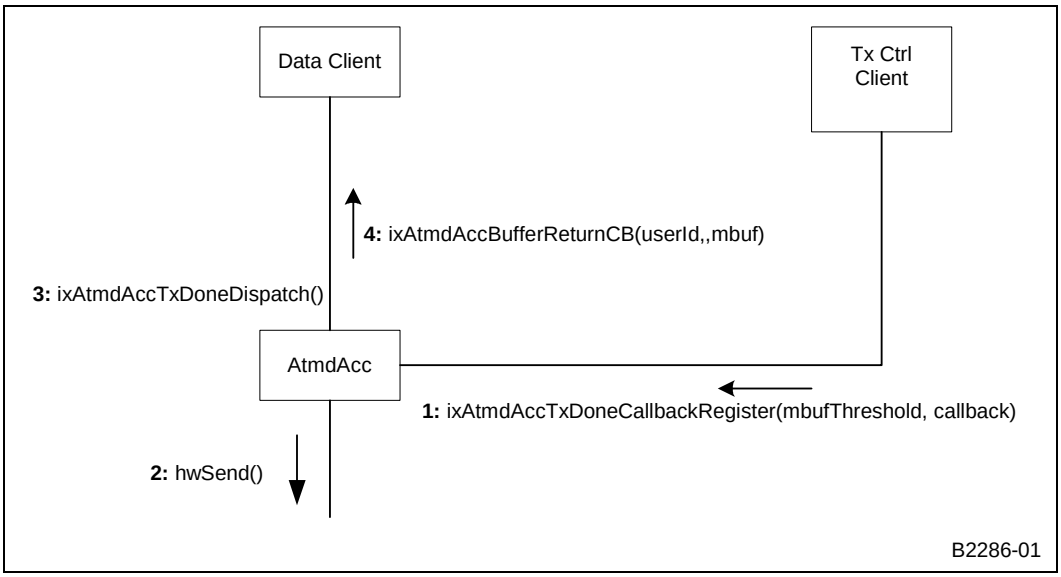
When buffers have been sent on a port, they are placed in a single, transmit-complete stream, which is common to all ports. IxAtmdAcc does not autonomously process this stream — the client, instead, deciding when and how many buffers will be processed.

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 5.

Figure 5. Tx Done Recycling — Using a Threshold Level



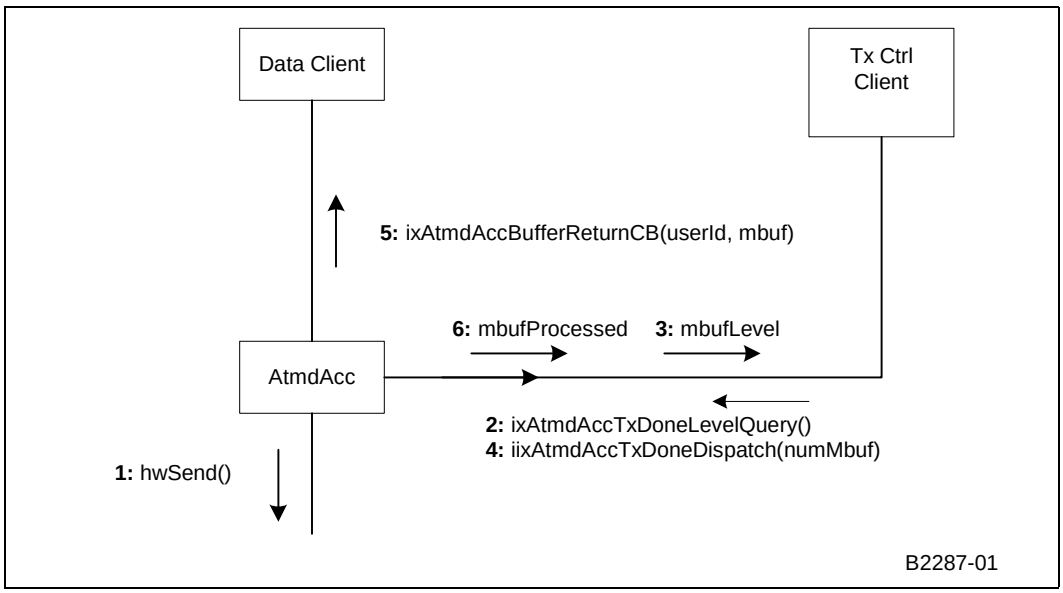
1. The control client wants to use the threshold services to process the transmitted buffers. The ixAtmdAccTxDoneCallbackRegister() function is called to set a buffer threshold level and register a callback. IxAtmdAcc provides the function ixAtmdAccTxDoneDispatch() to be used by the control client. This 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 6.

Figure 6. 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.

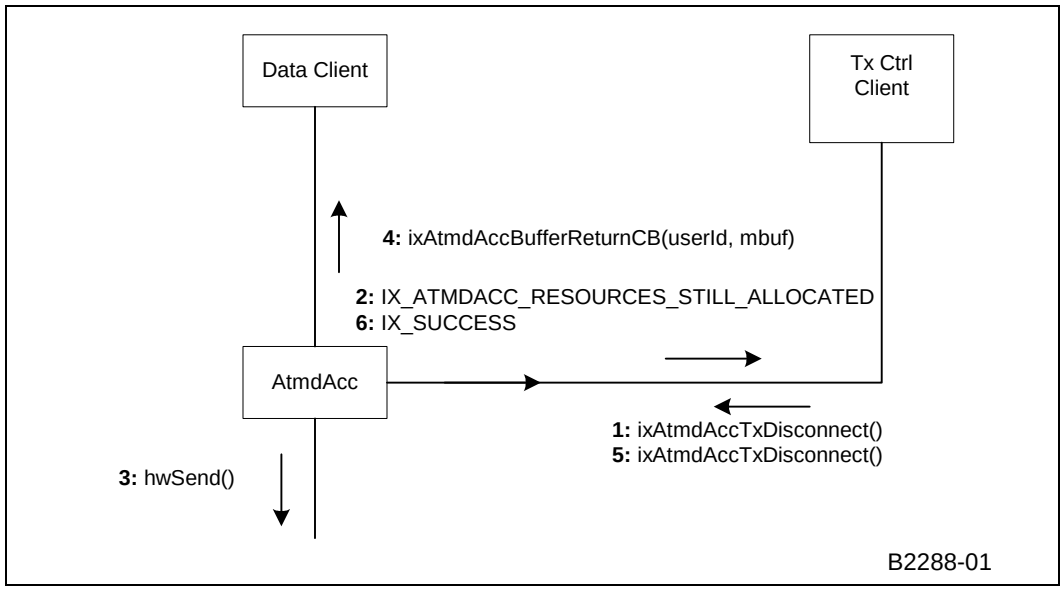
   This TxDoneCallback function is likely to free or recycle the chained ix_mbufs.

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 7. 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 7. 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

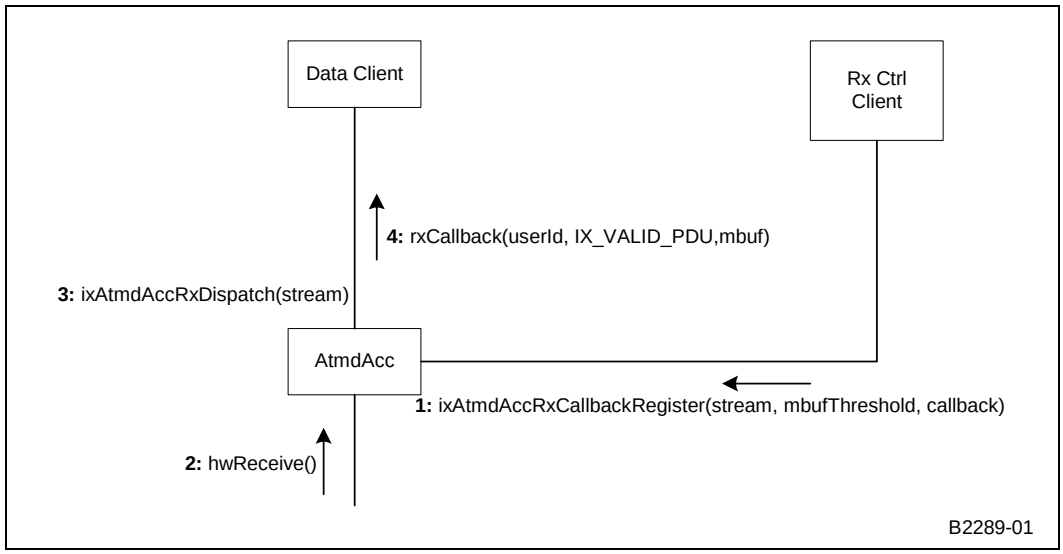
When buffers have been received on a port, they are placed in one of two Rx streams common to the VCs sharing this resource as decided by the client when establishing a connection. IxAtmdAcc does not autonomously process this stream, but instead the client decides when and how many buffers will be processed.

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 8.

Figure 8. 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.

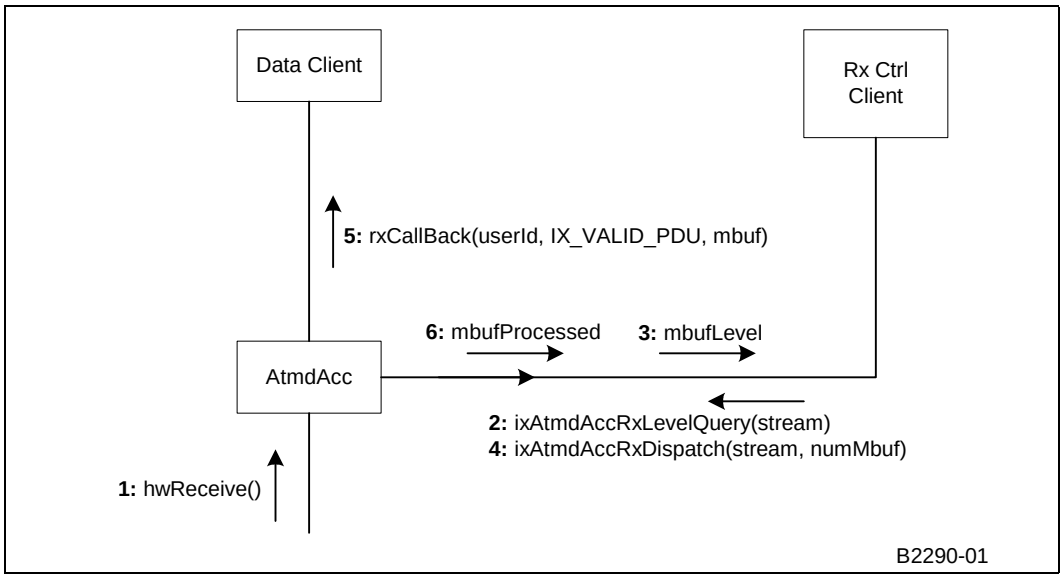
4. Based on connId, ixAtmdAccRxDispatch identified the data client to whom this buffer belongs. The corresponding data client's RxCallback function — as registered during a RxVcConnect — is invoked with the first ix_mbuf of a PDU.

   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 9.

Figure 9. RX Using a Polling Mechanism



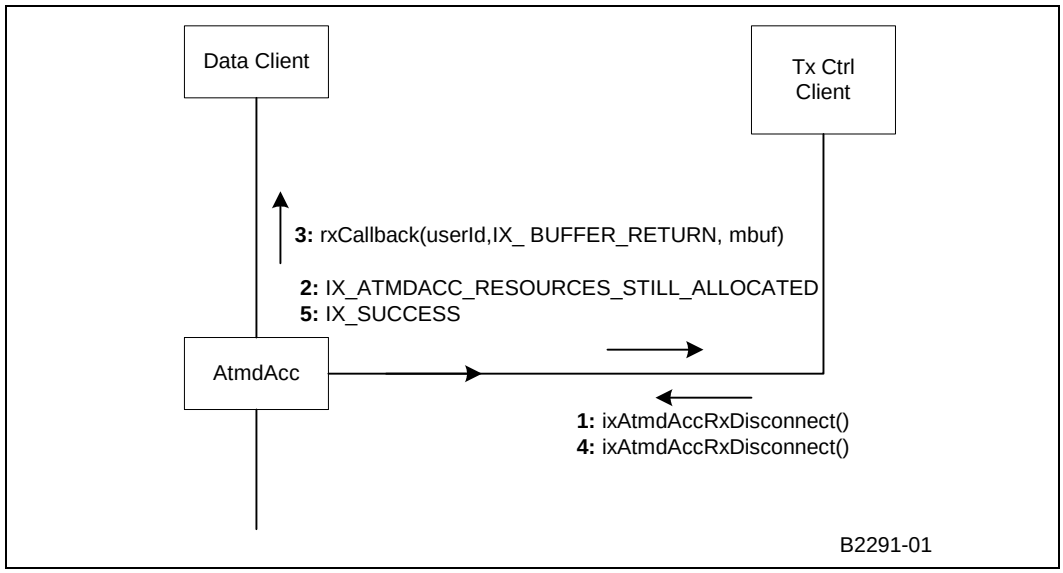
1. Cells are now received over the UTOPIA interface and a complete PDU is now available.
2,3. The control client does not want to use the threshold services. Therefore, the client can optionally query the current number of PDUs already received in one of the receive streams, using the ixAtmdAccRxLevelQuery() 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 10.

Figure 10. 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. The component addressing space for physical memory is limited to 28 bits. Therefore mBuf headers should be located in the first 256 Mbytes of physical memory.

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 6. These fields will not be changed during the Tx process.

Table 6. 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 7. They are set by the client which wants to provide available buffers to IxAtmdAcc Rx service.

Table 7. 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 8.

Table 8. 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 8. 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 9 shows the different possible errors and the way to resolve them.

Table 9. 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 v1.4'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 and IXC1100 control plane processors.

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 and IXC1100 control plane processors.

  In software release 1.4, 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 and IXC1100 control plane processors' 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 and IXC1100 control plane processors 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 — that the VC can be considered usable on this port.”

IxAtmm is not responsible for signaling and obtaining admission from the 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 11. Services Provided by Ixatmm

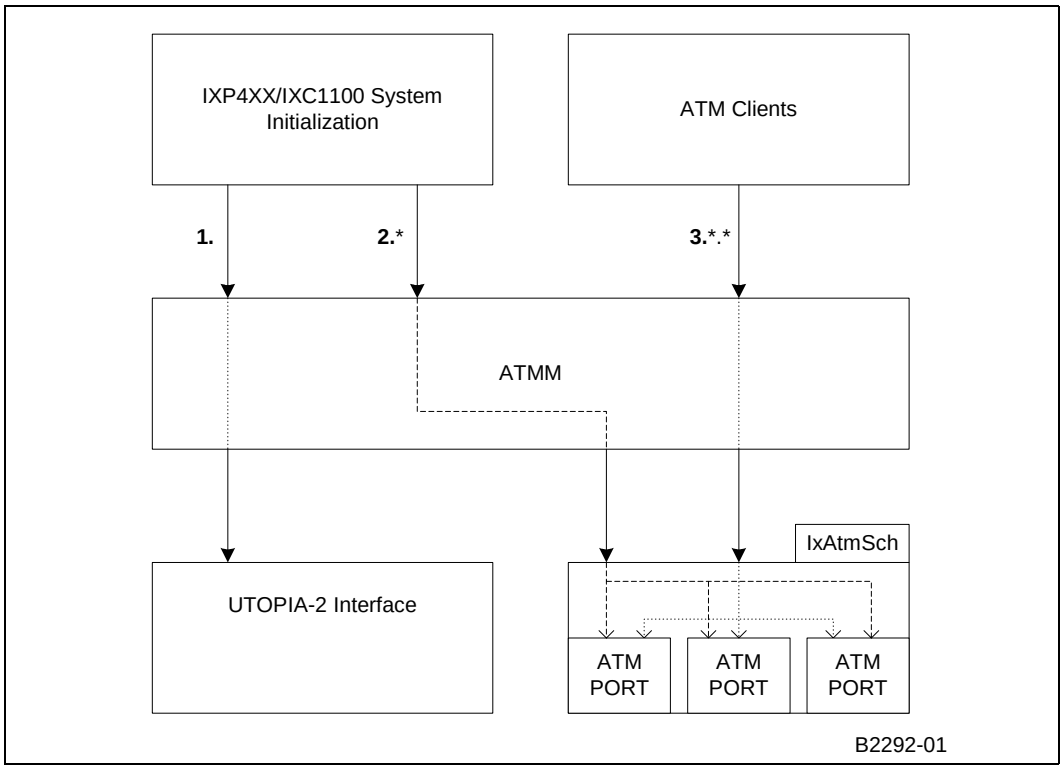


Figure 11 shows the main services provided by the IxAtmm component. In this diagram, the three services outlined are:

- IXP42X product line and IXC1100 control plane processors system-initialization routine will invoke an IxAtmm interface function to initialize the UTOPIA Level-2 device for all active ATM ports in the system. This function call is only performed once, encompassing the hardware configuration of all ports in a single call to the interface.
- 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 and IXC1100 control plane processors' 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 and IXC1100 control plane processors' 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.

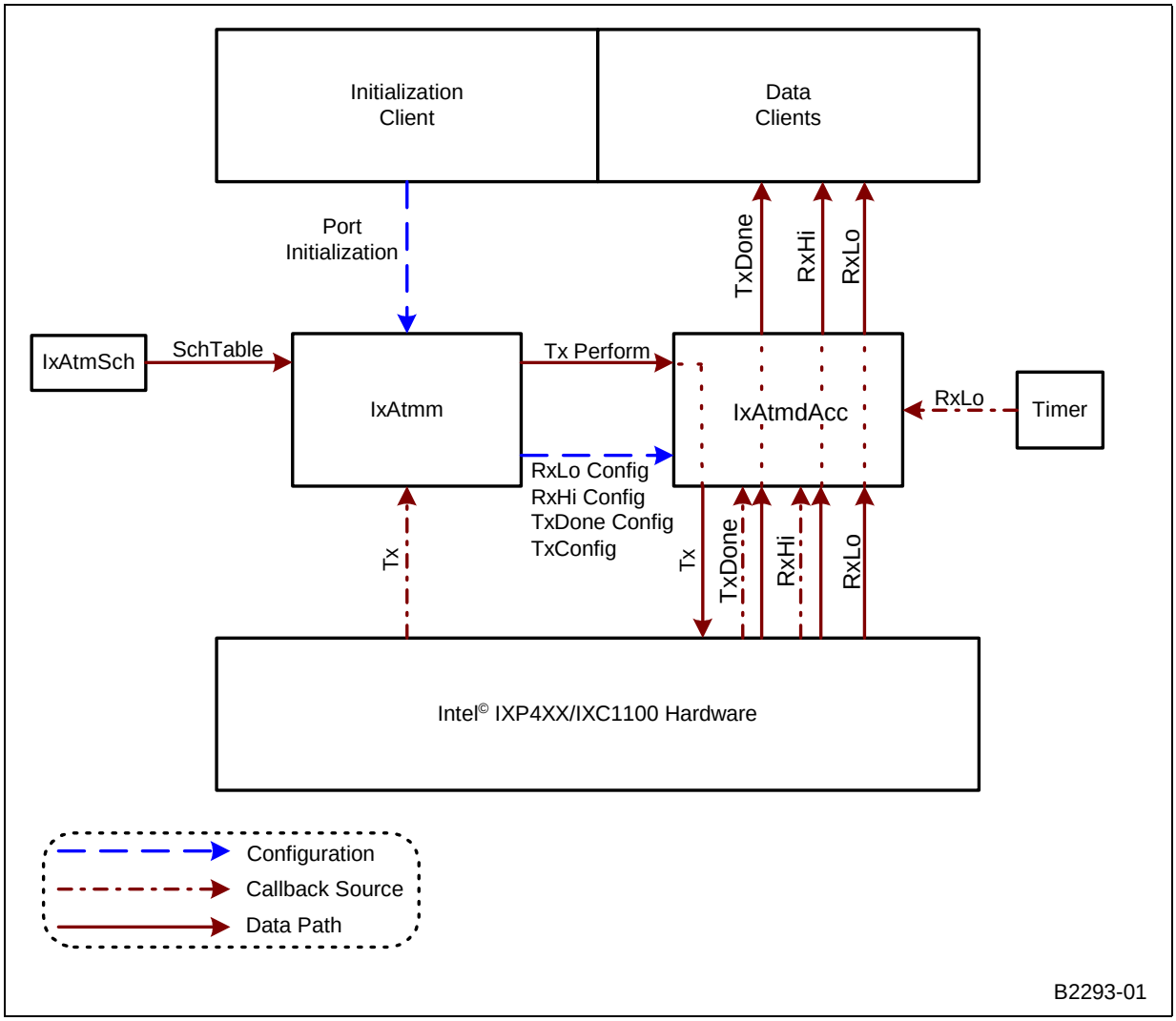
IxAtmm will configure IxAtmdAcc for the following traffic events:

- **Transmit Required** — The IXP42X product line and IXC1100 control plane processors' 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 and IXC1100 control plane processors' hardware to return the buffer to the relevant client. IxAtmm will configure the IXP42X product line and IXC1100 control plane processors 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 and IXC1100 control plane processors 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 and IXC1100 control plane processors 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 12. Configuration of Traffic Control Mechanism

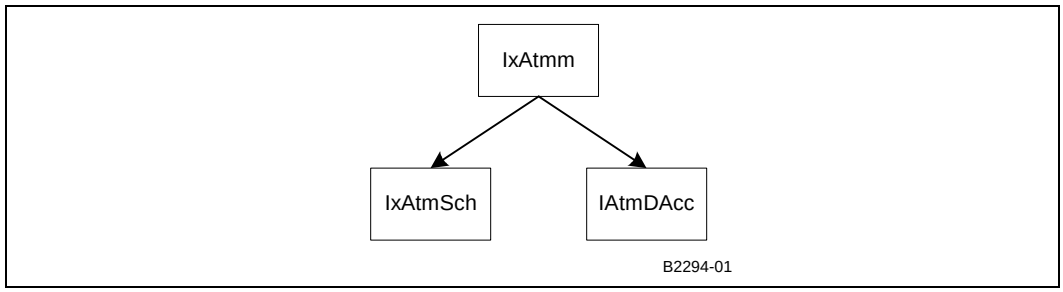


Only transmit traffic — which has already been queued by the client with IxAtmdAcc when the request for more traffic is made — will be scheduled and sent to the hardware. (That is, no callback to the data client will be made in the context of the transmit processing.) IxAtmdAcc makes IxAtmSch aware of 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 13. Component Dependencies of IxAtmm



IxAtmm configures the IXP42X product line and IXC1100 control plane processors' 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 and IXC1100 control plane processors' 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 and IXC1100 control plane 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 v1.4's "ATM Transmit Scheduler" (IxAtmSch) access-layer component.

6.1 Overview

IxAtmSch is an "example" software release 1.4 component, an ATM scheduler component supporting ATM transmit services on IXP42X product line and IXC1100 control plane 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 10 on page 70 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 and IXC1100 control plane processors' 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 and IXC1100 control plane processors' ATM transmit. It provides two basic services for managing transmission on ATM ports:

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

IxAtmSch implements a fully operational ATM traffic scheduler for use in the IXP42X product line and IXC1100 control plane processors' 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 and IXC1100 control plane processors' 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 and IXC1100 control plane processors' resource requirements are for a maximum of eight ports and a total of 32 VCs across all ports. This may increase in the future.

Table 10 shows the ATM service categories that are supported in the current scheduler model.

Table 10. 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

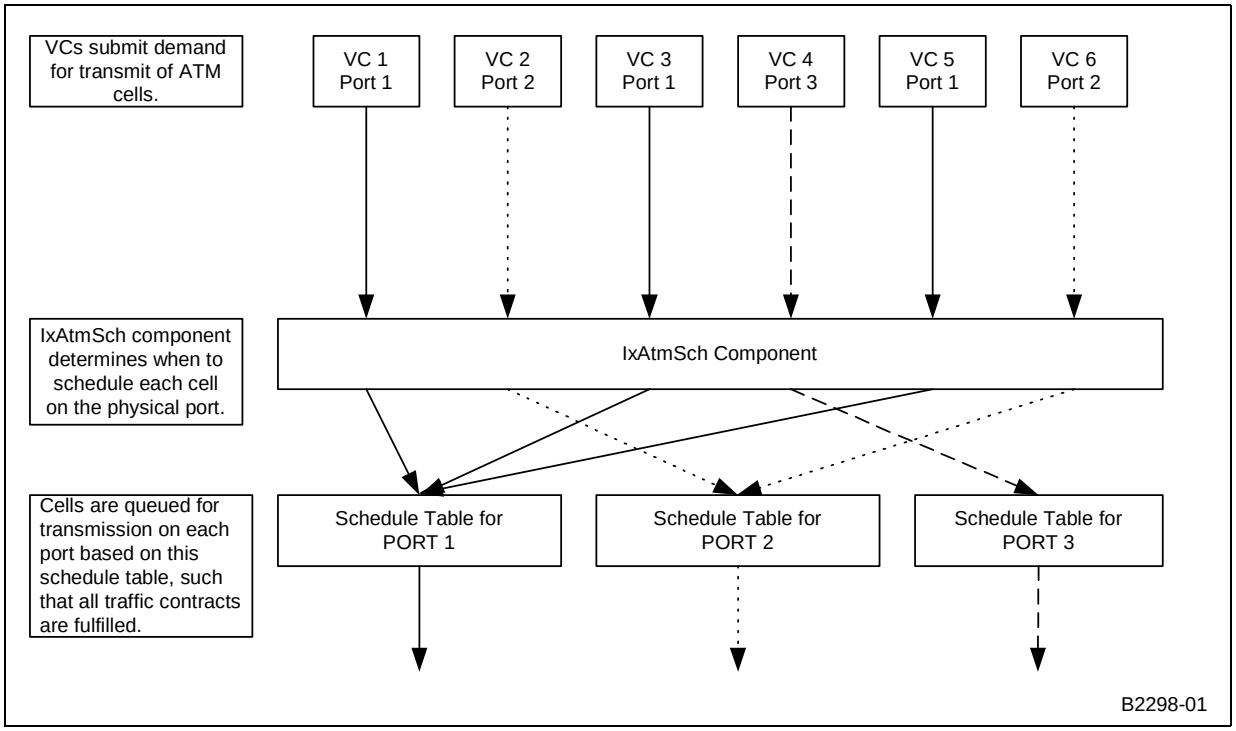
IxAtmSch works on a first-come-first-served basis. For example, if three existing CBR VCs on the ATM port each use one-fourth of the port’s capacity (PCR = R/4), the fourth CBR VCC asking for 1/3 of the port capacity (PCR = R/3) will be rejected. IxAtmSch issues a globally unique VCC ID for each accepted 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 and IXC1100 control plane processors device to the IxAtmSch component.

6.4 Scheduling and Traffic Shaping

Figure 14. 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.

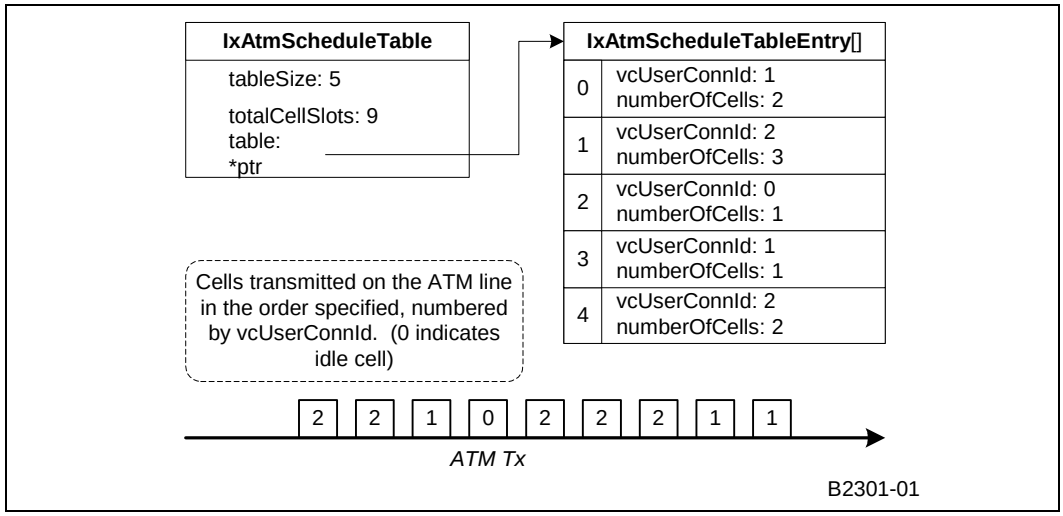
When the client calls the schedule function, the scheduler assumes that all previously scheduled cells on this port have been transmitted and that it may overwrite the previous schedule table with the new table. The client, therefore, must not be dependent on the integrity of the previous table when 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 15 shows how this table is translated into an ordered sequence of cells transmitted to the ATM port.

Figure 15. 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 and IXC1100 control plane processors 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 11.

Table 11. 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 the IXP42X product line and IXC1100 control plane processors.

6.8.1 Latency

The transmit latency introduced by the IxAtmSch component into the overall transmit path of the IXP42X product line and IXC1100 control plane 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 v1.4's "Security API" IxCryptoAcc access-layer component.

The Security Hardware Accelerator access component (IxCryptoAcc) provides support for authentication and encryption/decryption services needed in IPSec applications. IPSec clients can offload the task of encryption/decryption from the Intel XScale core by using the crypto coprocessor, depending on the cryptographic algorithm used. IPSec clients can also offload the task of authentication by using the hashing coprocessor. IxCryptoAcc also provides cryptography and encryption services for Wireless Equivalent Privacy (WEP) applications.

7.1 What's New

The following changes and enhancements were made to this component in software release 1.4:

- ARC4 (Applied RC4) algorithm support and WEP ICV generation and verification have been added, with acceleration services provided by either NPE A or the Intel XScale core.
- Single-pass AES-CCM encryption and authentication support has been added to specifically support the 802.11i specification. This support includes the ability to replace the currently registered cipher key without re-cycling the crypto context registration.
- Four new functions have been provided: IxCryptoAccConfig(), IxCryptoAccXscaleWepPerform(), IxCryptoAccNpeWepPerform(), and IxCryptoAccCipherKeyUpdate().

7.2 Overview

The IxCryptoAcc component provides the following major capabilities:

- 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, 56-bit + 8-bit parity each = 192 bits total)

  - AES (128-bit block cipher size; key sizes: 128-, 192-, 256-bit)
  - ARC4 (8-bit block cipher size, 128-bit key)
- Mode of operation for encryption and decryption:
  - ECB
  - CBC
  - CTR (for AES algorithm *only*)
  - Single-Pass AES-CCM encryption and security for 802.11i.
- 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)
  - WEP ICV generation and verification using the 802.11 WEP standard 32-bit CRC polynomial.
- 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.)

7.3 IxCryptoAcc API Architecture

The IxCryptoAcc API is an access layer component that provides cryptographic services to a client application. This section describes the overall architecture of the API. Subsequent sections describe the component parts of the API in more detail and describe usage models for the IxCrypto API.

7.3.1 IxCryptoAcc Interfaces

IxCryptoAcc is the API which provides IPSec and WEP features in software release 1.4. This API contains functions that can generally be grouped into two distinct “services.” The API utilizes a number of other access-layer components, as well as hardware-based acceleration functionality available on the NPEs and Intel XScale core. Figure 16 on page 80 shows the high-level architecture of IxCryptoAcc.

The IPSec and WEP **clients** are application-level code executing on the Intel XScale core that utilize the services provided by IxCryptoAcc. In this software release, the IxCryptoAccCodelet is provided as an example of client software.

The **Intel XScale core WEP Engine** is a software-based “engine” for performing ARC4 and WEP ICV calculations used by WEP clients. While this differs from the model of NPE-based hardware acceleration typically found in the **IXP400 software**, it provides additionally design flexibility for products that require NPE A to perform non-crypto operations.

IxQMgr is another access-layer component that interfaces to the hardware-based AHB Queue Manager (AQM). The AQM is SRAM memory used to store pointers to data in SDRAM memory, which is accessible by both the Intel XScale core and the NPEs. These items are the mechanism by which data is transferred between IxCryptoAcc and the NPEs. Separate hardware queues are used for both IPSec and WEP services.

The **NPEs** provide hardware acceleration for IxCryptoAcc. Specifically, AES, DES, and hashing acceleration can be provided by NPE C. NPE A offers ARC4 and WEP ICV CRC acceleration.

7.3.2 Basic API Flow

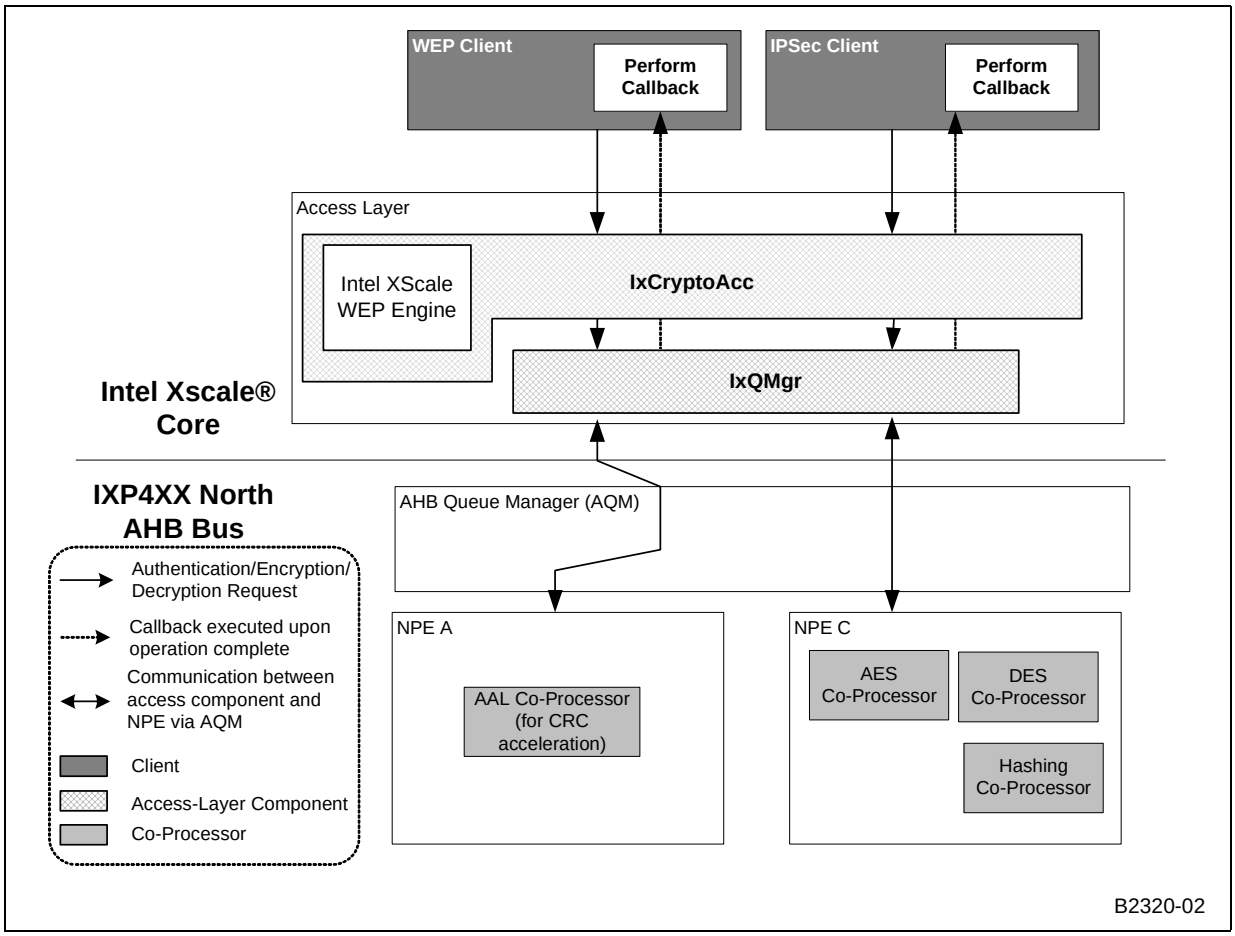
This section describes a high-level flow of the IxCryptoAcc API. A more detailed example of API usage is provided in a subsequent section.

The flow of the API is similar for both IPSec and WEP services. Packets for encryption/decryption and/or authentication are prepared by the client and passed to the IxCryptoAcc component via the component API. If the Intel XScale core's WEP Engine is not being used, IxCryptoAcc invokes IxQMgr to instruct the NPEs to gather the data and appropriate key information from SDRAM. The IxCryptoAcc component depends on the IxQMgr component to configure and use the hardware queues to access the NPE.

The NPE (or Intel XScale core WEP Engine) performs encryption/decryption and authentication using the appropriate acceleration component. The resulting data is stored back into the SDRAM. At this point, a previously registered callback will be executed (in most cases), giving the context back to the client application.

The basic API flow described above is shown in Figure 16.

Figure 16. Basic IxCryptoAcc API Flow



7.3.3 Context Registration and the Cryptographic Context Database

The IxCryptoAcc access component supports up to 10,000 simultaneous security association (SA) tunnels. While the term SA is well-known in the context of IPSec services, the IxCryptoAcc component defines these security associations more generically, as they can be used for WEP services as well. Each cryptographic “connection” is defined by registering it as a cryptographic context containing information such as algorithms, keys, and modes. Each of these connections is given an ID during the context registration process and stored in the Cryptographic Context Database. The information stored in the CCD is stored in a structure detailed below, and is used by the NPE or Intel XScale core WEP Engine to determine the specific details of how to perform the cryptographic processing on submitted data.

The context-registration process creates the structures within the CCD, but the crypto context for each connection must be previously defined in an IxCryptoAccCtx structure. The IxCryptoAccCtx structure contains the following information:

- The type of operation for this context. For example, encrypt, decrypt, authenticate, encrypt and authenticate, etc.

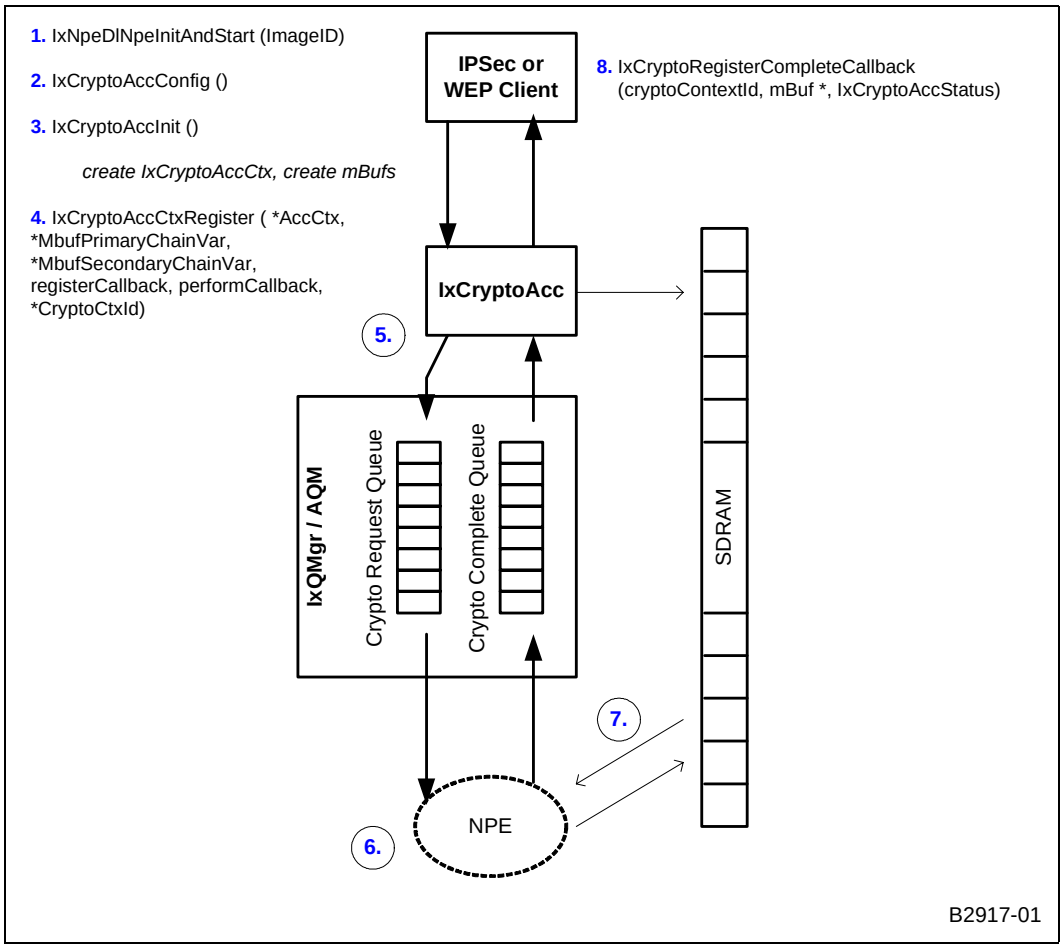
- Cipher parameters, such as algorithm, mode, and key length
- Authentication parameters, such as algorithm, digest length, and hash length
- In-place versus non in-place operation. In-place operation means the once the cryto processing of the source data is completed, the resulting data is placed onto the same mBuf as it was read from.

When the client performs calls the `ixCryptoAccCtxRegister`() function, the following data must be provided or received:

- The client provides a pointer to the crypto context (i.e., SA definition) being registered.
- The client is required to allocate two mBufs to the hardware accelerator will populate with the primary and secondary chaining variables.
- The client must register two callbacks. One callback is executed upon the completion of the registration function, the second is executed each time a cryptographic procedure (“perform” functions) has completed on the NPE for this context. There is one exception for the perform callback function, noted in section “ixCryptoAccXscaleWepPerform()” on page 97.
- The function returns a context ID upon successful registration in the CCD.

Figure 17 on page 82 shows the IxCryptoAcc API call process flow that occurs when registering security associations within the CCD. This process is identical for both IPSec and WEP services except in situations where NPE-based acceleration will not be used, such as when using WEP services using only the Intel XScale core WEP engine. For more detailed information on this usage model see “ixCryptoAccXscaleWepPerform()” on page 97.

Figure 17. IxCryptoAcc API Call Process Flow for CCD Updates



1. The proper NPE microcode images must be downloaded to the NPEs and initialized, if applicable.
2. IxCryptoAcc must be configured appropriately according to the NPEs and services that will be utilized. By default, IxCryptoAccConfig() configured the component for using NPE C and enabled the Intel XScale core WEP engine.
3. IxCryptoAcc must be initialized. At this point the client application should define the crypto context to be registered, as well as create the buffers for the initial chaining variables.
4. The crypto context must be registered using the IxCryptoAccCtxRegister() function.
5. The IxCryptoAcc API will write the crypto context structure to SDRAM. If NPE-based acceleration is being used, IxCryptoAcc will use IxQMgr to place a descriptor for the crypto context being registered into the Crypto Request Queue.
6. The NPE will read the descriptor on the Crypto Ready Queue, generate any reverse keys required, and generate the initial chaining variable if required.
7. The NPE or Intel XScale core WEP Engine writes the resulting data in the Crypto Context Database residing in SDRAM. The NPE will then enqueue a descriptor onto the Crypto Complete Queue to alert the IxCryptoAcc component that registration is complete.

8. IxCryptoAcc will return a context Id to the client application upon successful context registration, and will call the Register Complete callback function.

7.3.4 Buffer and Queue Management

The mBuf buffer format is expected to be used between the IxCryptoAcc access component and the client. All buffers used between the IxCryptoAcc access component and clients are allocated and freed by the clients. The client will allocate the mBufs required to be passed to IxCryptoAcc. The NPE or Intel XScale core WEP Engine in turn will allocate memory for the CCD. 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 component assumes that the allocated mBufs are sufficient in length and no checking has been put in place for the mBuf length. There is, however, mBuf checking when the code is compiled in DEBUG mode. 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 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. This selection of in-place versus non-in-place buffer operation may be defined for each crypto context prior to context registration.

When the AHB 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.

7.3.5 Memory Requirements

This section shows the amount of data memory required by IxCryptoAcc for it to operate under peak call-traffic load. The IxCryptoAcc component allocates its own memory for the CCD to store the required information, and for the NPE queue descriptors required when using NPE-based acceleration. The total memory allocation follows this general formula:

Total Memory Allocation = (Size of NPE queue descriptor + size of additional authentication data) * Number of descriptors + (size of crypto context) * (number of crypto contexts).

This shows the memory requirements for 1,000 security associations, the default value set by `IX_CRYPTO_ACC_MAX_ACTIVE_SA_TUNNELS`. This value can be increased or decreased as needed by the client.

Table 12. IxCryptoAcc Data Memory Usage (Sheet 1 of 2)

Structure	Size in Bytes	Total Size in Bytes
NPE Queue Descriptor	96	
Additional Authentication Data	64	
Total Memory per NPE Descriptor	96+64=160	
Number of NPE Descriptors	278	
Total Memory Allocated for NPE Descriptors	160 * 278=	44,480
Crypto Context	152	

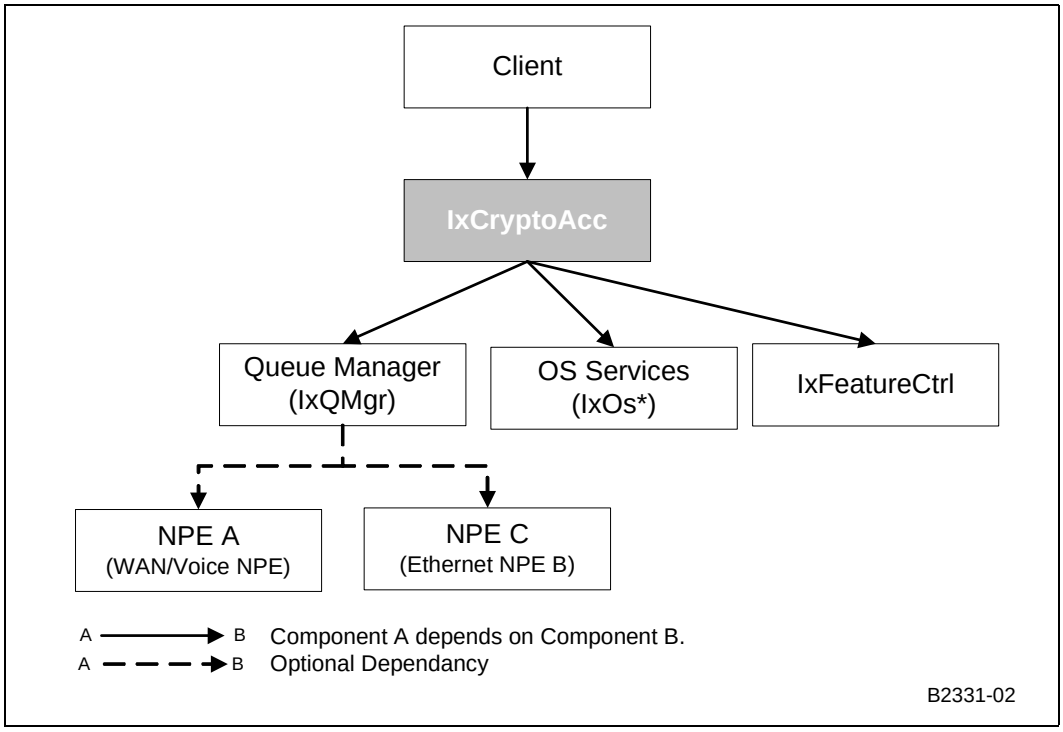
Table 12. IxCryptoAcc Data Memory Usage (Sheet 2 of 2)

Structure	Size in Bytes	Total Size in Bytes
Number of Crypto Context (IX_CRYPTO_ACC_MAX_ACTIVE_SA_TUNNELS)	1,000	
Total Memory Allocated for Crypto Contexts	152 * 1,000=	152,000
Total Memory Allocated by IxCryptoAcc	44,480 + 152,000 =	196,480

7.3.6 Dependencies

Figure 18 shows the component dependencies of the IxCryptoAcc component.

Figure 18. 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 IxQMgr.
- IxCryptoAcc depends on the IxQMgr component to configure and use the hardware queues to access the NPE.
- OS Services components are used for error handling and reporting.
- IxFeatureCtrl is used to detect the processor capabilities at runtime, to ensure the necessary hardware acceleration features are available for the requested cryptographic context registrations.

- In situations where only the Intel XScale core WEP Engine is used, the IxQMgr component is not utilized. Instead, local memory is used to pass context between the IxCryptoAcc API and the Intel XScale core WEP Engine.

After the CCD has been updated, the API can then be used to perform cryptographic processing on client data, for a given crypto context. This service request functionality of the API is described in "IPSec Services" on page 86 and "WEP Services" on page 95.

7.3.7 Other API Functionality

In addition to crypto context registration, IPSec and WEP service requests, the IxCryptoAcc API has a number of other features.

- A number of status definitions, useful for determining the cause of registration or cryptographic processing errors.
- The ability to un-register a specific crypto context from the CCD.
- Two status and statistics functions are provided. These function show information such as the number of packets returned with operation fail, number of packets encrypted/ decrypted/ authenticated, the current status of the queue, whether the queue is empty or full or current queue length.
- The ability to halt the API.

The two following functions are used in specific situations that merit further explanation.

ixCryptoAccHashKeyGenerate()

This function should be used in situations where an HMAC authentication key of greater than 64 bytes is required for a crypto context, and should be called prior to registering that crypto context in the CCD.

ixCryptoAccCtxCipherKeyUpdate()

This function is called to change the key value of a previously registered context. Key change for a registered context is only supported for CCM cipher mode. This is done in order to quickly change keys for CCM mode, without going through the process of context deregistration and registration. Changes to the key lengths are not allowed for a registered context. This function should only be used if one is invoking cryptographic operations using CCM as cipher mode.

The client should make sure that there are no pending requests on the "cryptoCtxtId" for the key change to happen successfully. If there are pending requests on this context the result of those operations are undefined.

For contexts registered with other modes the client should unregister and re-register a context for the particular security association in order to change keys and other parameters.

7.3.8 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

7.3.9 Endianness

The mode supported by this component is both big endian and little endian.

7.3.10 Import and Export 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.

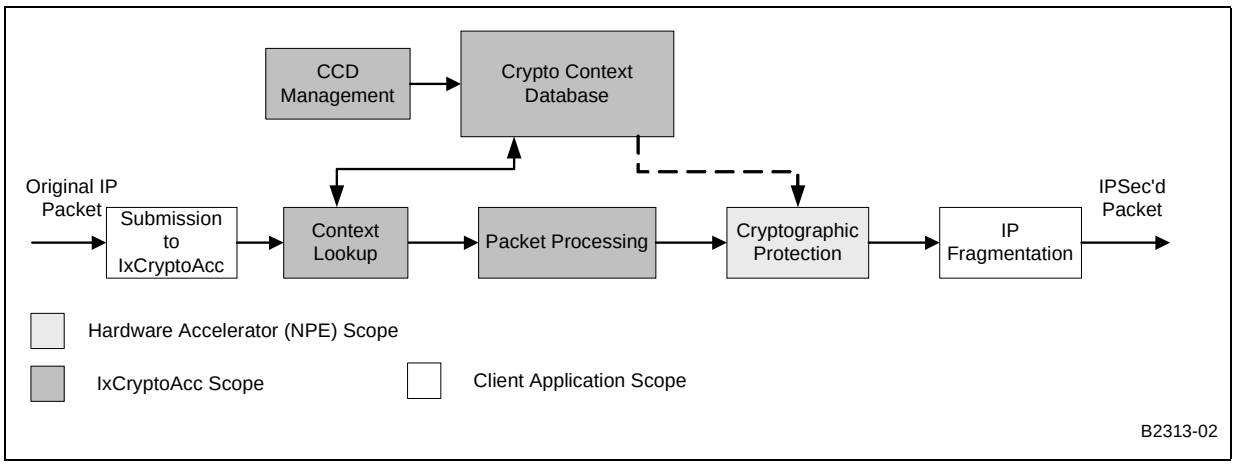
7.4 IPSec Services

This section describes the way that IxCryptoAcc is used in an IPSec usage model.

7.4.1 IPSec Background and Implementation

When deploying IPSec-related applications, the generalized architecture in Figure 19 is used. The figure shows the scope and the roles played by the NPE and the IxCryptoAcc component in an IPSec application.

Figure 19. IxCryptoAcc, NPE 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 (the IxCryptoAcc and supporting components, in this scenario) to encapsulate the IP packet into the ESP or AH packet format.

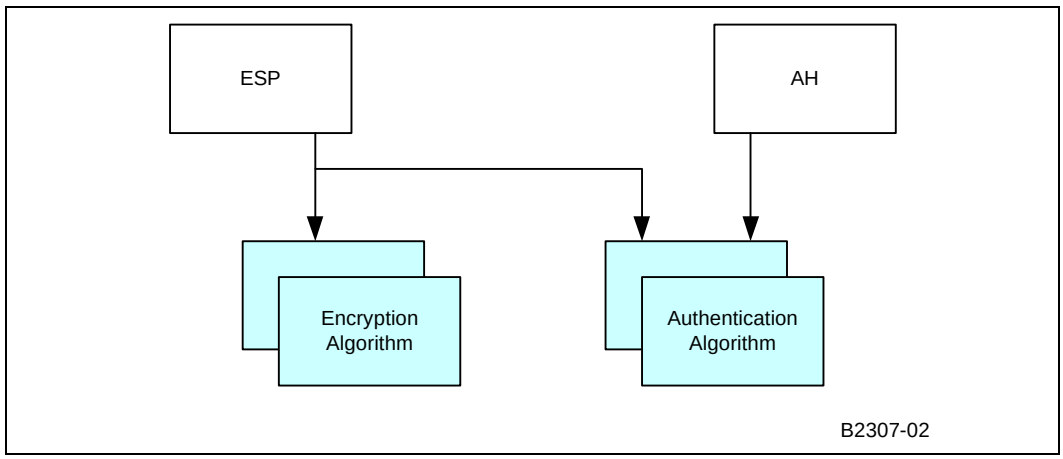
The client looks up the crypto context policy in the CCD database to determine the mode of transporting packets, the IPSec protocol (ESP or AH), etc. The client determines use of the transport or tunnel mode from the registered security context. The mode is transparent to ixCyptoAcc component.

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 whether transport or tunnel mode is used.

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 20 shows the relationship of encryption and authentication algorithms within the IPSec protocol.

Figure 20. Relationship Between IPSec Protocol and Algorithms

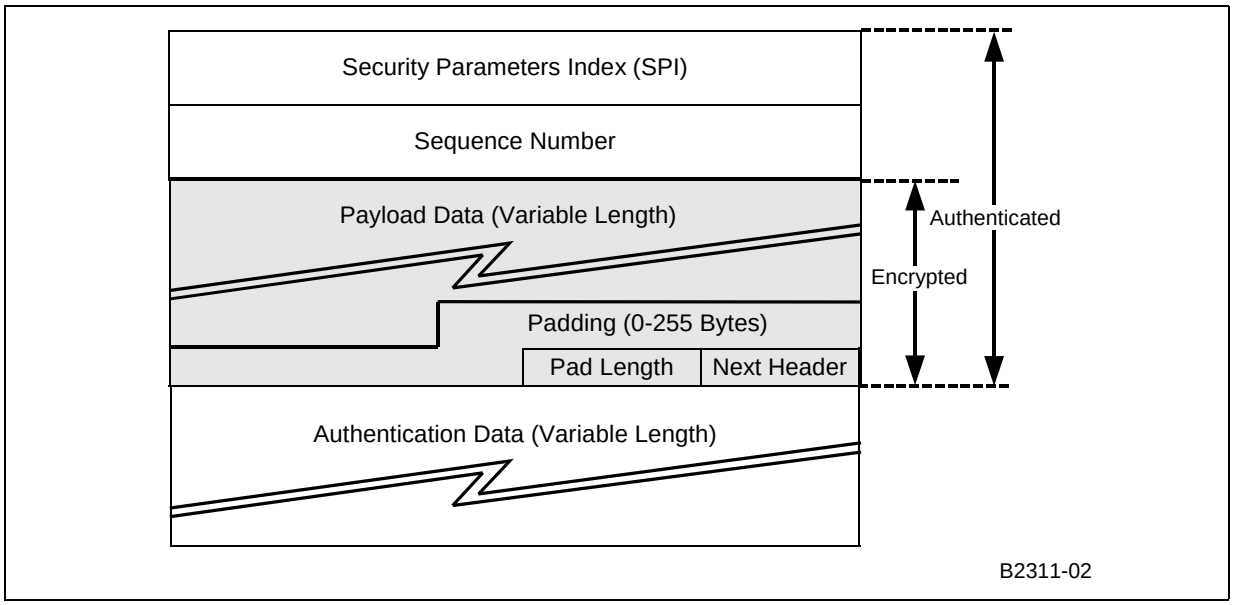


7.4.2 IPSec Packet Formats

IPSec standards have defined 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 both 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.

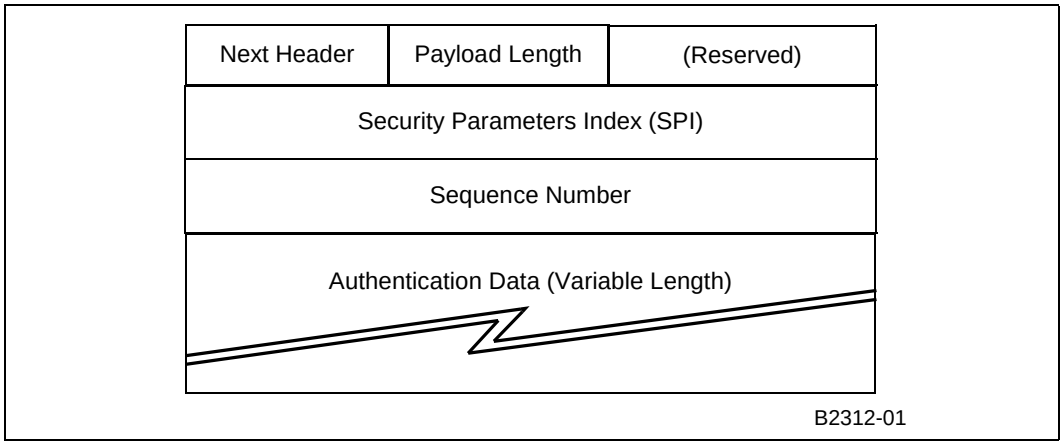
Figure 21. ESP Packet Structure



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 the perform function.

Figure 22. Authentication Header

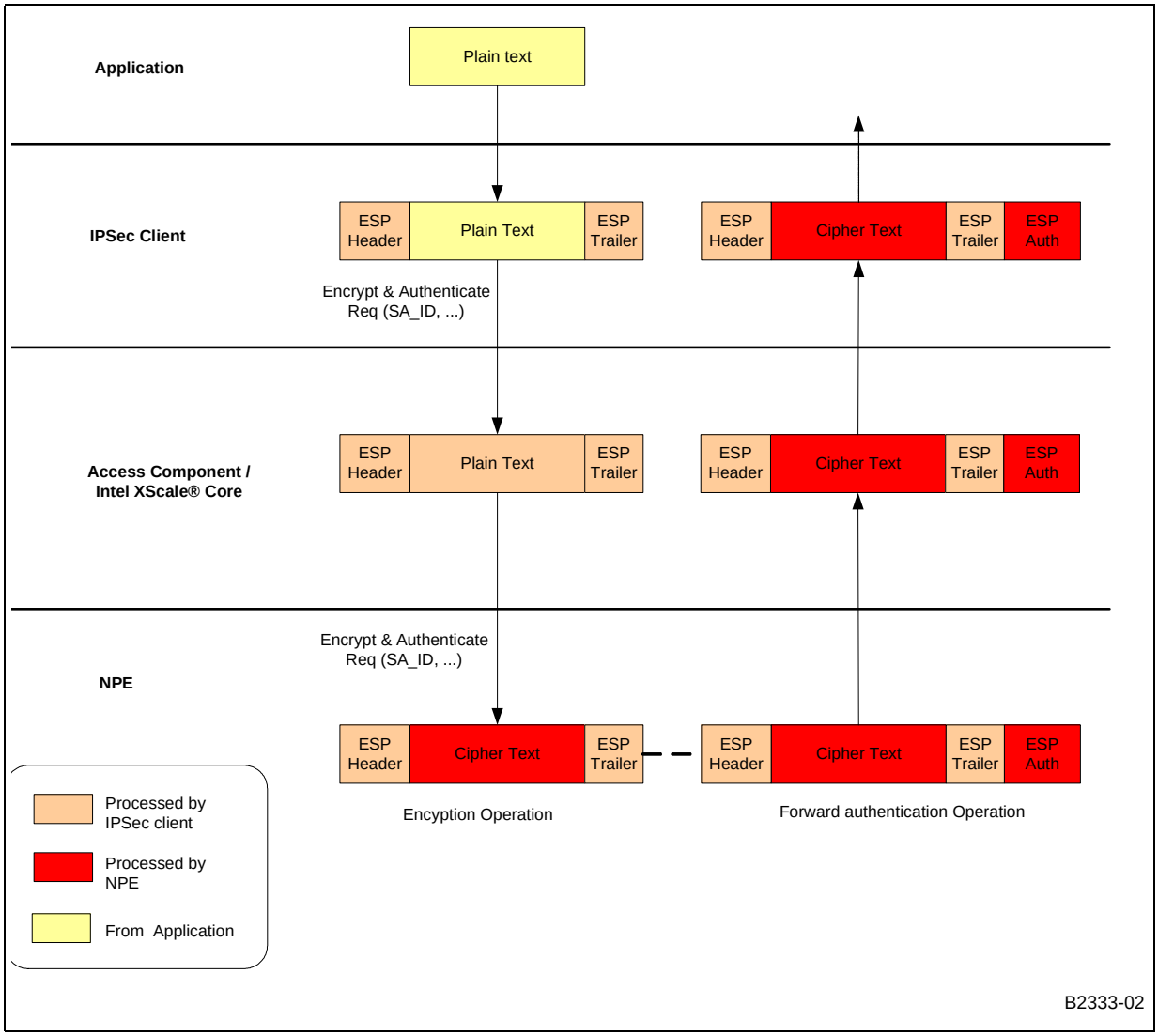


7.4.2.1 Reference ESP Dataflow

Figure 23 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.

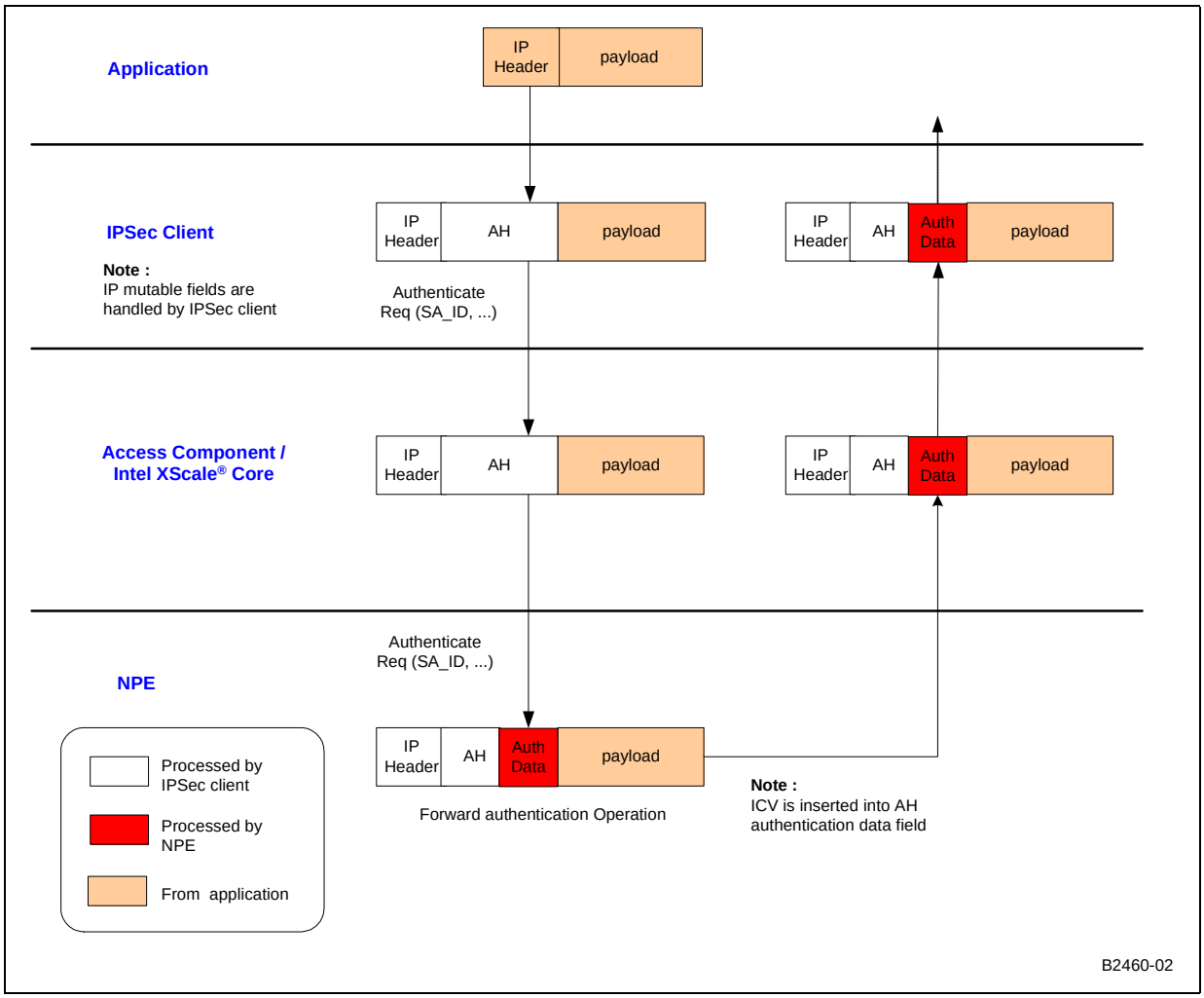
Figure 23. ESP Data Flow



7.4.2.2 Reference AH Dataflow

Figure 24 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 24. AH Data Flow



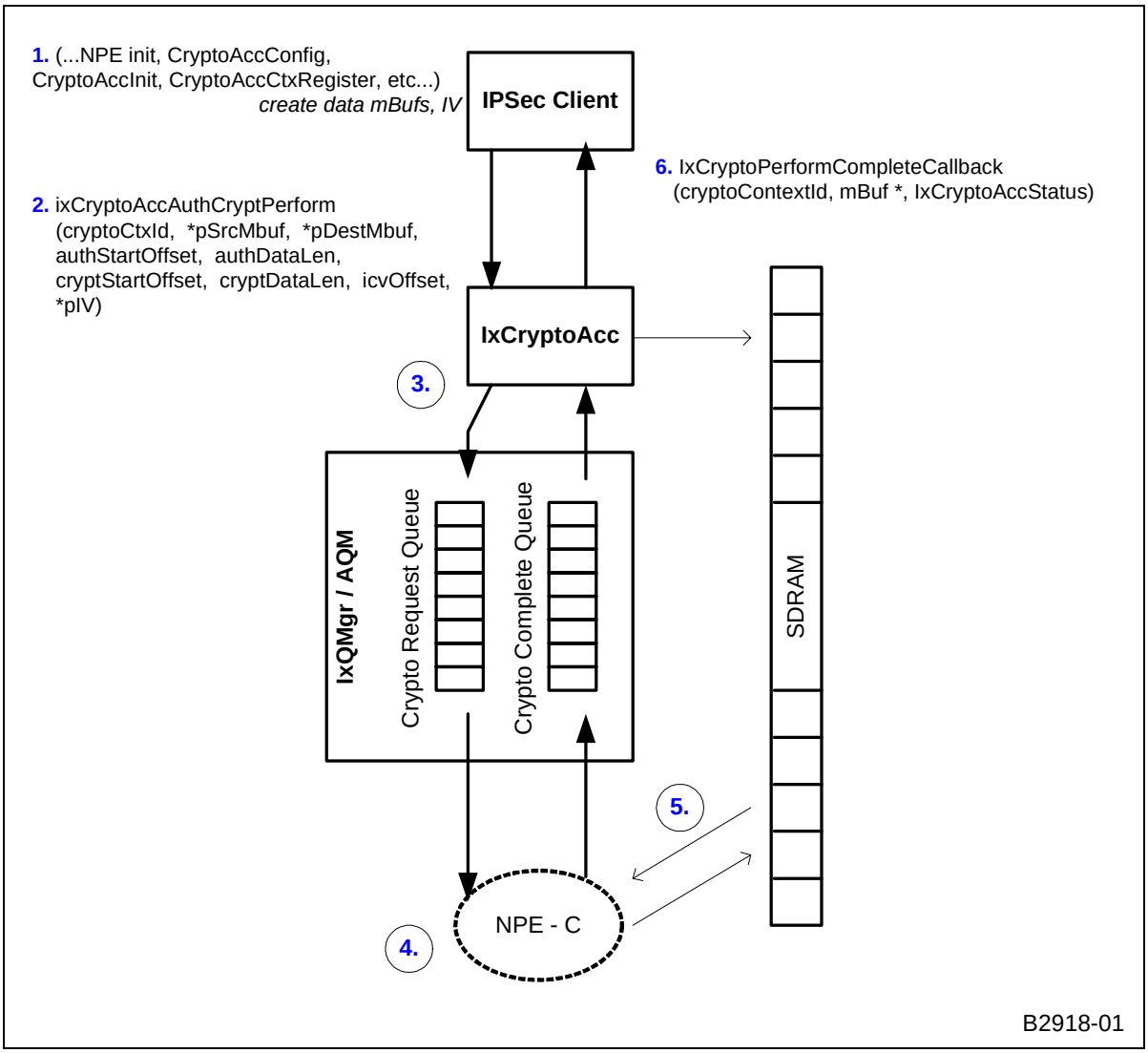
7.4.3 Hardware Acceleration for IPSec Services

The IxCryptoAcc API is dependant upon hardware resources within NPE C (also known as Ethernet NPE B) in order to perform many of the cryptographic encryption, decryption, or authentication functions. Specifically, NPE C provides an AES coprocessor, DES coprocessor and a hashing coprocessor (for MD5 and SHA1 calculations).

7.4.4 IPSec API Call Flow

Figure 25 on page 91 details the IxCryptoAcc API call flow that occurs when submitted data for processing using IPSec services. The process listed below assumes that the API has been properly configured and that a crypto context has been created and registered in the CCD, as described in "Context Registration and the Cryptographic Context Database" on page 80.

Figure 25. IPSec API Call Flow



1. The proper NPE microcode images must have been downloaded to the NPE and initialized. Additionally, the IxCryptoAcc API must be properly configured, initialized, and the crypto context registration procedure must have completed.

   At this point, the client must create the mBufs that will hold the target data and populate the source mBuf with the data to be operated on. Depending on the encryption/decryption mode being used, the client must supply an initialization vector for the AES or DES algorithm.

2. The client submits the IxCryptoAccAuthCryptPerform() function, supplying the crypto context ID, pointers to the source and destination buffer, offset and length of the authentication and crypto data, offset to the integrity check value, and a pointer to the initialization vector.

3. IxCryptoAcc will use IxQMgr to place a descriptor for the data into the Crypto Request Queue.

4. The NPE will read the descriptor on the Crypto Ready Queue and performs the encryption/decryption/authentication operations, as defined in the CCD for the submitted crypto context. The NPE inserts the Integrity Checksum Value (ICV) for a forward-authentication operation and verifies the ICV for a reverse-authentication operation.
5. The NPE writes the resulting data to the destination mBuf in SDRAM. This may be the same mBuf in which the original source data was located, if the crypto context defined in-place operations. The NPE will then enqueue a descriptor onto the Crypto Complete Queue to alert the IxCryptoAcc component that the perform operation is complete.
6. IxCryptoAcc will call the registered Perform Complete callback function.

7.4.5 Special API Use Cases

7.4.5.1 HMAC with Key Size > 64 Bytes

As specified in the RFC 2104, the authentication key used in HMAC operation must be at least of L bytes length, where L = 20 bytes for SHA1 or L = 16 bytes for MD5. Authentication key with key length >= L and <= 64 bytes can be used directly in HMAC authentication operation. No further hashing of authentication key is needed. Thus the authentication key can be used directly in crypto context registration.

However, authentication key with key length greater than 64 bytes must be hashed to become L bytes of key size before it can be used in HMAC authentication operation. The authentication key must be hashed before calling crypto context registration API as shown in steps below:

a. Call ixCryptoAccHashKeyGenerate() function and pass in the original authentication key using an mBuf. Also, you will need to register a callback function for when this operation is complete.
b. Wait for callback from IxCryptoAcc.
c. Copy generated authentication key from mBuf into a cyrptographic context structure (IxCryptoAccCtx) and call ixCryptoAccCtxRegister() to register the crypto context for this HMAC operation.

7.4.5.2 Performing CCM (AES CTR-Mode Encryption and AES CBC-MAC Authentication) for IPSec

A generic CCM cipher is not supported in the IXP400 software. However, it is possible to perform AES-CCM operations in an IPSec-application style. Single-pass AES-CCM is supported for WEP Services only, as documented in “Counter-Mode Encryption with CBC-MAC Authentication (CCM) for CCMP in 802.11i” on page 100.

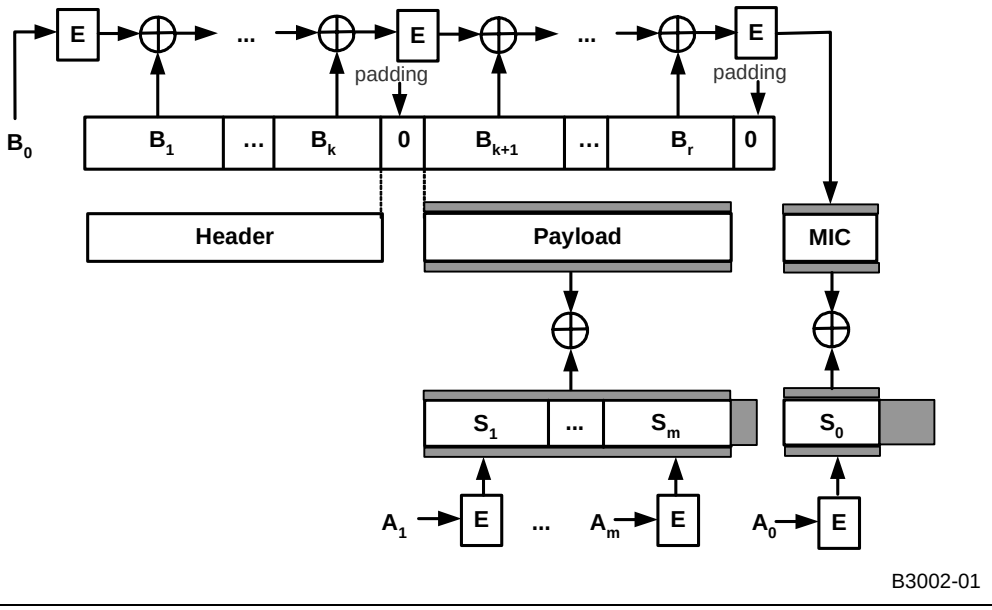
The overall strategy to accomplish the AES-CCM request involves two operations. The first operation does the AES-CBC operation to get the CBC-MAC. The second operation is to perform a AES-CTR encryption operation to encrypt the payload and create the CBC-MAC to get the MIC. Two crypto contexts are registered and two crypto perform service requests are invoked in order to complete the encryption and authentication for a packet.

Figure 26 on page 93 and Figure 27 on page 93 show the steps needed to encrypt and authenticate a packet in general by using CCM mode. Those steps are:

1. Use AES CBC-MAC to compute a MIC on plaintext header, and payload.
   The last cipher block from this operation will become MIC.

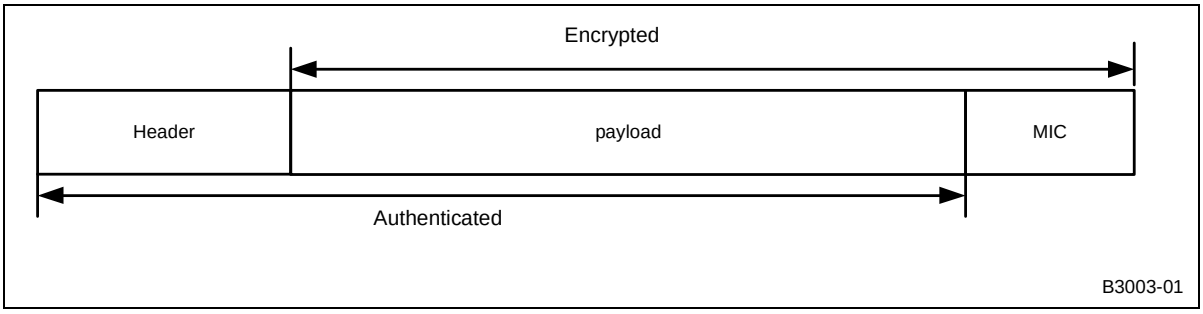
2. Use AES-CTR mode to encrypt the payload with counter values 1, 2, 3, …
3. Use AES-CTR mode to encrypt the MIC with counter value 0 (First key stream (S0) from AES-CTR operation)

Figure 26. CCM Operation Flow



Legend:
E – Encryption operation
B – Data blocks to be encrypted (data is split into multiple blocks of the size of cipher block length)
S – Key stream generated from AES-CTR encryption
A – Counter blocks for AES-CTR encryption
MIC – Message Integrity Code

Figure 27. CCM Operation on Data Packet



The API usage for performing an IPSec-style AES-CCM operation is as follows:

1. Register a crypto context for AES-CBC encryption (cipher context). A crypto context ID (A, in this example) will be obtained in this operation. Non in-place operation must be chosen (useDifferentSrcAndDestMbufs in IxCryptoAccCtx must set to TRUE) to avoid the original data being overwritten by encrypted data. This crypto context is used only for the purpose of authentication and generating the MIC.

2. Register another crypto context for AES-CTR encryption (cipher context). A crypto context ID (B) will also be obtained in this operation. This crypto context is used for payload and MIC encryption only.
3. After both crypto context registration for both contexts is complete, call the crypto perform API using context ID A. The IV for this packet is inserted as first block of message in the packet. The input IV to the crypto perform function is set to zeroes. Crypt start offset and crypt data length parameters are set to the same values as authentication start offset and authentication data length, as shown in Figure 28 on page 94. Authentication start offset and authentication data length can be ignored in the API for this operation, as this is an encryption operation only. The client should handle all the above-mentioned steps before calling the crypto perform function.
4. Wait for the operation in step 3 to complete and extract the MIC from the destination mBuf using the callback function.
5. Append the MIC from step 4 into the mBuf before the payload data.
6. Call the crypto perform function with crypto context ID B. Change the crypt start offset to point to the start offset of the MIC and change the crypt data length to include the length of MIC, as shown in Figure 29 on page 94.
7. Wait for operation in step 6 to complete and move the MIC back to its original location in mBuf. The MIC is now the final authentication data.

Figure 28. AES CBC Encryption For MIC

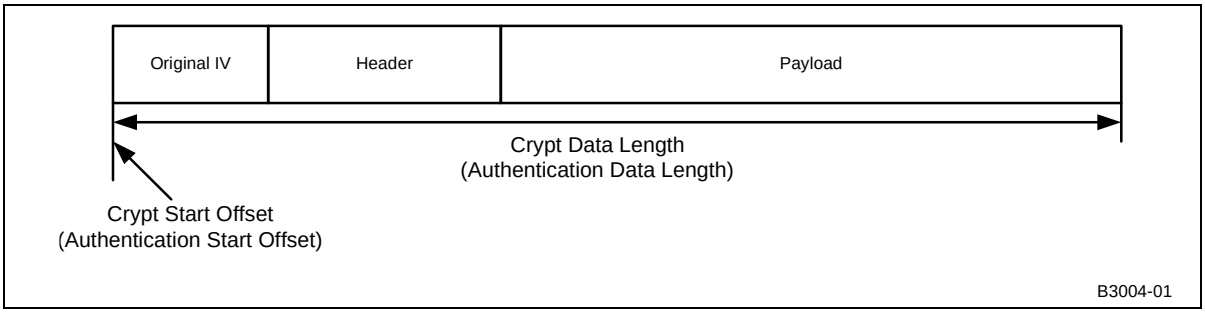
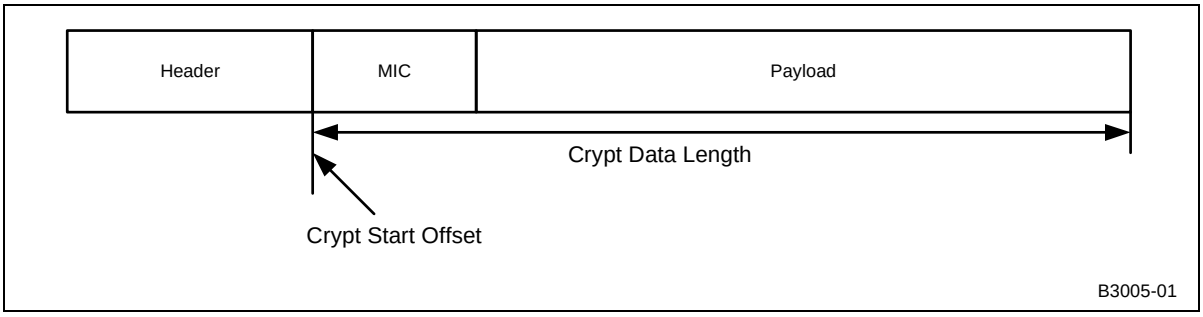


Figure 29. AES CTR Encryption For Payload and MIC



Since the data has to be read twice by the NPE, this two-pass mechanism will have slower throughput rate compared to the other crypto perform operations that combine encryption and authentication.

Note that memory copying is needed when performing the CCM request on a packet as mentioned above. Chained mBufs could be used to avoid excessive memory copying in order to get better performance. If a single mBuf is used, memory copying is needed to insert MIC from AES-CBC

operation into the packet, between header and payload. The payload needs to be moved in order to hold MIC in the packet. An efficient method of doing this could be to split the header and payload into two different mBufs. Then the MIC can be inserted after the header into the header mBuf for the AES CTR encryption operation.

7.4.6 IPSec Assumptions, Dependencies, and Limitations

- Mutable fields in IP headers should be set to a value of 0 by the client.
- The client must pad the IP datagram to be a multiple of the cipher block size, using ESP trailer for encryption (RFC 2406, explicit padding).
- The IxCryptoAcc component handles any necessary padding required during authentication operations, where the IP datagram is not a multiple of the authentication algorithm block size. The NPE pads the IP datagram to be a multiple of the block size, specified by the authentication algorithm (RFC 2402, implicit padding).
- The client must provide an initialization vector to the access component for the DES or AES algorithm, in CBC mode and CTR mode.
- IxCryptoAcc generates the primary and secondary chaining variables which are used in authentication algorithms.
- IxCryptoAcc generates the reverse keys from the keys provided for AES algorithm.

7.5 WEP Services

7.5.1 WEP Background and Implementation

The Wired Equivalent Privacy (WEP) specification is designed to provided a certain level of security to wireless 802.11 connections at the data-link level. The specification dictates the use of the RC4 cryptographic algorithm and the use of a CRC-32 authentication calculation (the Integrity Check Value) on the payload and data header.

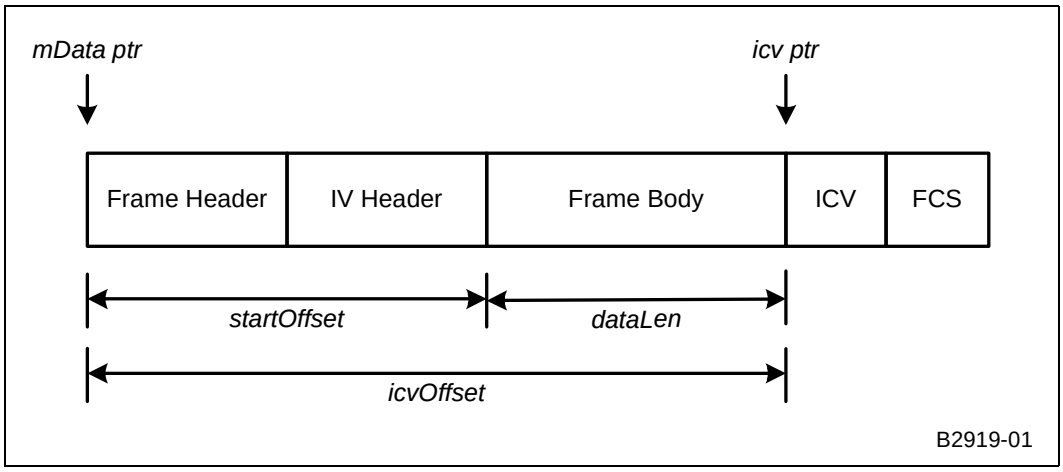
The IxCryptoAcc API provides both the encryption/decryption and authentication calculation or verification in a single-pass implementation. The API uses two functions for performing WEP service operations, depending on the hardware-acceleration component being utilized.

ixCryptoAccXScaleWepPerform() is used to submit data for WEP services using the Intel XScale core-based WEP engine.

ixCryptoAccNpeWepPerform() is used to submit data for WEP services using the hardware acceleration services of NPE A.

Both functions operate in a substantially similar manner, taking in the parameters discussed below and shown in Figure 30.

Figure 30. WEP Frame with Request Parameters



- *pSrcMbuf - 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.
- *pDestMbuf — 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 *cannot* span into two or more different mBufs, thus the mBuf supplied must be at least the length of expected result. The data is written back starting at startOffset in the pDestMbuf.
- startOffset — Supplied by the client to indicate the start of the payload to be decrypted/ encrypted or authenticated.
- dataLen — Supplied by the client to indicate the length of the payload to be decrypted/ encrypted in number of bytes.
- icvOffset — Supplied by the client to indicate the start of the ICV (Integrity Check Value) used for the authentication. This ICV field should not be split across multiple mBufs in a chained mBuf.
- *pKey — Pointer to IX_CRYPTO_ACC_ARC4_KEY_128 bytes of per packet ARC4 keys. This pointer can be NULL if the request is WEP IV gen or verify only.

In the figure above, it is assumed for the sake of simplicity that mData is a contiguous buffer starting from byte 0 to the end of the FCS.

FCS is not computed or touched by the component.

7.5.2 Hardware Acceleration for WEP Services

The WEP services provided in IxCryptoAcc depend on hardware-based resources for some of the cryptographic functions. This differs from the model of NPE-based hardware acceleration typically found in the IXP400 software in that the client software can select to use NPE-based acceleration or an Intel XScale core-based software engine that both provide equivalent functionality.

These acceleration components provide the following services to IxCryptoAcc:

- ARC4 (Applied RC4) encryption / decryption
- WEP ICV generation and verification

The API provides two functions for performing WEP operations. ixCryptoAccXScaleWepPerform() is used to submit data for WEP services using the Intel XScale core-based WEP engine. ixCryptoAccNpeWepPerform() is used to submit data for WEP services using the hardware acceleration services of NPE A.

It is important to note that the perform requests are always executed entirely on the specified engine. However, a single crypto context may be submitted to either engine. There are some specific behavioral characteristics for each engine.

ixCryptoAccNpeWepPerform()

The NPE-based WEP perform function acts identically to the IPSec service perform functions in terms of callback behavior. During crypto context registration, a callback is specified to be executed upon completion of the perform operation. For ixCryptoAccNpeWepPerform(), this callback is executed asynchronously. When the NPE has completed the required processing, it will initiate the client callback.

ixCryptoAccXscaleWepPerform()

The WEP perform function using the Intel XScale core WEP engine has two distinct differences from the NPE-based function.

First, ixCryptoAccXscaleWepPerform() operates synchronously. This is to say that once the perform function is submitted, the Intel XScale core function retains the context until the perform operation is complete. The Intel XScale core perform function will not execute the registered *performCallback* function. The client should initiate any local callback function on its own.

The second behavior difference is that the Intel XScale core perform function does not support non-in-place memory operations.The function returns an error if the non-in-place operation is requested.

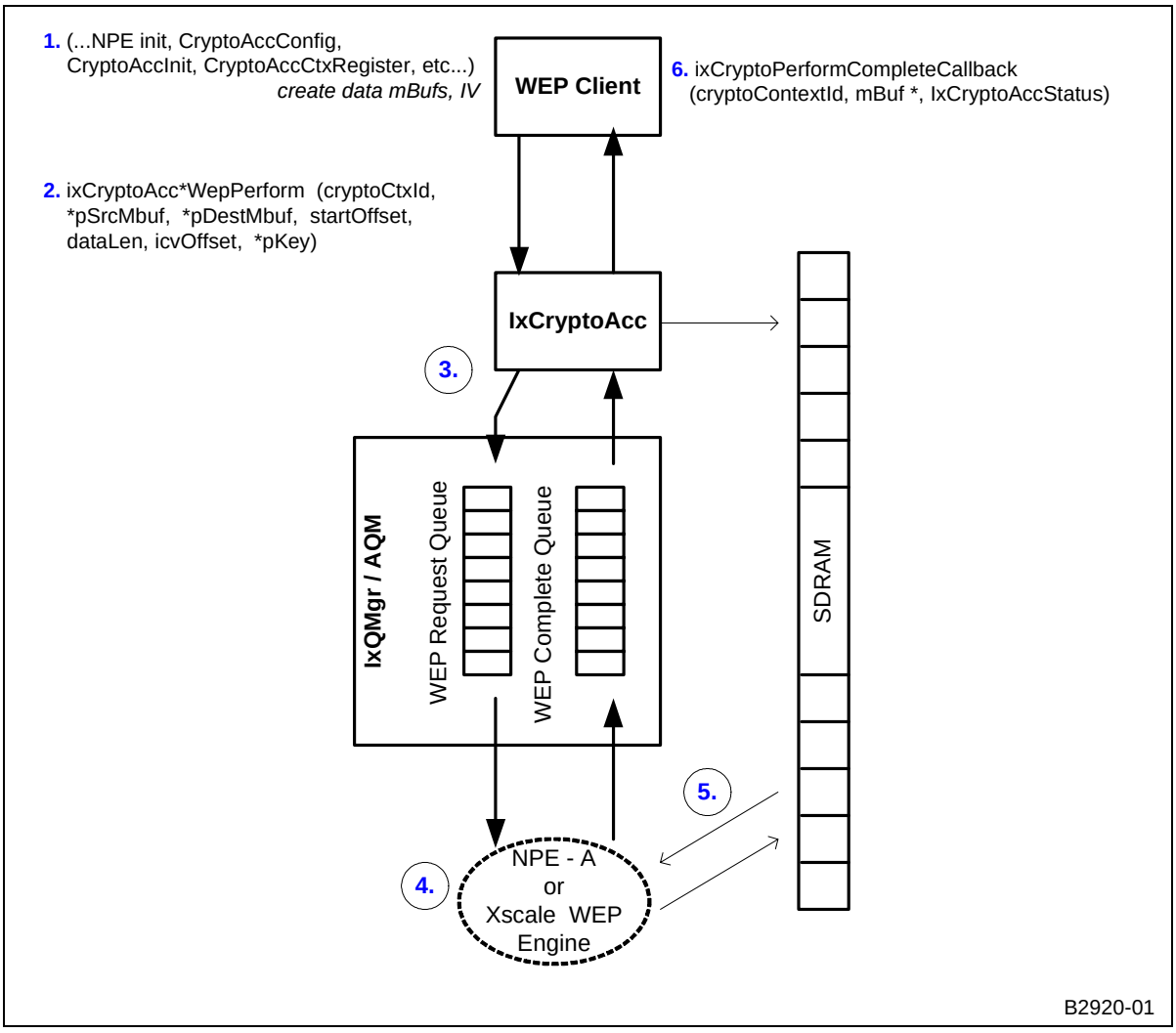
NPE Microcode Images

The WEP NPE image IX_NPEDL_NPEIMAGE_NPEA_WEP makes autonomous use of NPE A (also known as the WAN/Voice NPE) and cannot be used simultaneously with any other NPE images on NPE A. Should the product design require NPE A be used for another purpose (DMA or ATM processing, for example), then the Intel XScale core WEP engine should be used.

7.5.3 WEP API Call Flow

Figure 31 on page 98 details the IxCryptoAcc API call flow that occurs when submitted data for processing using WEP services. The process listed below assumes that the API has been properly configured and that a crypto context has been created and registered in the CCD, as described in “Context Registration and the Cryptographic Context Database” on page 80.

Figure 31. WEP Perform API Call Flow



1. The proper NPE microcode images must have been downloaded to the NPE and initialized. Additionally, the IxCryptoAcc API must be properly configured, initialized, and the crypto context registration procedure must have completed.

   At this point, the client must create the mBufs that will hold the target data and populate the source mBuf with the data to be operated on. The client must supply the initialization vector for the ARC4 algorithm.

2. The client submits the ixCryptoAccNpeWepPerform() or ixCryptoAccXscaleWepPerform() function, supplying the crypto context ID, pointers to the source and destination buffer, offset and length of the authentication and crypto data, offset to the integrity check value, and a pointer to the initialization vector.

3. IxCryptoAcc will use IxQMgr to place a descriptor for the data into the Crypto Request Queue.

4. The NPE will read the descriptor on the Crypto Ready Queue and performs the encryption/decryption/authentication operations, as defined in the CCD for the submitted crypto context. The NPE will also insert or verify the WEP ICV integrity check value.
5. The NPE writes the resulting data to the destination mBuf in SDRAM. This may be the same mBuf in which the original source data was located, if the crypto context defined in-place operations. The NPE will then enqueue a descriptor onto the Crypto Complete Queue to alert the IxCryptoAcc component that the perform operation is complete.
6. If the ixCryptoAccNpeWepPerform() function was executed in Step 2, IxCryptoAcc will call the registered Perform Complete callback function. Otherwise the client will need initiate any callback-type actions itself.

7.6 Supported Encryption and Authentication Algorithms

7.6.1 Encryption Algorithms

IxCryptoAcc supports four different ciphering algorithms

- Data Encryption Standard (DES)
- Triple DES
- Advanced Encryption Standard (AES)
- ARC4 (Applied RC4)

Table 13 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 13. 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
ARC4	128	NA	128	8

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.

3DES

The order the keys are passed in should be Key 1, Key 2, and Key 3.

ARC4

The ARC4 algorithm can only be used in standalone mode or along with WEP-CRC algorithm. It cannot be combined with any other authentication algorithms, like HMAC-SHA1 and HMAC-MD5. ARC4 keys used in WEP are generally 8 bytes (64-bit) or 16 bytes (128-bit). The ARC4 engine expects to be passed a key of 16 bytes in length, where it then copies the key to fill a 256-byte buffer. Therefore, if the key being used by the client is 8 bytes long, then the client should repeat it to fill the 16 bytes of key buffer.

7.6.2 Cipher Modes

There are four cipher modes supported by the NPE:

- Electronic code book (ECB)
- Cipher block chaining (CBC)
- Counter Mode (CTR)
- Counter-Mode / CBC-MAC Protocol (CCMP)

7.6.2.1 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.

7.6.2.2 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).

7.6.2.3 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.

The counter block is always constructed by the client.

7.6.2.4 Counter-Mode Encryption with CBC-MAC Authentication (CCM) for CCMP in 802.11i

A protocol based on AES and Counter-Mode/CBC-MAC is being adopted for providing enhanced security in wireless LAN networks. This protocol is called Counter-Mode/CBC-MAC Protocol (CCMP). The standard defines the CCMP encapsulation/decapsulation processes, CCMP-MPDU formats, CCMP-states and CCMP-procedures. Section on CCMP-Procedures provides details for

constructing CCM initial block (also called MIC-IV), MIC-Headers for performing CCMP MIC computation and CCM-CTR mode IV construction for performing CCM-CTR mode encryption/ decryption.

The hardware accelerator component provides an interface for performing a single pass CCMP-MIC computation and verification with CTR mode encryption /decryption.

Note: The implementation of AES-CCM mode in IxCryptoAcc is designed to support 802.11i type applications specifically. As noted below, the API expects a 48-byte Initialization Vector and an 8-byte MIC value. These values correspond with an 802.11i AES-CCM implementation. IPSEC implementations are expected to support 16 or 32-bit IV's and 8 or 16-bit MIC values, which are not supported by this component. Refer to "Performing CCM (AES CTR-Mode Encryption and AES CBC-MAC Authentication) for IPSec" on page 92 for details on non-WEP AES-CCM operations.

The following should be noted regarding the support for CCMP:

- The hardware accelerator component does not provide any support for constructing CCM initial block construction for MIC computation,
- The hardware accelerator component does not provide any support for constructing MIC-IV and MIC-Headers, and
- The hardware accelerator component does not provide any support to construct CTR-mode IV.
- The hardware accelerator expects that the initialization vector be 64 bytes of contiguous buffer consisting of 16 bytes of CTR-mode - IV followed by 48 bytes of MIC-IV-HEADER. If the MIC-IV-HEADER constructed is less than 48 bytes, then it should be padded with zero to 48 bytes (3 AES- blocks).
- Computed MIC is always 8 bytes and is not configurable to a different value.
- The hardware accelerator does the padding (with zeros, if required) of the data for the purposes of MIC computation. Once MIC is computed and the data has been encrypted the pad bytes are discarded and are not appended to the payload.
- CTR mode -IV, MIC-IV and MIC Headers are constructed by the client from RSN Header and other per packet information.

7.6.3 Authentication Algorithms

Table 14 summarizes the authentication algorithms supported by IxCryptoAcc. The HMAC algorithms are accelerated by the hashing coprocessor on NPE C. The WEP-CRC algorithm may be performed using either NPE A or the Intel XScale core WEP engine.

Table 14. Supported Authentication Algorithms

Authentication Algorithm Supported	Data Block Size (Bits)	Key Size (Bits)
HMAC-SHA1	512	160-512
HMAC-MD5	512	128-512
WEP-CRC	8	-

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

This chapter describes the Intel® IXP400 Software v1.4's "DMA Access Driver" access-layer component.

8.1 What's New

There are no changes or enhancements to this component in software release 1.4.

8.2 Overview

The IxDmaAcc provides DMA capability to offload large data transfers between peripherals in the IXP42X product line and IXC1100 control plane 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.

8.3 Features

The IxDmaAcc component provides these features:

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

8.4 Assumptions

The DMA service is predicated on the following assumptions:

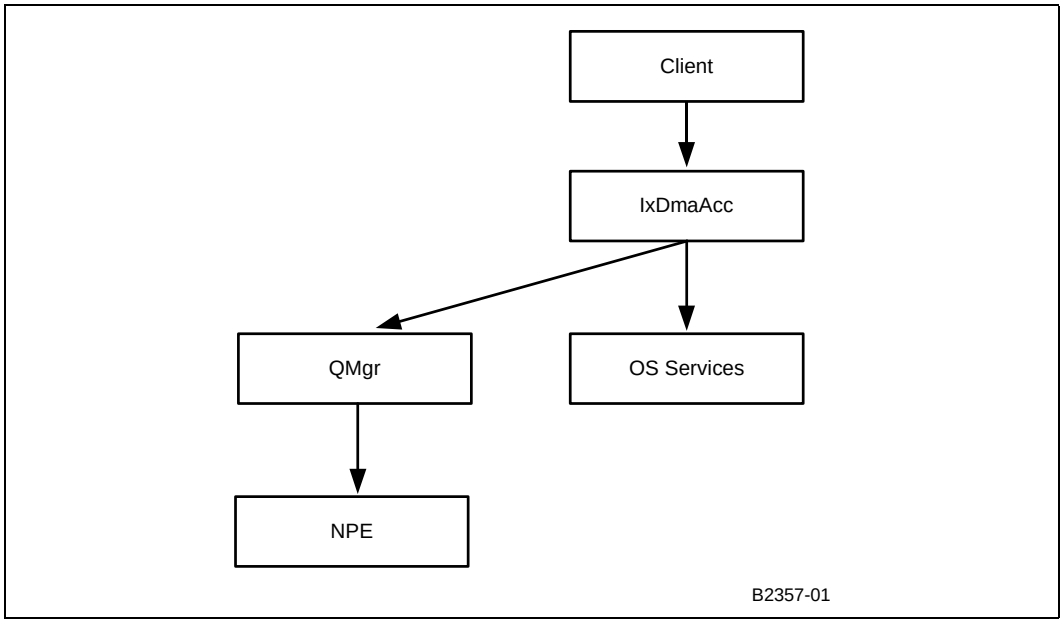
- IxDmaAcc has no knowledge about the IXP42X product line and IXC1100 control plane processors memory map.The client needs to verify the validity of the source address and destination address of the DMA transfer.
- IxDmaAcc has no knowledge on the devices that involve in the DMA transfer. The client is responsible for ensuring the devices are initialized and configured correctly before request for DMA transfer.

8.5 Dependencies

Figure 32 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

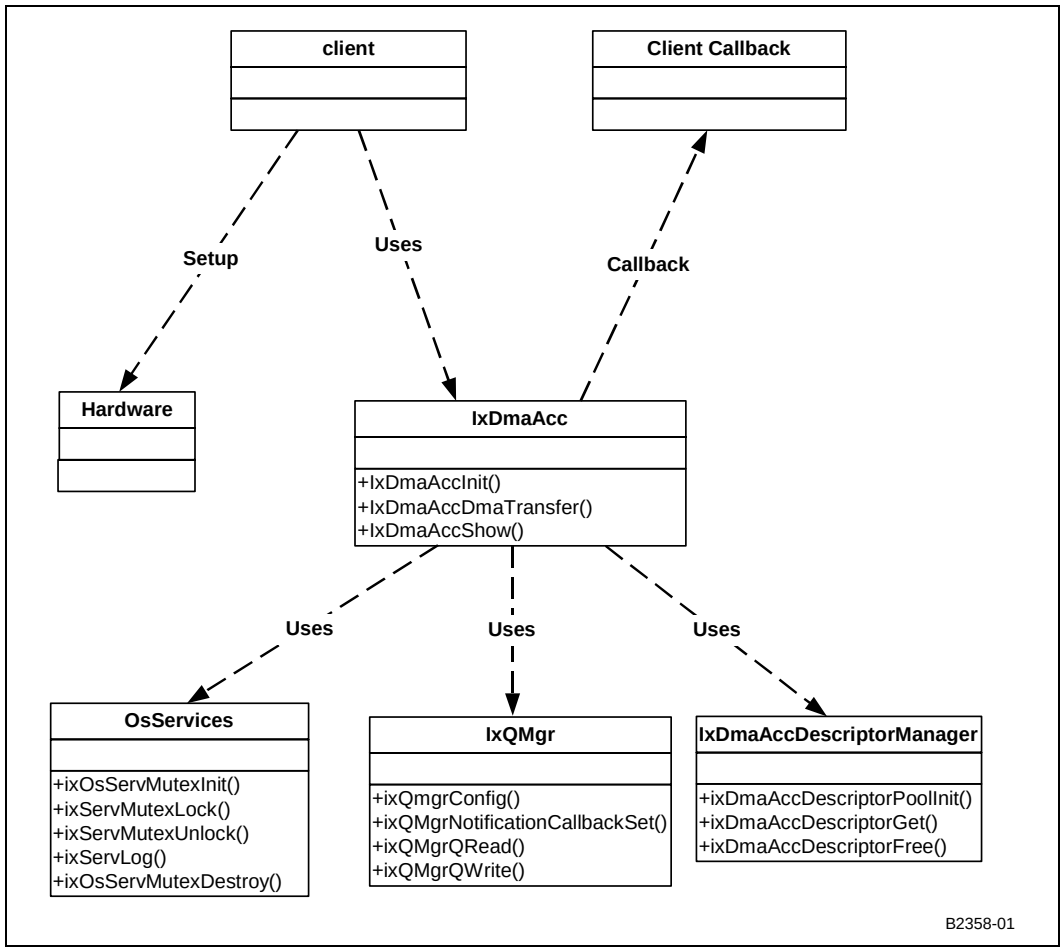
Figure 32. ixDmaAcc Dependencies



8.6 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 33. IxDmaAcc Component Overview



Note: IxDmaAcc components are in white.

Figure 33 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).

8.6.1 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.

Note: The IxDmaAcc component addressing space for physical memory is limited to 28 bit. Therefore mBuf headers should be located in the first 256 Mbyte of physical memory.

8.7 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.

8.7.1 Source Address

Source address is a valid IXP42X product line and IXC1100 control plane processors 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 and IXC1100 control plane processors' memory map.

8.7.2 Destination Address

Destination address is a valid IXP42X product line and IXC1100 control plane processors' 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 and IXC1100 control plane processors memory map.

8.7.3 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).

8.7.4 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).

8.7.5 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).

8.7.6 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

8.7.7 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 115.

Table 15. 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 16. 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 17. 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

8.8 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.

8.9 Control Flow

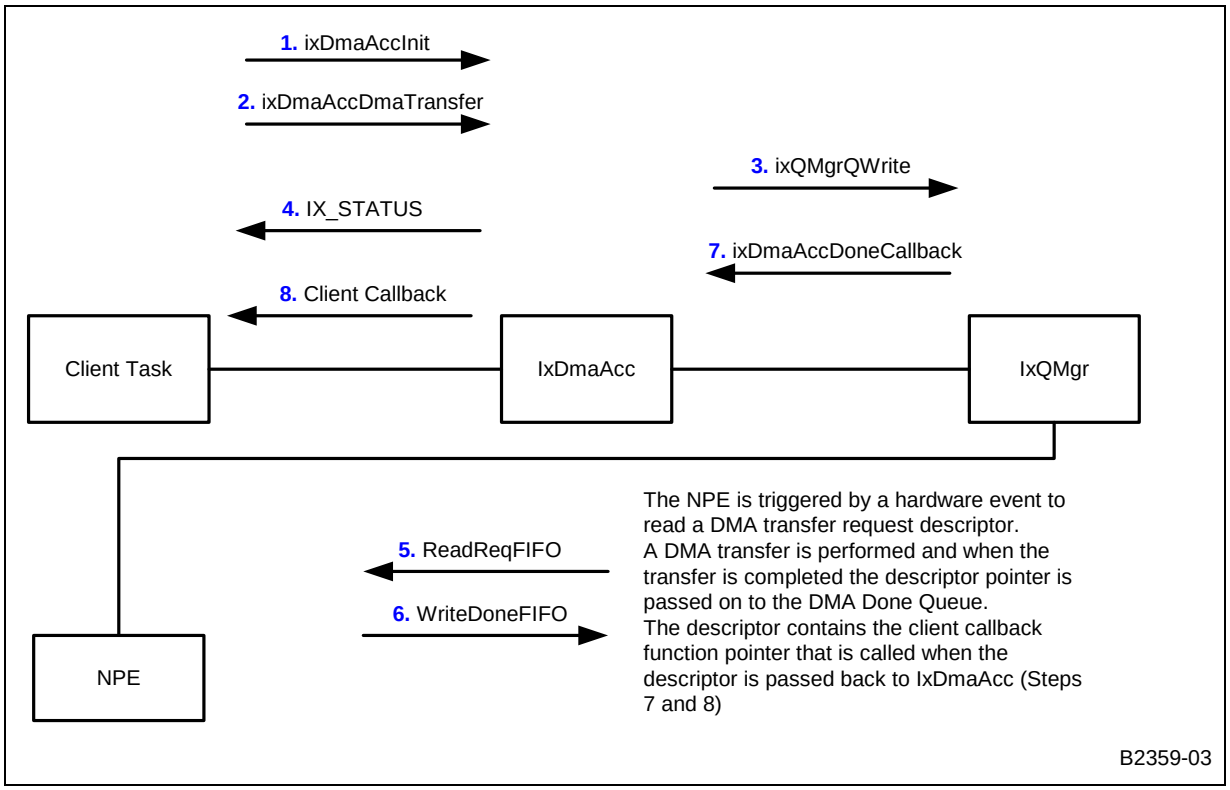
For a DMA transaction to start, the client must initialize the DMA access layer, write to the queue manager, and receive a status of the transaction.

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

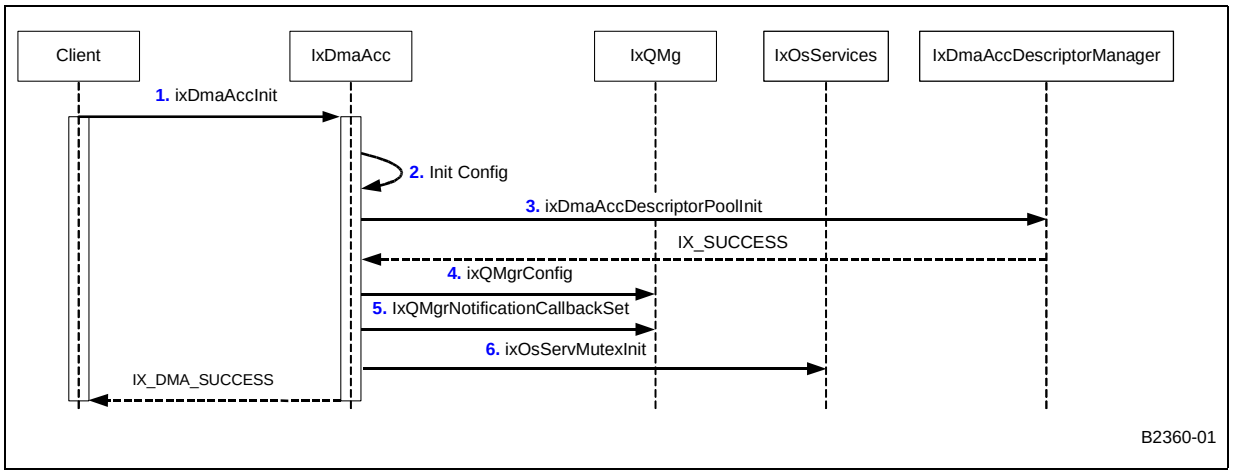
Figure 34. IxDmaAcc Control Flow



8.9.1 DMA Initialization

Figure 35 and the following steps describe the DMA access-layer initialization:

Figure 35. 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.

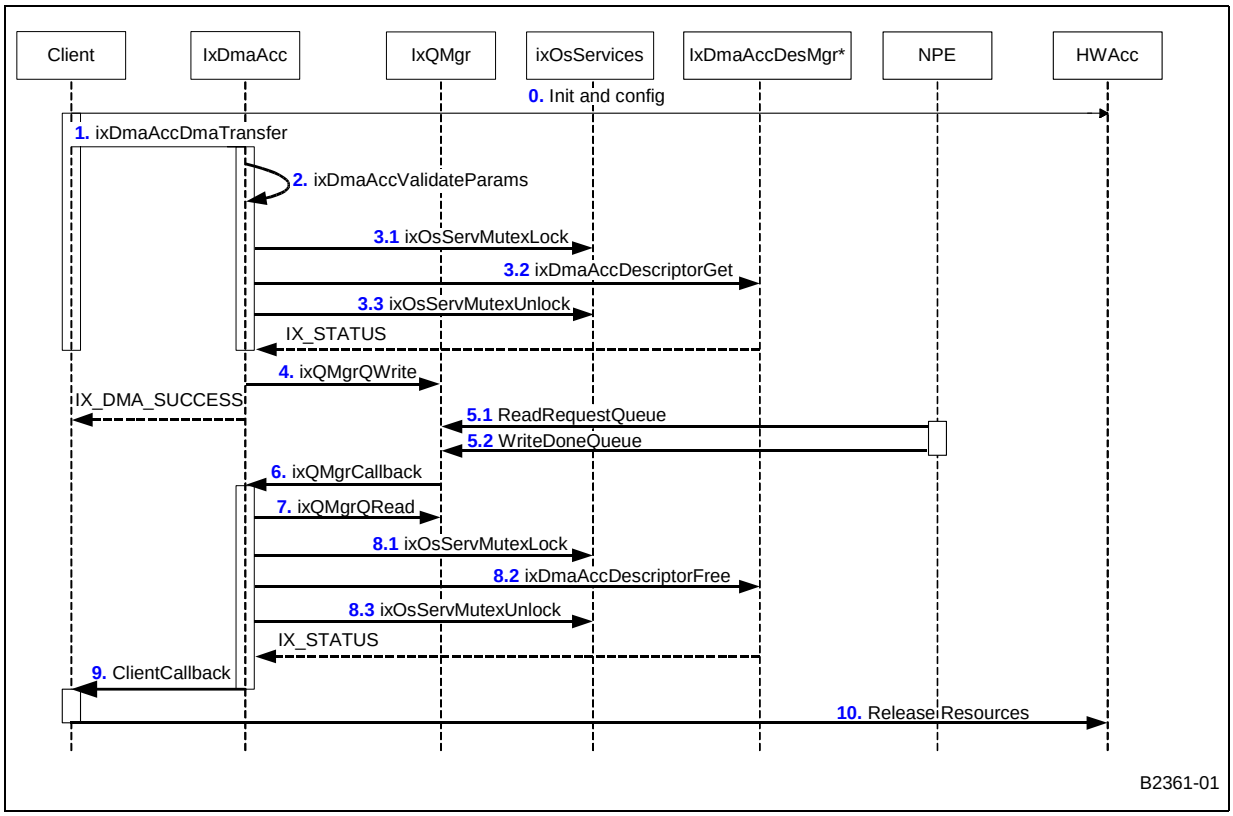
   The mutex ID will be used to access queue descriptor entry pool.

   ixDmaAccInit returns IX_DMA_SUCCESS upon completion of the DMA initialization.

8.9.2 DMA Configuration and Data Transfer

Figure 36 describes the configuration and DMA data transfer between a client and an NPE.

Figure 36. 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.

8.10 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 decoding 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 and 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 and IXC1100 control plane processors' 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.

8.11 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 and IXC1100 control plane processors error-reporting techniques, such as the OS services error-reporting mechanism.

8.12 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 v1.4's "Ethernet Access API" access-layer component.

9.1 What's New

The following changes and enhancements were made to this component in software release 1.4:

- The IX_MBUF_NEXT_PKT_IN_CHAIN_PTR mBuf field is required to be a NULL value, or it will trigger the IX_ASSERT condition and fail. IxEthAcc now uses this field internally and then returns this field to NULL before passing it back up the stack. In previous releases, this field was not used by IxEthAcc.

  Important: **Care must be taken when porting code that used a previous release of this component, since the new version checks for a NULL value in this field.**

- Some of the IxEthAcc behaviors have changed as a result of the removal of the IxFpathAcc access-layer component and it's dependencies on Ethernet related resources. Specifically, the Tx queues have been increased to hold up to 128 entries per port.
- Support for large Ethernet frames has been added, up to 16,320 bytes.
- The ixEthAccPortRxFreeReplenish() function now checks that the supplied mBuf size is at least IX_ETHACC_RX_MBUF_MIN_SIZE, in order to prevent poor performance due to excessive chaining.

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 the IXP42X product line and IXC1100 control plane processors. The IXP42X product line and IXC1100 control plane 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 the IXP42X product line and IXC1100 control plane 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

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 by 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 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 stand-alone 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 Microcode

The Ethernet NPE microcode 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 microcode performs a number of data-processing operations.

On the Ethernet receive path, the Ethernet NPE microcode 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 microcode performs priority queuing of outgoing frames and MAC statistics collection, and destination port lookup based upon destination MAC address.

It is important to note that the Ethernet NPE microcode 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 1.4 version of Ethernet NPE microcode. However, the lack of NPE-level support for these features in no way inhibits the Intel XScale core-based software from implementing them.

Communication between an Ethernet NPE and the Intel XScale core is facilitated by two mechanisms. The IxQMgr component is used to handle the data path communications between the Intel XScale core-based code and NPEs, and is described below. IxNpeMh is used to facilitate the communication of control-type messages between IxEthAcc and the NPE's. An example of a control message would be IxEthAcc instructing an NPE to update it's local MAC database.

9.3.2 Queue Manager

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.

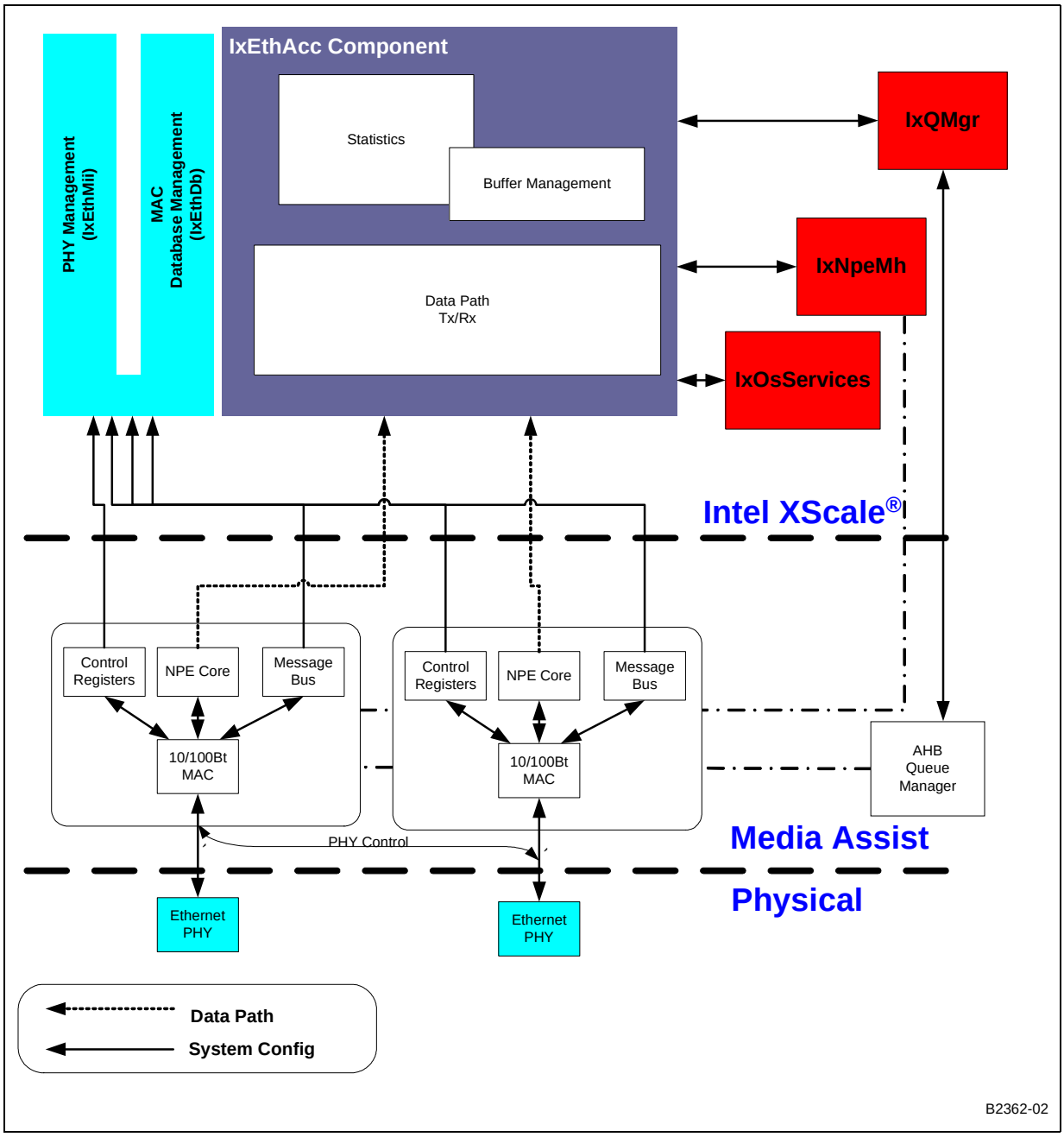
9.4 Ethernet Access Layers: Component Features

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 37. 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

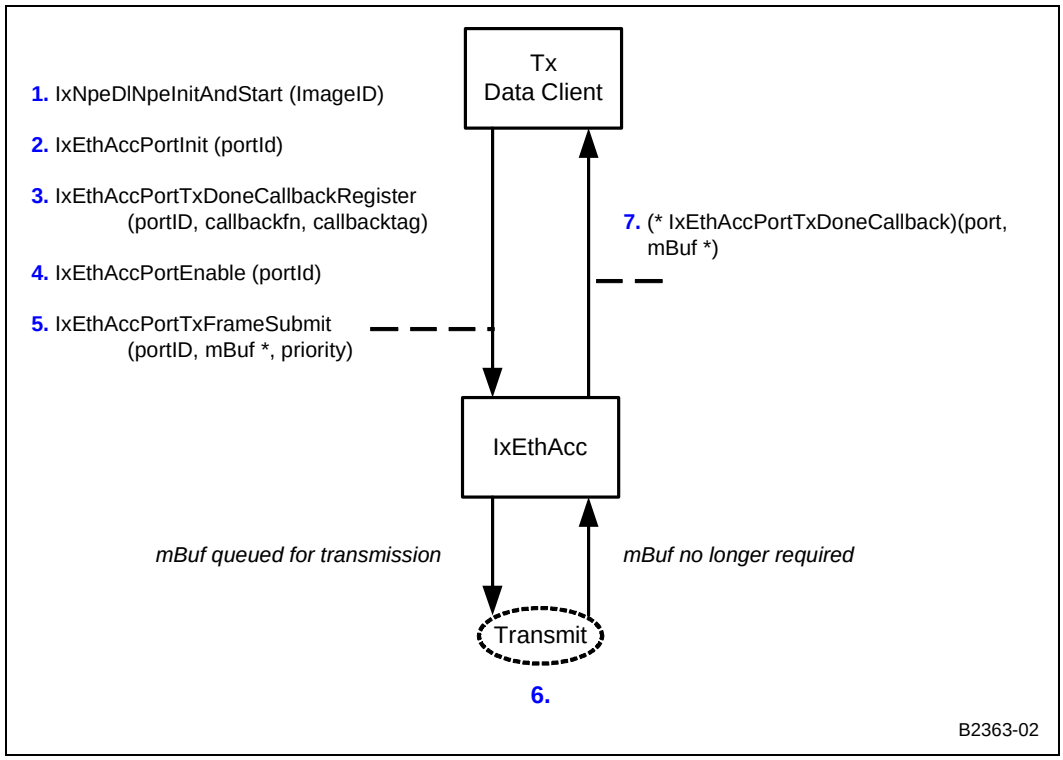
Prior to any operation being performed on a port, the appropriate microcode must be downloaded to the NPE using the IxNpeDl component.

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 mBuf 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 121.

Figure 38. 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. When transmission is complete, the mBuf is placed in the TxDone queue.
7. Frame transmission is complete when the TxDone callback function is invoked. 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 at the MAC level for failed transmit attempts.

9.5.2.2 Transmit Buffer Management and Priority

The overall queuing topology for the Ethernet transmission system is made up of the following queues:

- Software queues within IxEthAcc for buffering traffic when downstream queues are full, or for establishing priority queuing.
- IxQMgr queues for passing data to and from the NPE's. A maximum of 128 entries per port are supported for the TxEnet queues, and there is a single 128 entry queue for TxEnetDone.
- NPE microcode queues, used to hold mBuf data for transmission. There are 64 entries in the NPE microcode queue(s).

"Ethernet Transmit Frame Data Buffer Flow" on page 123 provides a visual explanation of queue management for Ethernet transmission.

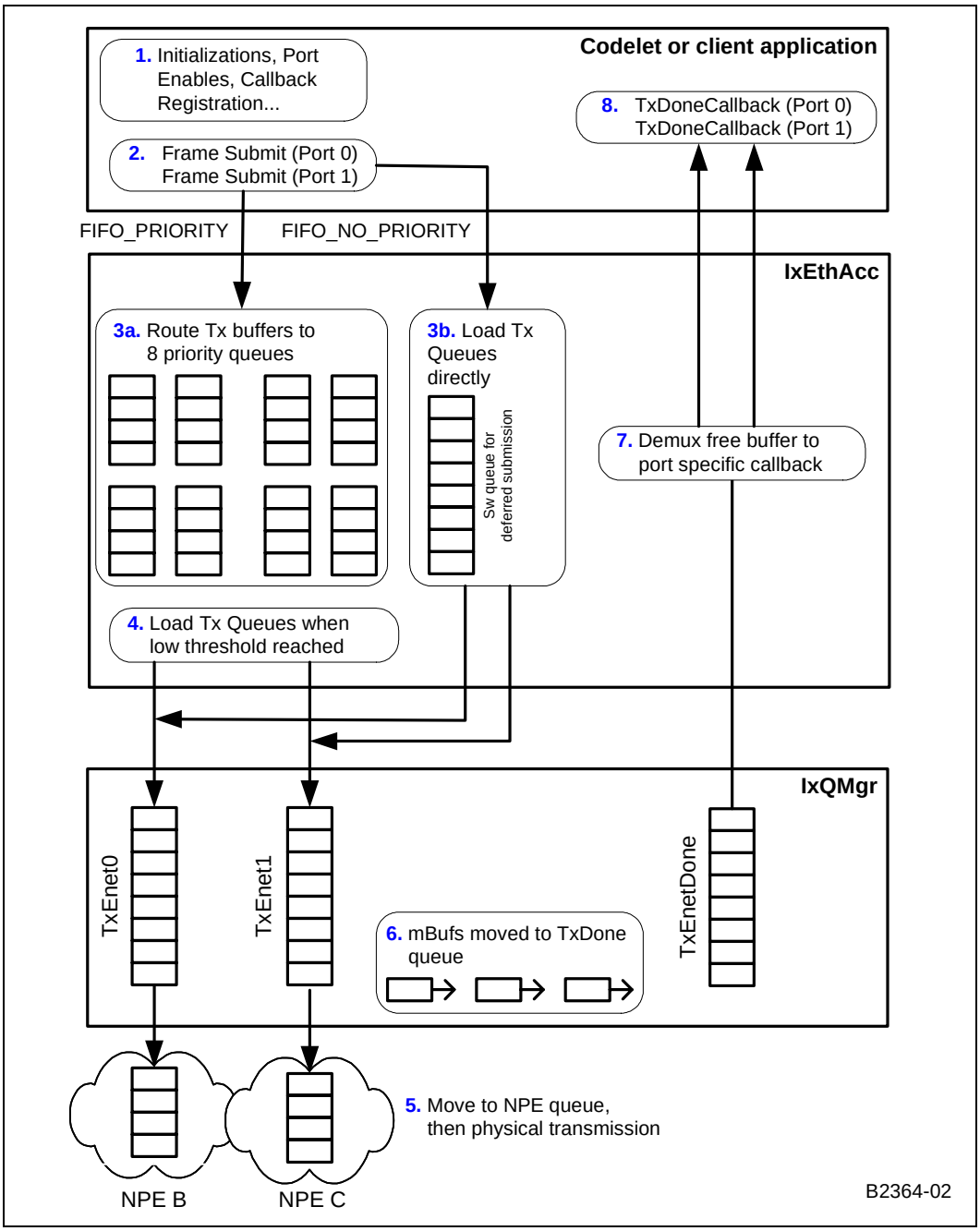
The IxQMgr queues are a maximum of 128 entries deep per port. The frame submit function must internally queue (in the IxEthAcc software) frames which are submitted in excess of a predefined limit. All internally queued buffers submitted for transmission but not queued to the hardware queues are stored in IxEthAcc software queues. If priority FIFO queuing is being used, the frames will be saved in individual per priority FIFOs.

Frames will be submitted to the port specific IxQMgr queue when a low/empty threshold is reached on the queue. From there, the buffer header is passed into the NPE queue that supports that respective port. If priority queueing is enabled, the NPE can re-order the frames internally to ensure that higher priority frames are transmitted before lower priority frames.

Once frame transmission has completed, the buffer is placed on the TxEnetDone 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( ).

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 Tx buffers to the TxDone queue and then passed to the user via the registered TxDone callback.

Figure 39. 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.

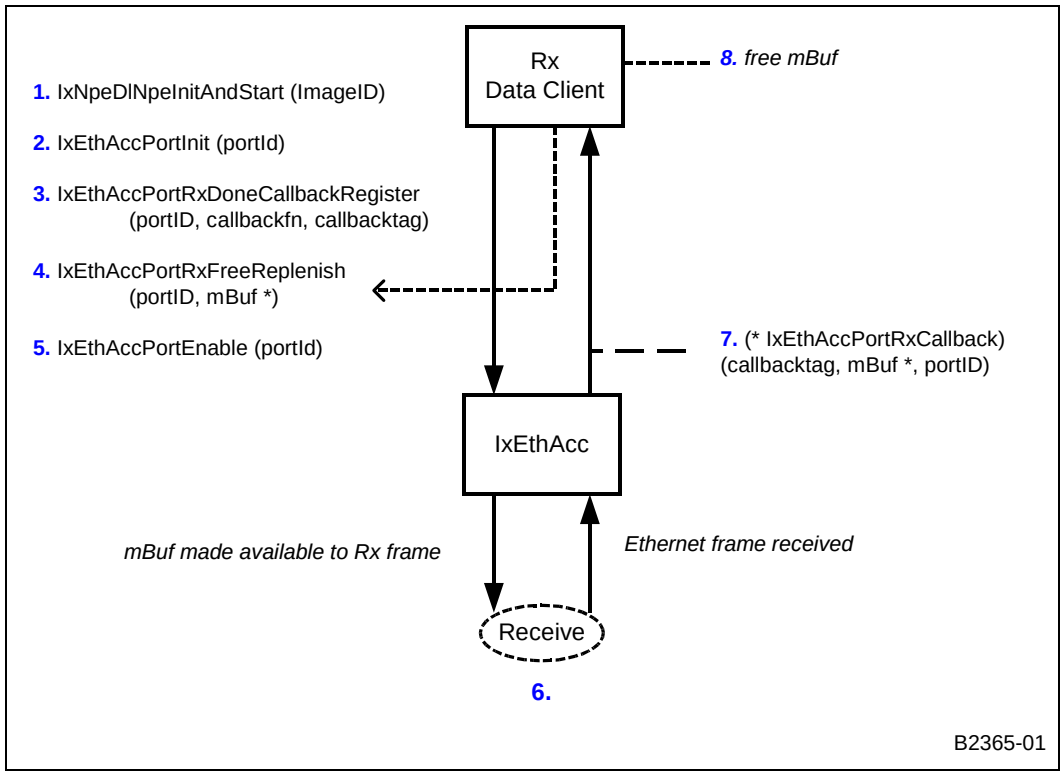
9.5.2.3 Using Chained mBufs for Transmission

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 and for large packet handling. 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 125.

Figure 40. 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 128.

Buffer Sizing

The receive data plane subcomponent must provide receive buffers to the NPEs. These mBufs should be size appropriately to ensure optimal performance of the Ethernet receive subsystem. The mBuf should contain IX_ETHACC_RX_MBUF_MIN_SIZE bytes in a single data cluster, though chained mBufs are also supported. It is expected that chained mBufs will be used to handle large frames. Receive frames may be pushed into a chained mBuf structure, but excessive chaining will have an adverse impact upon performance

For maximum performance, the mBuf size should be greater than the maximum frame size (Ethernet header, payload and FCS) + 64. Supplying smaller mBufs to the service results in mBuf chaining and degraded performances. The recommended size is 2,048 bytes, which is enough to take care of 802.3 frames, “baby jumbo” frames without chaining, and “jumbo” frames with chaining.

Buffers may not be filled up to their length. The NPE microcode will fill the mBuf fields up to the 64-byte boundary. The user has to be aware that the length of the received mBufs may be smaller than the length of the supplied mBufs.

Supplying Buffers

There are two separate free buffer IxQMgr queues allocated to providing the NPEs with receive buffers (one per port). The buffers are supplied on a per port basis via the user level interface ixEthAccPortRxFreeReplenish() function. The replenish function loads the port specific free buffer IxQMgr queue with an IX_mBuf pointer. The replenish function can provide checking to ensure that the mBuf is at least as large as IX_ETHACC_RX_MBUF_MIN_SIZE. 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. Frames greater in size than the size of the mBuf provided by the replenish function will trigger chaining.

Note: The ixEthAccPortRxFreeReplenish() function can receive chained mBufs, which the NPEs will be able to unchain as needed. This method may offer a performance improvement for some usage scenarios.

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. Errored frames (FCS errors, size overrun) are not passed to the user.

Freeing Buffers

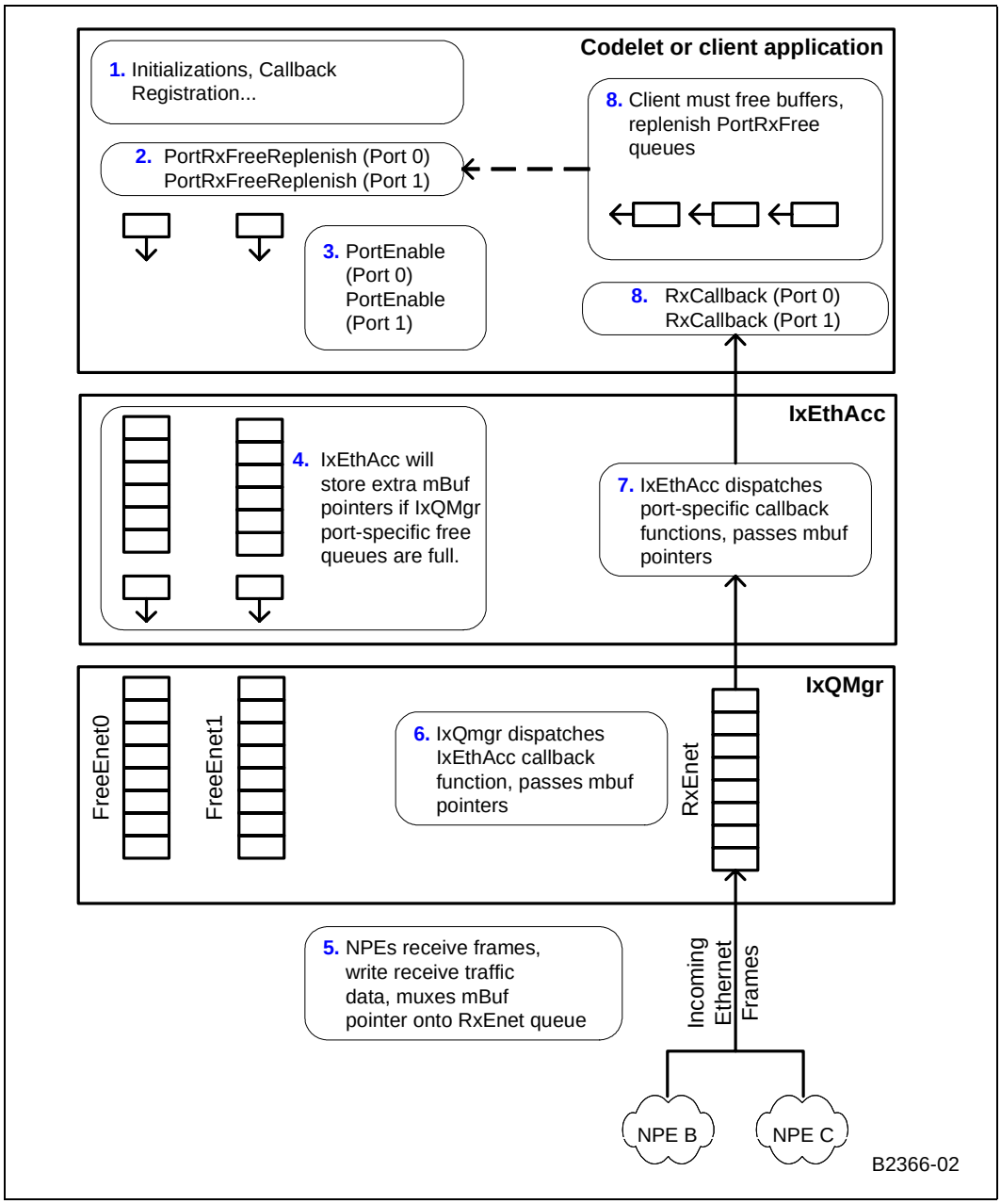
Once this service calls the callback with the receive mBuf, “ownership” of the buffer is transferred to the user of the access component (i.e., the access component will not free the buffer). Once IxEthAcc calls the registered user-level receive callback, the receive mBuf “ownership” is transferred to the user of the access component. IxEthAcc will not free the buffer. Should a chain of mBufs be received, the head of the buffer chain is passed to the rx callback.

Buffers can also be freed by disabling the port, using the IxEthAccPortDisable() function. This has the result of returning all Rx buffers to the Rx registered callback, which may then de-allocate the mBufs to free memory.

Recycling Buffers

Buffers received (chained or unchained) on the Rx path can be used without modification in the Tx path. Rx and TxEnetDone buffers (chained or unchained) should have the length of each cluster reset to the cluster original size before re-using it in the ixEthAccPortRxFreeReplenish() function.

Figure 41. Ethernet Receive Plane Data Buffer Flow



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.

9.5.4 Data-Plane Endianness

All data structures provided to the IxEthAcc components, such as mBuf headers or statistic structures, are defined by the target system byte order. No changes to data structures are required in order to use the access component data path interfaces as IxEthAcc takes care of any conversion that is required in order to communicate to the NPEs. The data pointed to by the mBuf (the mBuf payload) 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 Maximum Ethernet Frame Size

The maximum supported Ethernet frame size is 16,320 bytes. This value is set on a per-port basis using the IxEthDB API.

9.5.6 External Memory Requirements

Locations in Physical Memory

The IxEthAcc component addressing space for physical memory is limited to 28 bits. Therefore mBuf headers should be located in the first 256 Mbytes of physical memory.

The buffer descriptor format supported is the *IX_MBUF*, which is defined in the API for the IXP400 software. It is loosely based on the *mBuf* which is described in detail in *TCP/IP Illustrated, Volume 2*. The Ethernet NPE microcode expects that all such structures (*i.e. IX_MBUF* structures) will be word-aligned. The size of the mBuf data clusters to be used on the Ethernet receive path must be at least 1,536 bytes, and preferably 2K bytes or more (which, in turn, implies that all Rx-path mbufs must have external data clusters). For applications utilizing jumbo frames (which are larger than 2K bytes) the preferable buffer size is the one reflecting maximum frame size, so no buffers have to be chained in order to receive the whole frame. Although buffer chaining is possible it requires more processing from NPE and the Intel XScale core to receive frame. Buffer chaining occurs whenever buffer size is smaller than the frame size. Smaller buffers can be preferable for applications where the memory requirements are the main concern.

On the transmit side, the NPE microcode is capable of handling chained mbufs (via the *IX_MBUF_NEXT_BUFFER_IN_PKT_PTR* 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. Any mbufs used on the Ethernet transmit path may have either “internal” or “external” data clusters. The *IX_MBUF_NEXT_PKT_IN_CHAIN_PTR* field, which facilitates the linking of multiple packets together into “queues” is not used by the Ethernet NPE microcode; 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 42. mBuf Fields Written by Intel XScale® Core (left) and NPE (right) on Ethernet Rx Path

	0	1	2	3
0	IX_MBUF_NEXT_BUFFER_IN_PKT_PTR(1)			
4	IX_MBUF_NEXT_PKT_IN_CHAIN_PTR(2)			
8	IX_MBUF_MDATA			
12	IX_MBUF_MLEN(3)			
16	IX_MBUF_TYPE	IX_MBUF_FLAGS		(Reserved)
20	(Reserved)			
24	IX_MBUF_PKT_LEN(4)			

	0	1	2	3
0	IX_MBUF_NEXT_BUFFER_IN_PKT_PTR			
4	IX_MBUF_NEXT_PKT_IN_CHAIN_PTR			
8	IX_MBUF_MDATA			
12	IX_MBUF_MLEN(3)			
16	IX_MBUF_TYPE	IX_MBUF_FLAGS(5)		(Reserved)
20	(Reserved)			
24	IX_MBUF_PKT_LEN(5)			

Notes:

1. On the Ethernet Rx path free buffers can be chained and there is no limit to the number of mbufs in the chain.
2. All IXP400 software is designed under the assumption that packet chaining will not be used. Thus, the value of the *IX_MBUF_NEXT_PKT_IN_CHAIN_PTR* must always be NULL.
3. It is required that the Intel XScale core client explicitly set the *IX_MBUF_MLEN* field, the Ethernet NPE microcode uses it to determine the size of data cluster to which the *IX_MBUF_MDATA* field points. The size must be at least 1,536 bytes (and preferably 2K or more) in length.
4. In the first buffer in the buffer chain, the Intel XScale core sets *IX_MBUF_PKT_LEN* to the length of the entire frame received (the sum of the *IX_MBUF_MLEN* fields of each buffer in the chain).
5. The NPE writes IX_MBUF_NEXT_BUFFER_IN_PKT_PTR field for each mBuf in the chain, however IX_MBUF_NEXT_PKT_IN_CHAIN_PTR is set only in the last mBuf in the chain or first mBuf if only one mBuf was used to store frame. This field encodes Rx flags and frame length accordingly to the following table.

Table 18. Contents of the IX_MBUF_NEXT_PKT_IN_CHAIN_PTR Field in the Last mBuf in a Chain

Bit	Description
26	Set to 1 if frame type field contains 0x800. Otherwise, 0.
25	Set to 1 if frame destination MAC address is multicast
24	Set to 1 if frame destination MAC address is broadcast
15:0	Frame length, in bytes.

Figure 43. mBuf Fields Written by Intel XScale® Core (left) and NPE (right) on Ethernet Tx Path

	0	1	2	3
0	IX_MBUF_NEXT_BUFFER_IN_PKT_PTR(1)			
4	IX_MBUF_NEXT_PKT_IN_CHAIN_PTR(2)			
8	IX_MBUF_MDATA			
12	IX_MBUF_MLEN			
16	IX_MBUF_TYPE	IX_MBUF_FLAGS		(Reserved)
20	(Reserved)			
24	IX_MBUF_PKT_LEN(3)			

	0	1	2	3
0	IX_MBUF_NEXT_BUFFER_IN_PKT_PTR			
4	IX_MBUF_NEXT_PKT_IN_CHAIN_PTR			
8	IX_MBUF_MDATA			
12	IX_MBUF_MLEN			
16	IX_MBUF_TYPE	IX_MBUF_FLAGS		(Reserved)
20	(Reserved)			
24	IX_MBUF_PKT_LEN			

Notes:

1. Unlike the Ethernet Rx path, it is possible for buffers to be chained on the Ethernet Tx path.
2. All IXP400 software is designed under the assumption that packet chaining will not be used. Thus, the value of the *IX_MBUF_NEXT_PKT_IN_CHAIN_PTR* 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 *IX_MBUF_PKT_LEN* field set correctly (i.e., to the length of the entire frame.

9.6 Control Path

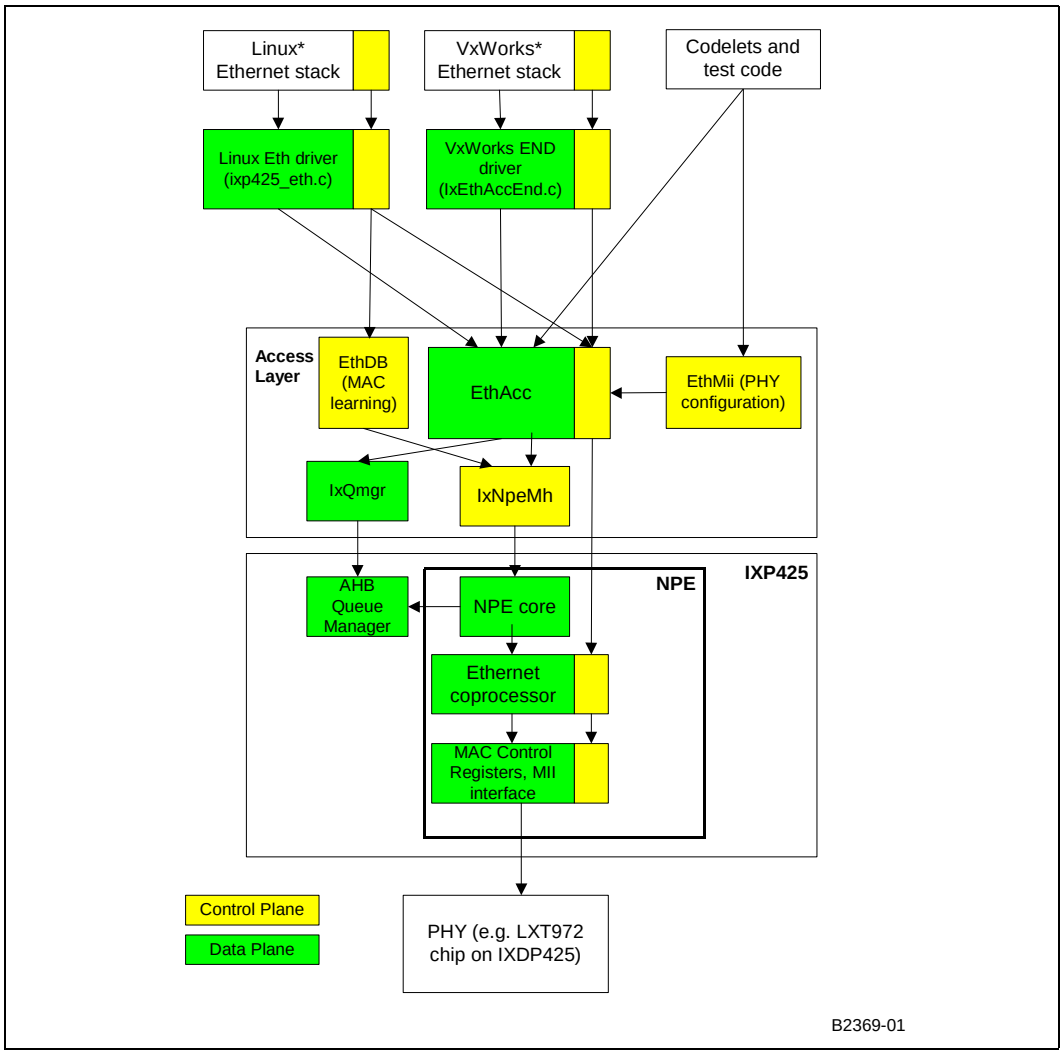
The main control path functions are performed by two external components: IxEthMii and IxEthDB.

IxEthMii is used primarily to manipulate a minimum number of necessary configuration registers on Ethernet PHYs supported on the IXDP425 / IXCDP1100 platform 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 that runs within the context of the Intel XScale core. The IxEthDB component handles the database structure, maintenance, searching, and aging and has an API for the provisioning of dynamic and static addresses. This database populates filtering entries on the NPEs and also retrieves learning entries from the NPEs. An API is provided to the access layer.

The relationship between IxEthAcc, IxEthDB, and IxEthMii is demonstrated in Figure 44.

Figure 44. 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 interfaces for the PHY's connected to the NPEs on the Intel® IXDP425 / IXCDP1100 Development Platform. These functions do not support reading PHY registers of devices connected on the PCI 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 NPEs provide a function whereby MAC address source learning is performed on received (ingress) Ethernet frames. If source learning is enabled, the source MAC address 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. More information on the Learning/Filtering Database is available in "Access-Layer Components: Ethernet Database (IxEthDB) API" on page 137.

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. The MAC uses a mask and a multicast filter address. Packets where (dstMacAddress & mask) == (mCastfilter & mask) are forwarded to the NPE.

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 19 and Table 20. 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 19. 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 && !FCS_ERR)
LearnedEntryDiscards	Received frame dropped due to MAC destination address filtering.
UnderflowEntryDiscards	Received frame dropped due to replenishing starvation.

Table 20. 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

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

This chapter describes the Intel® IXP400 Software v1.4'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

The following changes and enhancements were made to this component in software release 1.4:

- A new function, ixEthDBFilteringPortMaximumFrameSizeSet(), was added to provide support for larger ethernet frames.

10.3 MAC Database Theory

There are two major elements involved in the IxEthDB subsystem: a software database that resides executes on the Intel XScale core of the processor, and a learning tree and filtering capability for each of the NPEs capable of Ethernet co-processing. Although it is possible to create static entries in the NPE learning tree via the IxEthDB API, most information in the learning tree is created dynamically via the MAC address learning process. The Intel XScale core-based database aggregates all of the NPE learning and filtering entries and can also push learning or filtering entries down to the NPEs.

This document refers to the NPE-based learning trees and filtering capabilities by referring to them as the "NPE database."

10.3.1 Address Learning and Filtering

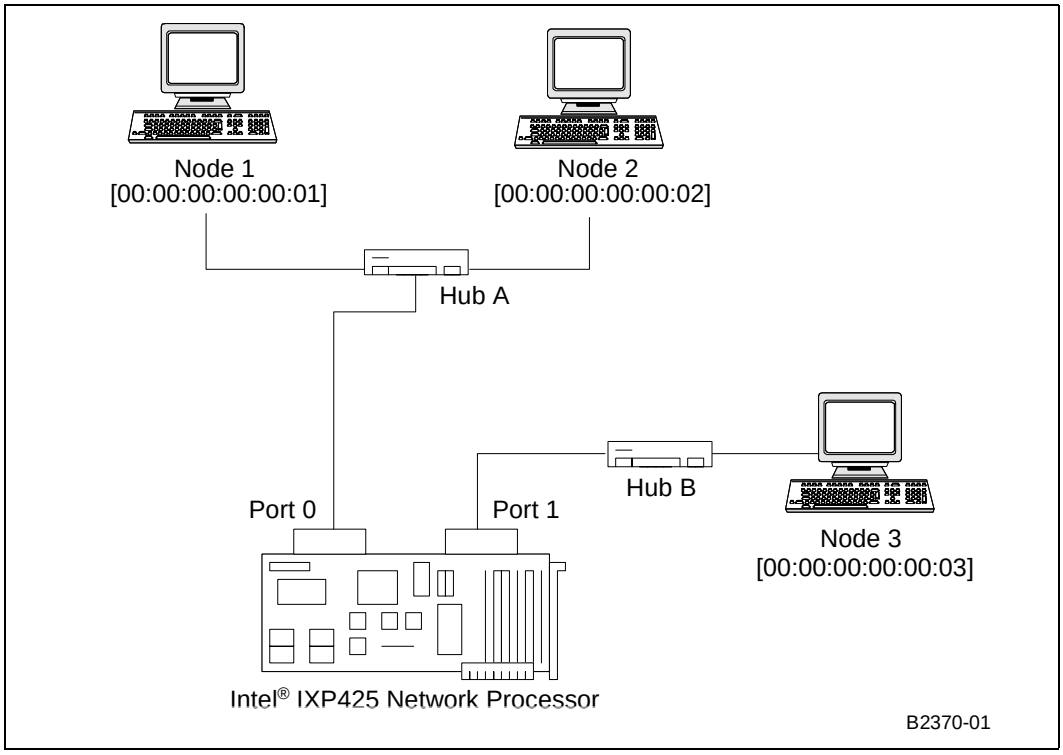
The NPEs provide a function whereby MAC address-source learning is performed on received (ingress) Ethernet frames. If source learning is enabled, the source MAC addresses are automatically populated in the Intel XScale core database. Each entry in the database is primarily composed of a MAC address and a logical port number. Whenever the bridge receives a frame, the frame is parsed to determine the destination MAC address, and the database is consulted to determine the port to which the frame should be forwarded.

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**.

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 (**filtering**). 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. Otherwise the frame is forwarded from the NPE to the IxEthAcc component.

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. Filtering can also be done according to some characteristics of a packet received on a port. For example, a maximum tolerable frame size can be set for a port. This means that if a port receives a frame that is larger than the maximum frame size, that frame will be filtered. An example of this type of filtering can be found on "Filtering Example Based Upon Port Characteristics" on page 144.

Figure 45. Example Network Diagram for MAC Address Learning and Filtering



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

10.3.2 MAC Database in Other Usage Models

If a terminal (source of Ethernet traffic on the network) is moved from one NPE port to another, IxEthDB is responsible for ensuring the consistency of the Intel XScale core and NPE learning/filtering databases. The Intel XScale core database and NPE learning trees are 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. This behavior is described as "**migrating**."

There are some situations in which the NPE databases may not have learned the proper destination port for a received packet. The NPEs will then pass the packet to the IxEthAcc component to allow it to search the Intel XScale core-based database for the proper destination port. If the system is operating in a **bridging or switching** fashion, the Intel XScale core-based database will know the appropriate port to send the packet out on. If the Intel XScale core database does not know the appropriate destination port, the receive callback function will set the port ID field to a value greater than IX_ETH_DB_NUMBER_OF_PORTS, indicating that the destination port of this packet is unknown. The client may then initiate **flooding** to forward the packet on all ports in the hopes that a node somewhere on the network will respond.

The IxEthDB 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.

10.3.3 MAC Database General Characteristics

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, aging, filtering and migrating) 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 or migrating 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-based 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 Intel XScale core 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 ixp400_xscale_sw/src/ethDB/include/IxEthDB_p.h.

A high-level overview of the system is provided in “Ethernet Receive Frame Database Overview” on page 141.

Figure 46. Ethernet Receive Frame Database Overview

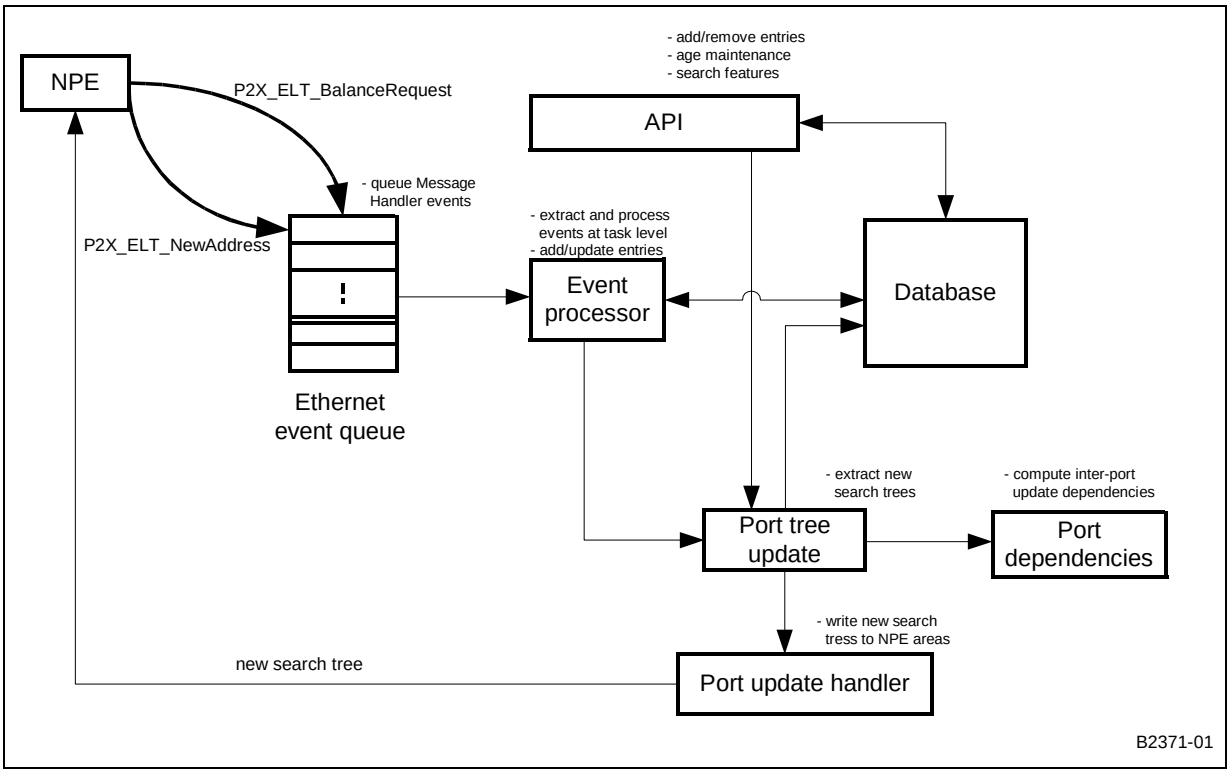


Figure 46 exemplifies the main event flows from NPEs and API, which may trigger changes to the database, which in turn determines tree updates that provides the NPEs with updated binary search trees. It should be noted that the design allows for ports that are not specifically Ethernet NPEs to be integrated into the same access and update scheme (for example, PCI).

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

IxEthDB 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 received by the NPE.
- Add entries signaled by NPE balancing requests 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 IxEthDB API

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 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.

Special consideration must be made with one specific NPE microcode image. For more information, see “Access-Layer Components: Ethernet Database (IxEthDB) API” on page 137. The image IX_NPEDL_NPEIMAGE_NPEC_CRYPTO_AES_CCM_ETH does not support NPE learning. However, NPE filtering will work if enabled as described above.

10.4.2 Promiscuous-Mode Requirement

MAC address-frame filtering, based on learning trees, is usable only when a port operates in *non-promiscuous* mode. Otherwise the frames will be filtered 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 address is the broadcast address or if the destination is a multicast address subscribed to by the port

- The destination address has not been learned on this port.
- The frame is a broadcast/multicast frame.

Configuration of promiscuous mode and multi-cast configuration is described in the section for IxEthAcc, "MAC Filtering" on page 133.

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 Maximum Ethernet Frame Size

The API provides the ability to set the maximum size of Ethernet frames supported per port, using the ixEthDBFilteringPortMaximumFrameSizeSet() function. When a maximum frame size value is set for a port, there are two effects:

- Any incoming (Rx) frames on the specified port larger than the set value will be dropped.
- Any oversized Rx frames coming in on a *different* port but whose destination MAC is through the size-restricted port will be dropped.

The maximum supported value is 16,320 bytes. For purposes of clarification, the number of bytes making up the Maximum Frame Size value is the Ethernet MSDU (Media Service Data Unit) and defined as the sum of the sizes of:

- the Ethernet header: dest MAC + src MAC + optional VLAN Tag + length/type
- the Ethernet payload
- the Ethernet trailer (FCS), if not stripped out by IxEthAccPortRxFrameFcsDisable().

10.4.6 Filtering Example Based Upon Port Characteristics

Usage Example 1

On a system with two ports (0 and 1), execute:

```
ixEthDBFilteringPortMaximumFrameSizeSet(1, 3000)
```

The following behaviors will result:

- The NPE on Port 1 will filter all received frames larger than 3,000 bytes.
- The NPE on Port 0 will filter the received frames bigger than 3,000 where the destination MAC address matches an address learned on port 1.

Usage Example 2

On a system with three ports (0, 1, 2), execute:

```
ixEthDBFilteringPortMaximumFrameSizeSet(0, 9014);
ixEthDBFilteringPortMaximumFrameSizeSet(1, 9014);
ixEthDBFilteringPortMaximumFrameSizeSet(2, 1514).
```

The NPE on Port 0 will:

- Filter all Rx frames over 9,014 bytes.

A frame of 3,000 bytes is received on Port 0. The NPE will determine the destination port based on learned MAC address, and:

- If the port is unknown, forward the frame to IxEthAcc
- If the port is 0 (itself), filter the frame
- If the port is 1, pass the frames to IxEthAcc
- If the port is 2, discard the oversized frame

The NPE on Port 1 will:

- Filter all Rx frames over 9,014 bytes.

A frame of 3,000 bytes is received on Port 1. The NPE will determine the destination port based on learned MAC address, and:

- If the port is unknown, forward the frame to IxEthAcc
- If the port is 0, pass the frame to IxEthAcc
- If the port is 1 (itself), filter the frame
- If the port is 2, discard the oversized frames

10.4.7 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.

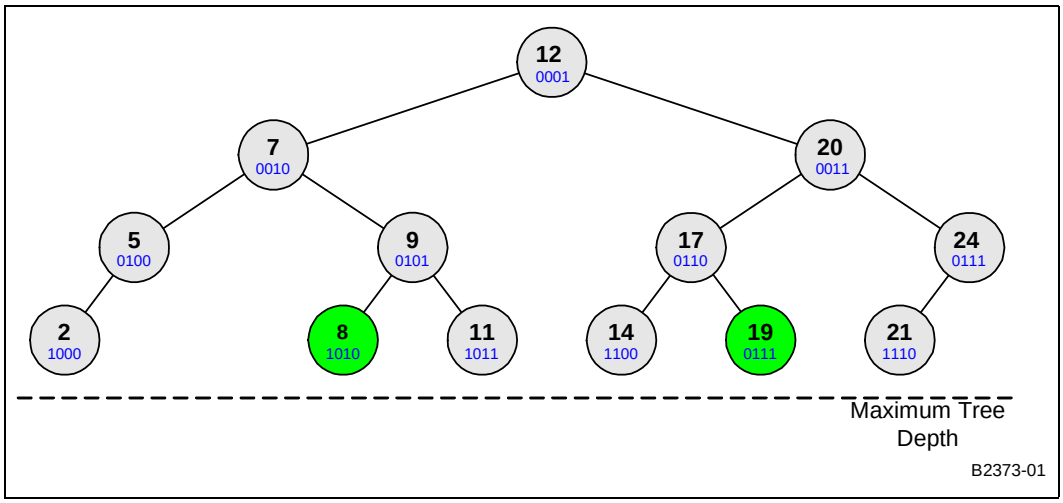
10.4.8 Database Maintenance

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.

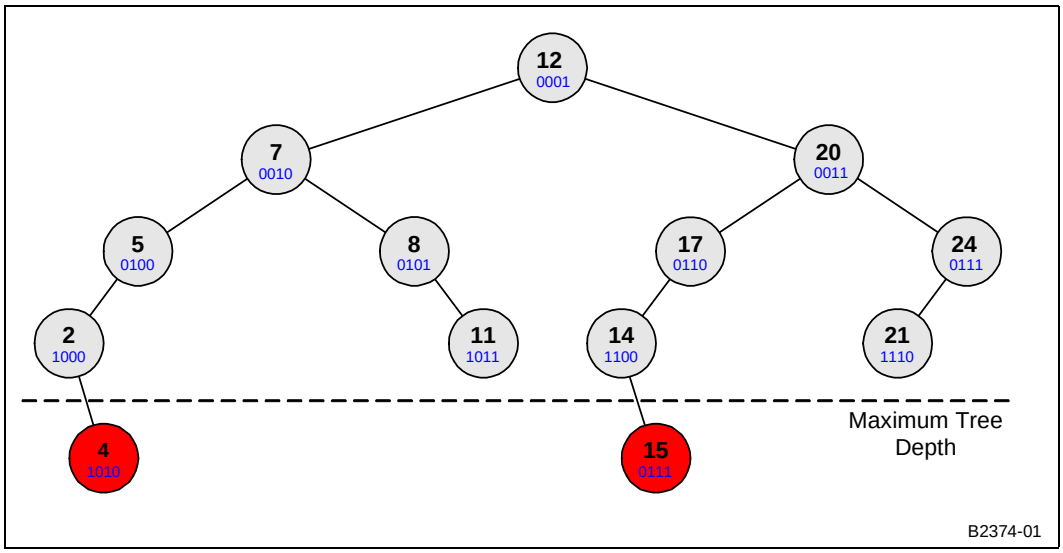
As new MAC addresses are inserted into the filtering database via the learning process, the binary tree structure used to maintain this database tends to become unbalanced.

Figure 47. 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 48. 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.

10.4.9 Database Elements

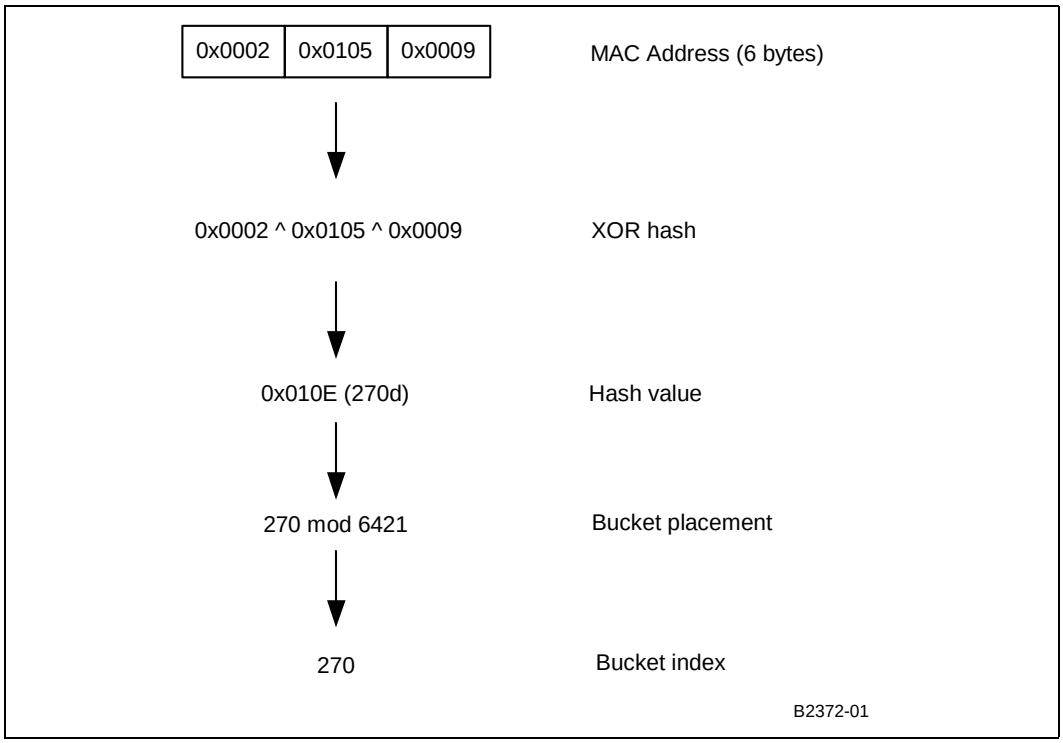
MAC addresses are unique keys in the database. Therefore, no records can share the same address. Adding an existent address on a different port, either using the API or when detected by an NPE, will automatically update the existing record with the new port information. In real terms, this corresponds 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.

10.5 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 148 describes the process of hashing a MAC address entry, assuming 6,421 buckets are used.

Figure 49. 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 v1.4's "Ethernet PHY API" access-layer component.

11.1 What's New

There are no changes or enhancements to this component in software release 1.4.

11.2 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 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.

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. IxEthMii interacts with the MII interfaces for the PHY's connected to the NPEs on the Intel® IXDP425 / IXCDP1100 Development Platform. These functions do not support reading PHY registers of devices connected on the PCI interface. 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 in the following path:

```
ixp400_xscale_sw/src/ethMii/IxEthMii_p.h
```

Table 21. 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 v1.4's "Feature Control API" access-layer component.

IxFeatureCtrl is a component that detects the capabilities of the Intel® IXP42X Product Line of Network Processors and IXC1100 Control Plane Processor processors 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

There are no changes or enhancements to this component in software release 1.4.

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' coprocessor 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 22. Product ID Values

Bits	Description
31:28	Reserved. Value: 0x6
27:24	Reserved. Value: 0x9
23:20	Reserved. Value: 0x0
19:16	Reserved. Value: 0x5
15:12	Reserved. Value: 0x4
11:4	Maximum Achievable Intel 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 coprocessor, NPEs, etc.).

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

Table 23. 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

† 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 23. Feature Control Register Values (Sheet 2 of 2)

Bits	Description
5 †	HDLC Coprocessor
4 †	DES Coprocessor
3 †	AES Coprocessor
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 1.4, 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 microcode to disabled or unavailable NPEs.

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

13.1 What's New

The IxFpathAcc component has been depreciated in software release 1.4. If Fast Path functionality is needed, software release 1.3 is the most current version to include this feature.

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

This chapter describes the Intel® IXP400 Software v1.4's "HSS-Access API" access-layer component.

14.1 What's New

The following changes and enhancements were made to this component in software release 1.4:

- Support was added for 56-Kbps Raw and 56-Kbps HDLC packetized data.
- Changes have been made to the `IxHssAccPktPortConnect()` function to accommodate 56-Kbps packetized data. These changes mean that *the function is not backwards compatible* with previous releases.
- A new data structure for supporting 56-Kbps HDLC configuration parameters, `IxHssAccHdlcMode`, was added.
- Support has been added for CAS bit polarity ('0'/'1') configuration per termination point for 56-Kbps HDLC. Further, the CAS bit can be configured to be in the LSB or MSB position, per termination point
- Support has been added for complete bit inversion per HDLC or Raw termination point. The inversion would occur post-HDLC co-processor processing on the transmit direction and pre-HDLC co-processing on the receive direction.

14.2 Overview

The IxHssAcc component provides client applications with driver-level access to the High-Speed Serial (HSS) and High-Level Data Link Control (HDLC) co-processors available on NPE A. This API and its supporting NPE-based hardware acceleration enable the Intel® IXP42X Product Line of Network Processors to support packetized or channelized TDM data communications.

This chapter provides the details of how to use IxHssAcc to:

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

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 on the IXP42X product line of network processors.
- Configuration of the HSS and HDLC coprocessors residing on NPE A.
- Support for TDM signals up to a rate of 8.192 Mbps (Quad E1/T1) on an HSS port.

Channelized Service

- Support a single Channelized client per HSS port. For each Channelized client:
  - — Support up to 32 channels, where each channel is composed of one time slot
  - — Each channel is independently configurable for 56-Kbps or 64-Kbps mode

    For 56-Kbps mode:
    - Configurable CAS bit position - least significant or most significant bit position. Configurable on a per-port basis only.
    - Configurable CAS bit polarity for transmitted data

Packetized Service

- Support a single Packetized client (termination point) per T1/E1 trunk, up to maximum of four per HSS port. For each Packetized client:
  - — Configurable for RAW or HDLC mode
  - — Configurable bit inversion - all data inverted immediately upon reception from and transmission to the trunk
  - — Configurable for 56 Kbps or 64 Kbps (specifically, 65,532 bytes) mode. Maximum recommended size of received HDLC packets is 16 Kbytes. For 56-Kbps mode:
  - — Configurable CAS bit position - least significant or most significant bit position
  - — CAS bit always discarded for data received from trunk
  - — CAS bit insertion for data transmitted to trunk
  - — Configurable CAS bit polarity for transmitted data

14.3 IxHssAcc API Overview

The IxHssAcc API is an access layer component that provides high-speed serial and packetized or channelized data services to a client application. This section describes the overall architecture of the API. Subsequent sections describe the component parts of the API in more detail and describe usage models for packetized and channelized data.

14.3.1 IxHssAcc Interfaces

The *client* application code executes on the Intel XScale core and utilizes the services provided by IxHssAcc. In this software release, the IxHssAccCodelet is provided as an example of client software. As previously described, the *IxHssAcc API* is the interface between the client application code and the underlying hardware services and interfaces on the Intel® IXP42X Product Line of Network Processors.

IxHssAcc presents two “services” to the client application. The *Channelized Service* presents the client with raw serial data streams retrieved from the HSS port, while the *Packetized Service* provides packet payload data that has been optionally processed according to the HDLC protocol.

IxQMgr is another access-layer component that interfaces to the hardware-based *AHB Queue Manager* (AQM). The AQM is SRAM memory used to store pointers to data in SDRAM memory, which is accessible by both the Intel XScale core and the NPEs. These items are the mechanism by which data is transferred between IxHssAcc and the NPE. Queues are handled in a different manner depending on whether packetized or channelized data services are being utilized. The queue behavior is described in subsequent sections of this chapter.

IxNpeMh is used to allow the IxHssAcc API to communicate to the NPE coprocessors described below. *IxNpeDl* is the mechanism used to download and initialize the NPE microcode.

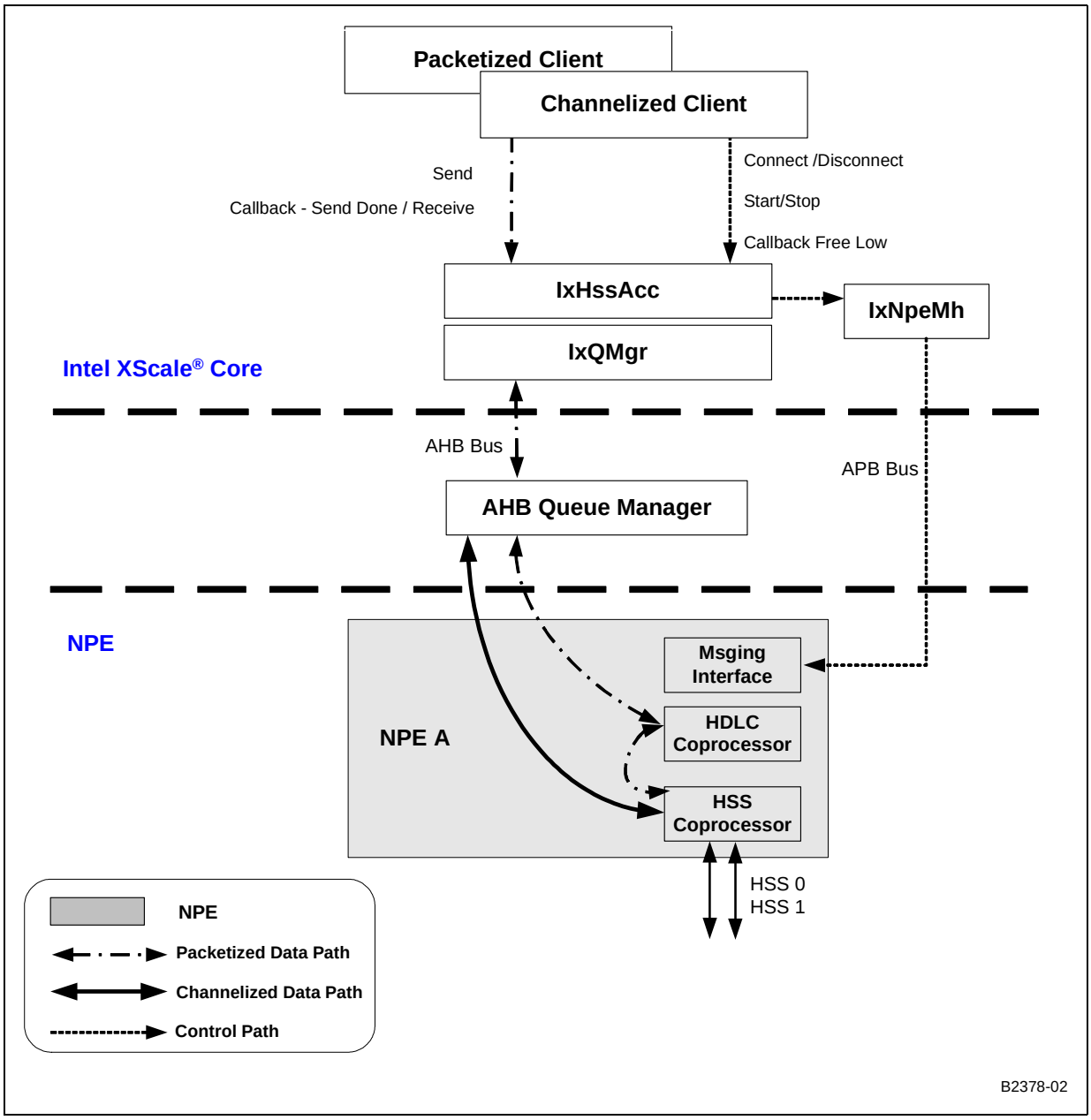
The NPE provides hardware acceleration, protocol handling, and drives the physical interface to the High-Speed Serial ports. NPE-A is the specific NPE that contains an HSS coprocessor and an HDLC coprocessor utilized by this API.

14.3.2 Basic API Flow

An overview of the data and control flow for IxHssAcc is shown in Figure 50.

The client initializes and configures HSS using IxHssAcc to configure the HSS port signalling to match the connected hardware PHY’s or framers. The HSS co-processor on NPE-A drives the HSS physical interfaces and handles the sending or receiving of the serial TDM data. Data received on ports configured for channelized data will be sent up the stack from the HSS co-processor. Received Packetized data — with the HDLC option turned on — will be passed to HDLC coprocessor as appropriate. The IxHssAcc API uses callback functions and data buffers provided by the client to exchange NPE-to-Intel XScale core data for transmitting or receiving with the help of the IxQMgr API.

Figure 50. HSS/HDLC Access Overview



14.3.3 HSS and HDLC Theory and Coprocessor Operation

The HSS coprocessor enables the IXP42X product line 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 consists of 8 bits (1 byte) which contains the data and an indicator of the time slot's location within the frame.

The maximum frame size is 1,024 bits and the maximum frame pulse offset is 1,023 bit. The line clock speed can be set using the API to one of the following values to support various E1, T1 or aggregated serial (MVIP) specifications:

- 512 KHz
- 1.536 MHz
- 1.544 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.

HSS Output Clock Jitter and Error Characterization

The high-speed serial (HSS) port on the IXP425 and IXP421 network processors can be configured to generate an output clock on the HSS_TXCLK pin. This output clock, however, is not as accurate as using an external oscillator. If the system is intended to clock a framer, DAA, or other device with a sensitive input PLL, an external clock should be used.

Clock signalling is defined in the file IxHssAccCommon.c. The following tables describe the error and jitter characteristics of signals based upon the values established in software release 1.4.

Table 24. HSS Tx Clock Output frequencies and PPM Error

HSS Tx Freq.	Min. Freq. (Mhz)	Avg. Freq. (Mhz)	Max. Freq. (Mhz)	Avg. Freq. Error (PPM)
512 KHz	0.508855	0.512031	0.512769	-60.0096
1.536 MHz	1.515	1.536	1.55023	-60.0096
1.544 MHz	1.515	1.5439	1.55023	+60.0024
2.048 MHz	2.01998	2.0481	2.08313	-60.0096
4.096 MHz	3.92118	4.0962	4.16625	-60.0096
8.192 MHz	7.4066667	8.1925	8.3325	-60.0096

Note: Characterization data of the HSS TX clock output frequency data was determined by silicon simulation. PPM parts per million error rate is calculated using average output frequency vs. ideal frequency.

Table 25. HSS TX Clock Output Frequencies and Associated Jitter Characterization

HSS Tx Freq.	Pj Max (ns)	Cj Max (ns)	Aj Max (ns)
512 KHz	12.189	15	18.283
1.536 MHz	9.063	15	86.102
1.544 MHz	12.359	15	210.099
2.048 MHz	-8.204	15	118.957
4.096 MHz	10.9	15	190.742
8.192 MHz	12.951	15	226.634

Table 26. Jitter Definitions

Jitter Type	Jitter Definition
Period Jitter (Pj)	$Pj_{(i)} = Period_{(i)} - Period_{average}$
Cycle to Cycle Jitter (Cj)	$Cj_{(i)} = Pj_{(i+1)} - Pj_{(i)}$
Wander or Accumulated Jitter (Aj)	$Aj_{(i)} = \sum_{i}^{'} Pj$

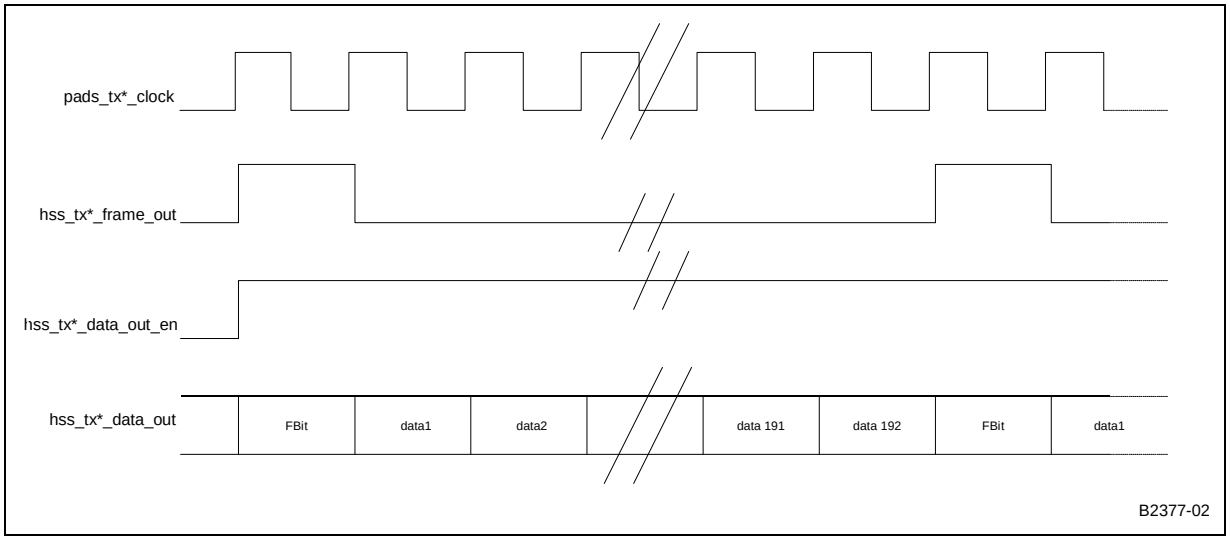
Table 27. HSS Frame Output Characterization

HSS Tx Freq.	Frame Size (Bits)	Actual Frame Length (µs)	Frame Length Error (PPM)
512 KHz	32	62.496249	-60.0096
1.536 MHz	96	62.496249	60.016
1.544 MHz	193	125.007499	60.0024
2.048 MHz	256	124.9925	-60.0096
4.096 MHz	512	62.496	-60.0096
8.192 MHz	1024	62.49624	-60.0096

Note: PPM frame length error is calculated from ideal frame frequency.

Figure 51 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 51. T1 Tx Signal Format



The time slots within a stream can be configured as packetized (raw or HDLC, 64 Kbps, and 56 Kbps), channelized voice64K, or channelized voice56K or left unassigned. “Voice” slots are those that will be sent to the channelized services. For more details, see “HSS Port Initialization Details” on page 165.

For packetized 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 and can also provide “raw” packets, those which do not require HDLC processing, to the client. The HDLC coprocessor can support up to four packetized services per HSS port.

The following HDLC parameters are programmable:

- The pattern to be transmitted when a HDLC port is idle.
- The HDLC data endianness.
- The CRC type (16-bit or 32-bit) to be used for this HDLC port.
- CAS bit polarity and bit inversion.

For more details, see “Packetized Connect and Enable” on page 172.

14.3.4 High-Level API Call Flow

The steps below describe the high-level API call-process flow for initializing, configuring, and using the IxHssAcc component.

1. The proper NPE microcode images must be downloaded to the NPEs and initialized, if applicable. Also, the IxNpeMh and IxQMgr components must be initialized.
2. Client calls **ixHssAccInit()**. This function is responsible for initializing resources for use by the packetised and channelised clients.
3. 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 will configure each time slot in a frame to provide either packetized or channelized service as well as other characteristics of the HSS port.
4. 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 “HSS Port Initialization Details” on page 165.
5. The client then calls the **ixHssAccPktPortConnect()** or **ixHssAccChanConnect()** to connect the client to the IxHssAcc service. Additionally, the client provides callback functions for the service to inform the client when data is received and ready to delivered to the client.
6. The client will begin receiving data once a port is enabled. The functions to enable the packetized or channelized service ports are **ixHssAccPktPortEnable()** and **ixHssAccChanPortEnable()**.

   As traffic is being transmitted and/or received on the HSS interfaces and passed to the client, via a channelized or packet service, a variety of tasks may be called by the client to check the status, replenish buffers, retrieve statistics, etc. Callback functions or a polling mechanism are used in the transmitting and receiving process.

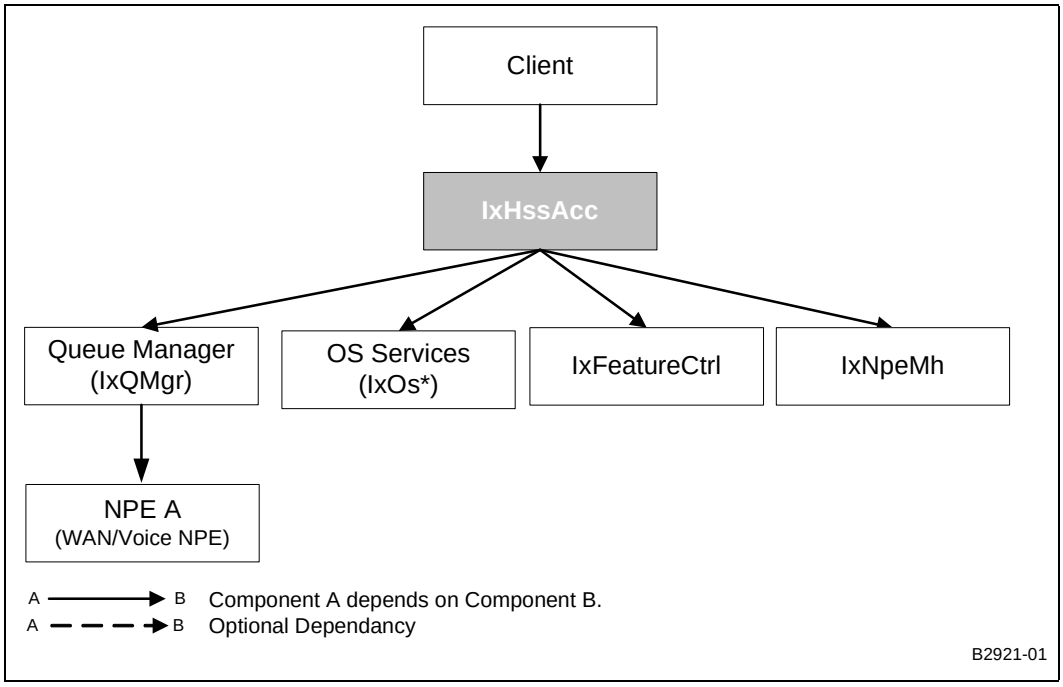
   The client will process the received data or provide new data for transmission. This is done by providing new buffer pointers or by adjusting the existing pointers. The data path and requisite buffer management are described in more detail in “Buffer Allocation Data-Flow Overview” on page 179.

7. Finally, when the HSS component is no longer needed, **ixHssAccPktPortDisable()** and/or **ixHssAccPktPortDisconnect()** — or **ixHssAccChanDisconnect()** and/or **ixHssAccChanPortDisable()** — are called. The Disable functions will instruct the NPE's to stop data handling, while the Disconnect functions will clear all port configuration parameters. The Disconnect functions will automatically disable the port.

14.3.5 Dependencies

Figure 52 on page 164 shows the component dependencies of the IxHssAcc component.

Figure 52. IxHssAcc Component Dependencies



The dependency diagram can be summarized as follows:

- Client component will call IxHssAcc for HSS and HDLC data services. NPE A will perform the protocol conversion, signalling on the HSS interfaces, and data handling.
- IxHssAcc depends on the IxQMgr component to configure and use the hardware queues to pass data between the Intel XScale core and the NPE.
- NpeMh is used by the component to configure the HSS and HDLC co-processor operating characteristics.
- OS Services components are used for error handling and critical code protection.
- IxFeatureCtrl is used to detect the existence of the required hardware features on the host processor. Specifically, IxHssAcc detects the existence of NPE A.

14.3.6 Key Assumptions

The HSS service is predicated on the following assumptions:

- 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.

14.3.7 Error Handling

The IxHssAcc component will use IxOsServices to report internal errors and warnings. Parameters passed through the IxHssAcc API interfaces will be error checked whenever possible.

HDLC CRC errors and byte alignment errors will be reported to packetized clients on a per packet basis. Port disable and disconnect errors on a transmit or receive packetized service pipe will be transmitted to the client as well.

HSS port errors such as over-run, under-run and frame synchronization will be counted by NPE A, along with other NPE software errors. This count of the total number of errors since configuration will be reported to packetized clients on a per packet basis and to channelized clients at the trigger rate.

IxHssAcc provides an interface to the client to read the last error from the NPE. There is no guarantee that the client will be able to read every error. A second error may occur before the client has had the opportunity to read the first one. The client will, however, have an accurate total error count.

14.4 HSS Port Initialization Details

***ixHssAccPortInit*()**

The HSS ports must be configured to match the configuration of any connected PHY. No channelized or packetized connections should exist in the IxHssAcc layer while this interface is being called.

This includes configuring the time slots within a frame in one of the following ways:

- Configuring as HDLC — For packetized service, include raw packet mode
- 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

This function takes the following arguments:

- IxHssAccHssPort hssPortId — The HSS port ID.
- IxHssAccConfigParams *configParams — A pointer to the HSS configuration structure.

- IxHssAccTdmSlotUsage *tdmMap — A pointer to an array defining the HSS time-slot assignment types.
- IxHssAccLastErrorCallback lastHssErrorCallback — 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 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 56Kbps data endianness, when using the 56Kbps type (Tx)
- Set 56Kbps data transmission type, when using the 56Kbps 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 co-processor 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	Packetized
IX_HSSACC_TDMMAP_VOICE56K	Channelized
IX_HSSACC_TDMMAP_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.

When complete, the HSS coprocessor will be running, although no access is given to the client until a connect occurs followed by an enable.

14.5 HSS Channelized Operation

14.5.1 Channelized Connect and Enable

ixHssAccChanConnect()

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, and 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:

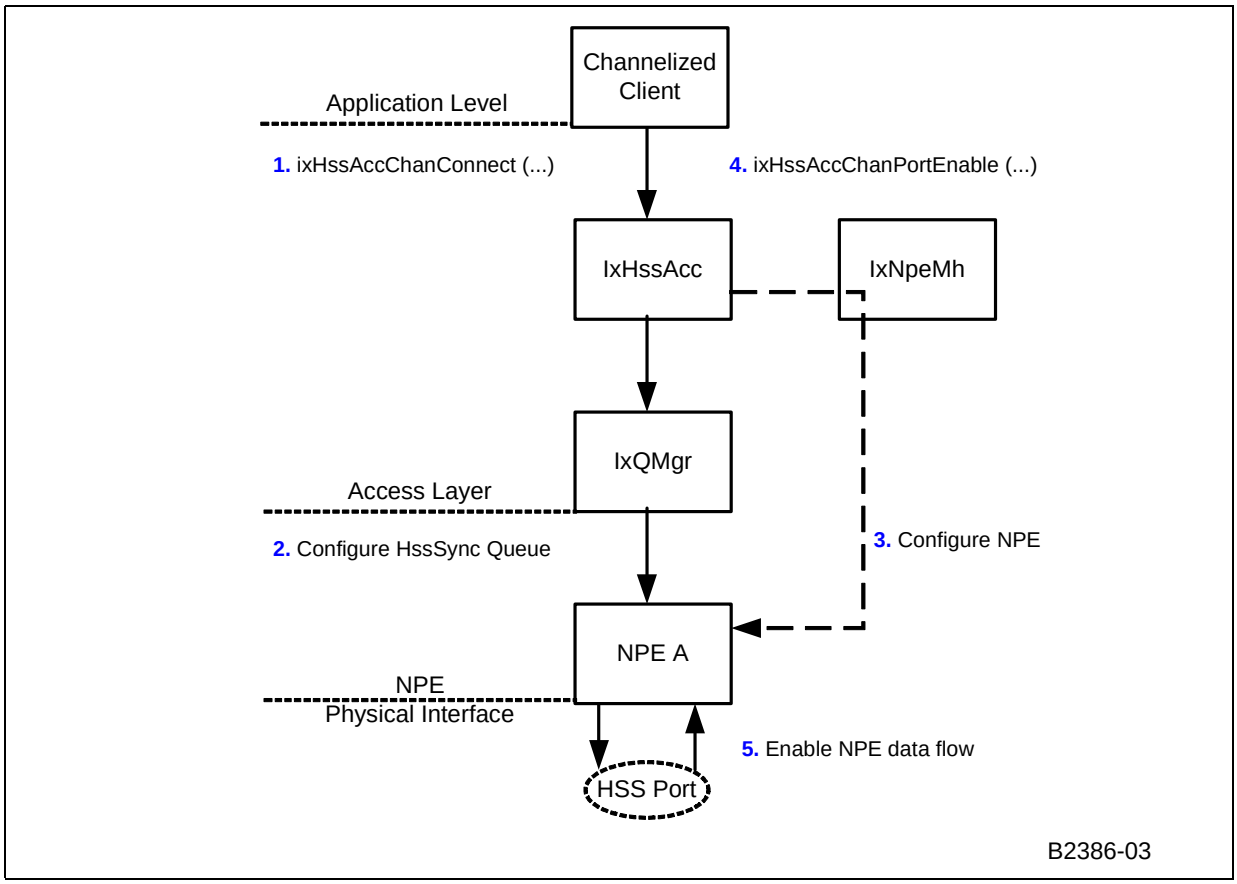
- IxHssAccHssPort hssPortId — The HSS port ID. There are two identical ports (0-1).
- unsigned bytesPerTSTrigger — The NPE will trigger the access component to call the Rx call back function rxCallback() after bytesPerTSTrigger bytes have been received for all trunk time slots. bytesPerTSTrigger is a multiple of eight. For example: 8 for 1-ms trigger, 16 for 2-ms trigger.

- UINT8 *rxCircular — 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 — The length of each Rx circular buffer in the Rx data pool. The buffers need to be deep enough for data to be read by the client before the NPE re-writes over that memory.
- UINT32 *txPtrList — 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 — The number of pointer lists in txPtrList. This number is dependent on jitter.
- unsigned numTxBytesPerBlk — The size of the Tx data, in bytes, that each pointer within the PtrList points to.
- IxHssAccChanRxCallback rxCallback — 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 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 figure shows what is done in IxHssAcc when the ixHssAccChanPortConnect() and ixHssAccChanPortEnable() functions are called.

Figure 53. Channelized Connect



1. The client issues a channelized connect request to IxHssAcc.
2. If an rxCallback is configured, the client expects to be triggered by events to drive the Tx and Rx block transfers. IxHssAcc registers the function pointer with IxQMgr to be called back in the context of an ISR when the HssSync queue is not empty.
3. IxHssAcc configures the NPE appropriately.
4. The client enables the channelized service through IxHssAcc.
5. IxHssAcc enables the NPE flow.

   If the service was configured to operate in polling mode (i.e., the rxCallback pointer is NULL), the client must poll the IxHssAcc component using the ixHssAccChanStatusQuery() function. IxHssAcc will check the HssSync queue status and return a pointer to the client indicating an offset to the data in the Rx buffers, if any receive data exists at that time.

14.5.2 Channelized Tx/Rx Methods

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

14.5.2.1 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 a 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 preparation, 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 NPE since the buffers are circular.

For TX data preparation, *rxCallback*() provides 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.

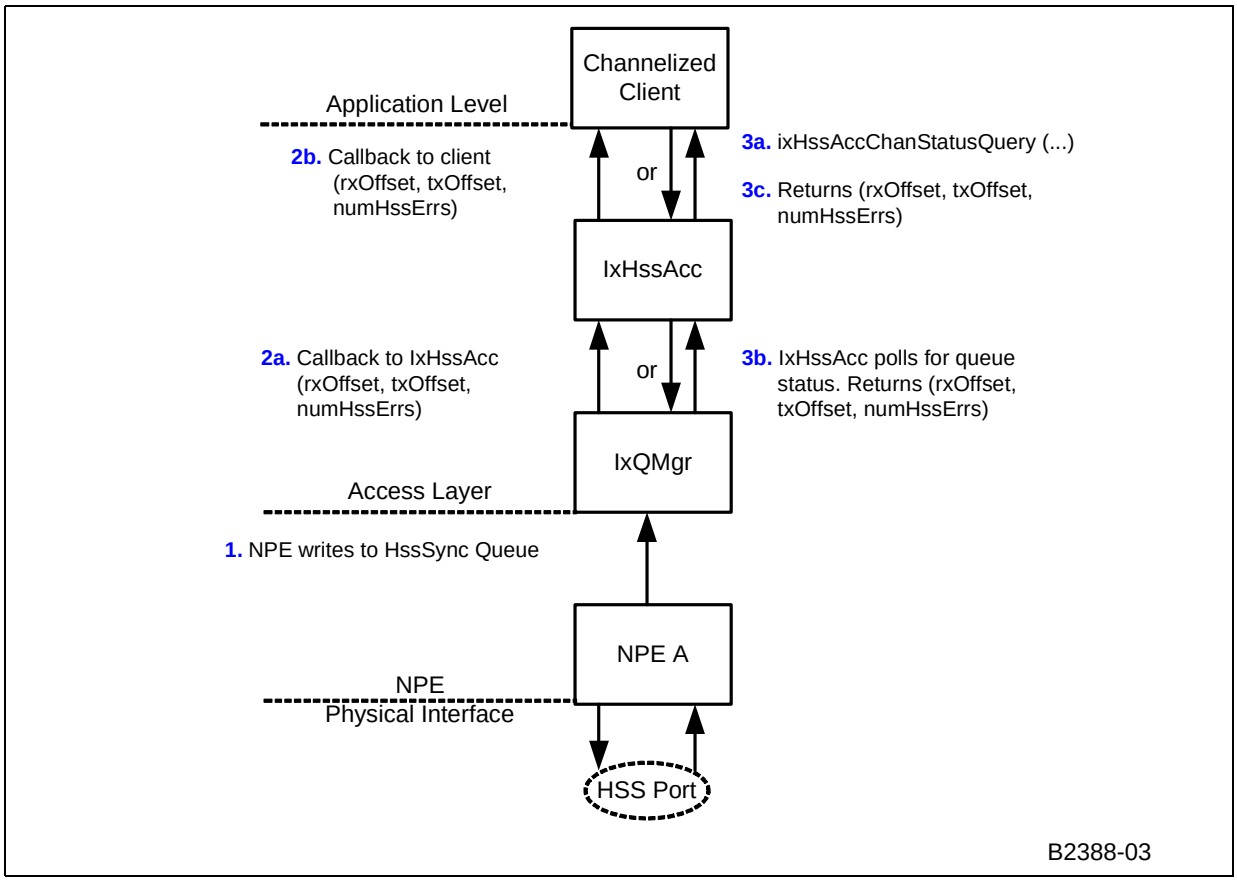
14.5.2.2 Polled

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 54 shows the Tx/Rx process.

Figure 54. Channelized Transmit and Receive



1. After reading a configurable amount of data from the HSS port and writing the same amount of data to the HSSport, the NPE writes to the hssSync queue.

   There are two possible paths after that depending on how the client is connected:

2. Callback Mode

   a. Through an interrupt, the IxQMgr component will callback IxHssAcc with details of the hssSync queue entry.

   b. IxHssAcc will initiate the registered callback of the client.

OR

3. Polling Mode

   a. The client will poll IxHssAcc using ixHssAccChanStatusQuery().

   b. IxHssAcc will, in turn, poll IxQMgr's hssSync queue for status.

   c. If IxHssAcc reads an entry from the hssSync queue, it returns the details to the client.

14.5.3 Channelized Disconnect

When the channelized service is not needed any more on a particular HSS port, ixHssAccChanPortDisable() is called to stop the channelized service on that port, then ixHssAccChanDisconnect() is called to disconnect the service.

14.6 HSS Packetized Operation

14.6.1 Packetized Connect and Enable

ixHssAccPktPortConnect()

After the HSS component is configured, *ixHssAccPktPortConnect*() has to be called to connect the client application with the packetized services. This function is responsible for connecting a client to one of the four available packetized ports on a configured HSS port.

There are four packetized services per HSS port, so this function has to be called once per packetized service.

Note: This functions structures have changed significantly in software release 1.4 and the function is no longer directly compatible with previous versions.

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.

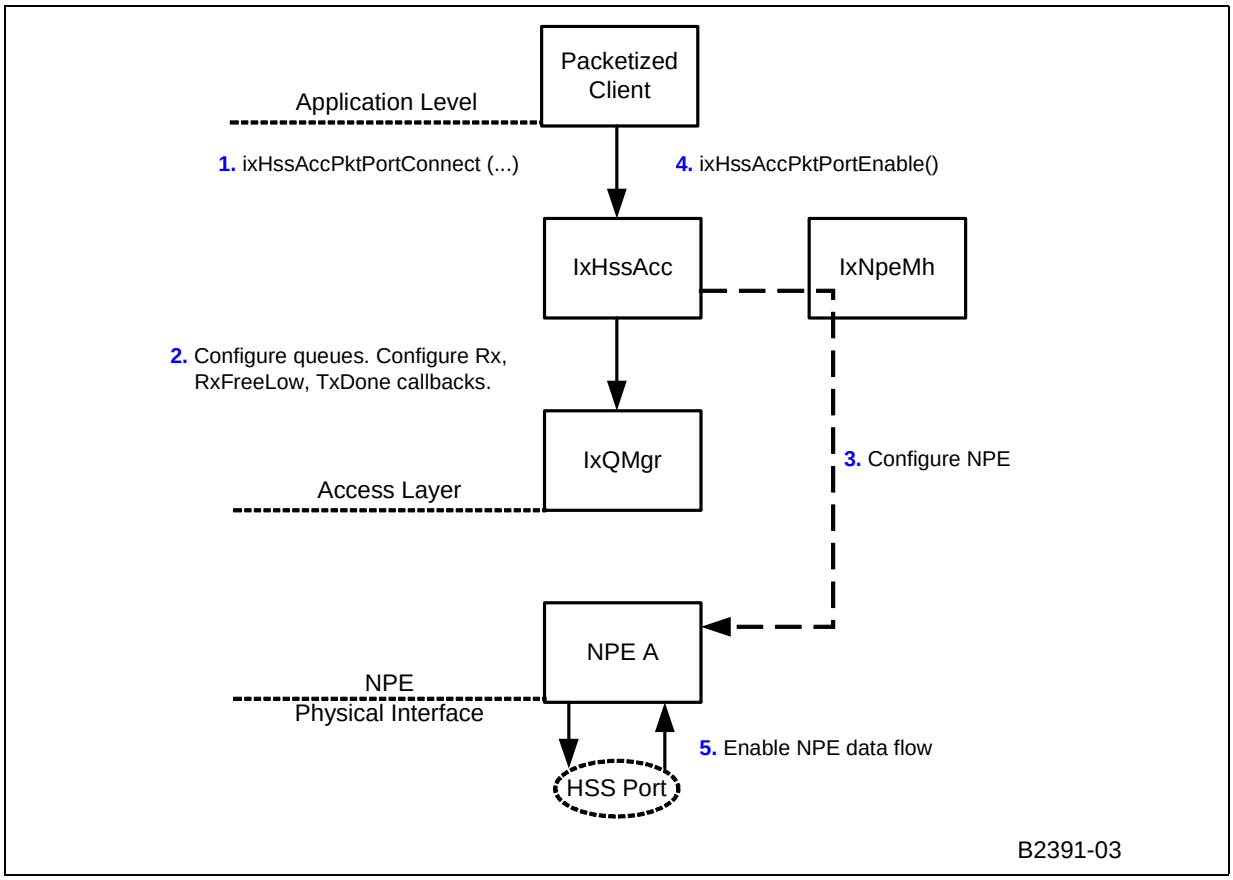
The parameters for *ixHssAccPktPortConnect*() include:

- IxHssAccHssPort hssPortId — The HSS port ID. There are two identical ports (0-1).
- IxHssAccHdlcPort hdlcPortId — 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 — This value determines whether the service will use HDLC data or the raw data type (i.e., no HDLC processing).
- IxHssAccHdlcMode hdlcMode — This structure contains 56Kbps, HDLC-mode configuration parameters.
- BOOL hdlcBitInvert — This value determines whether bit inversion will occur between HDLC and HSS coprocessors (i.e., post-HDLC processing for transmit and pre-HDLC processing for receive, for the specified HDLC Termination Point).

- unsigned blockSizeInWords — The max tx/rx block size.
- UINT32 rawIdleBlockPattern — Tx idle pattern in raw mode.
- IxHssAccHdlcFraming hdlcTxFraming — This structure contains the following information required by the NPE to configure the HDLC coprocessor for Tx.
- IxHssAccHdlcFraming hdlcRxFraming — This structure contains the following information required by the NPE to configure the HDLC coprocessor for Rx.
- unsigned frmFlagStart — Number of flags to precede to transmitted flags (0-2).
- IxHssAccPktRxCallback rxCallback — Pointer to the clients packet receive function.
- IxHssAccPktUserId rxUserId — The client supplied Rx value to be passed back as an argument to the supplied rxCallback.
- IxHssAccPktRxFreeLowCallback rxFreeLowCallback — Pointer to the clients Rx-free-buffer request function. If NULL, it is assumed client will free Rx buffers independently.
- IxHssAccPktUserId rxFreeLowUserId — The client supplied RxFreeLow value to be passed back as an argument to the supplied rxFreeLowCallback.
- IxHssAccPktTxDoneCallback txDoneCallback — Pointer to the clients Tx done callback function.
- IxHssAccPktUserId txDoneUserId — 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. Figure 55 shows what is done in IxHssAcc when the packetized service connect function is called.

Figure 55. Packetized Connect



1. The client issues a packet service connect request to IxHssAcc.
2. IxHssAcc instructs IxQMgr to configure the necessary queues and register callbacks.
3. IxHssAcc configures the NPE with the HDLC parameters passed by the client.
4. The client enables the packet service.
5. IxHssAcc enables the NPE flow.

14.6.2 Packetized Tx

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 data 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 TxDone 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:

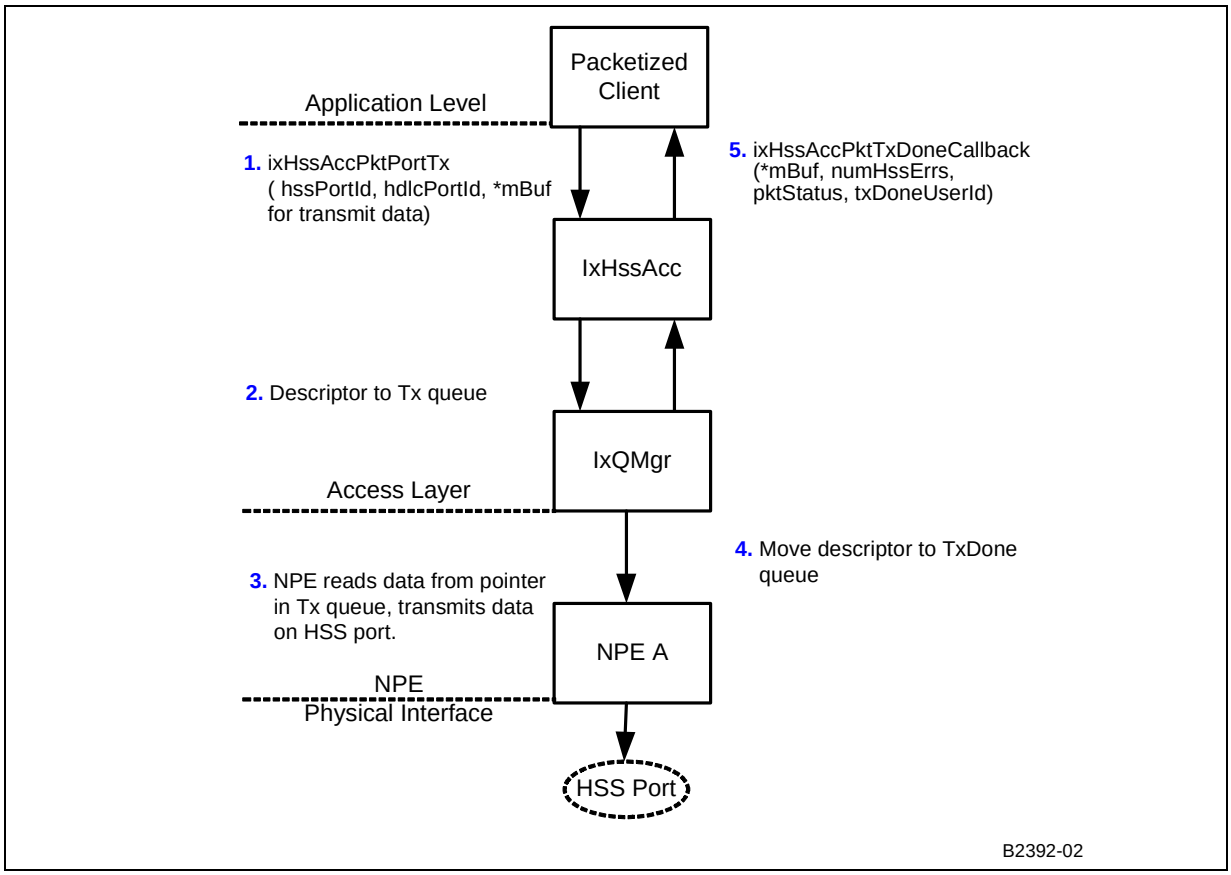
```
IX_MBUF *txBuffer;
IX_MBUF *txBufferChain = NULL;
// get a mBuf
IX_MBUF_POOL_GET(poolId, &txBuffer);
// set the data length in the buffer
IX_MBUF_MLEN(txBuffer) = NumberOfBytesToSend;
/* set the values to transmit */
for (byteIndex = 0; byteIndex < IX_MBUF_MLEN(txBuffer); byteIndex++)
((UINT8 *)IX_MBUF_MDATA(txBuffer))[byteIndex] =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 56.

Figure 56. Packetized Transmit



1. The client presents an mBuf to IxHssAcc for transmission.
2. IxHssAcc gets a transmit descriptor from its transmit descriptor pool, fills in the descriptor, and writes the address of the descriptor to the Tx queue.
3. The NPE reads a transmit descriptor from the Tx queue and transmits the data on the HSS port.
4. On completion of transmission, the NPE writes the descriptor to the TxDone queue.
5. IxHssAcc is triggered by this action, and the registered callback is executed. The descriptor is freed internally.
6. IxHssAcc initiates the TxDoneCallback on the client, passing it back its mBuf pointer.

14.6.3 Packetized Rx

Before packetized service is enabled, the Rx queue in the IxHssAcc component has to be replenished. This can be done by calling *IX_MBUF_POOL_GET*() to get an 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 the 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 callback 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 callback function, and perform Rx data processing 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 mBuf, depending on the packet size received. *IX_MBUF_NEXT_BUFFER_IN_PKT_PTR*() can be used to get access to each of the mBuf 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);
```

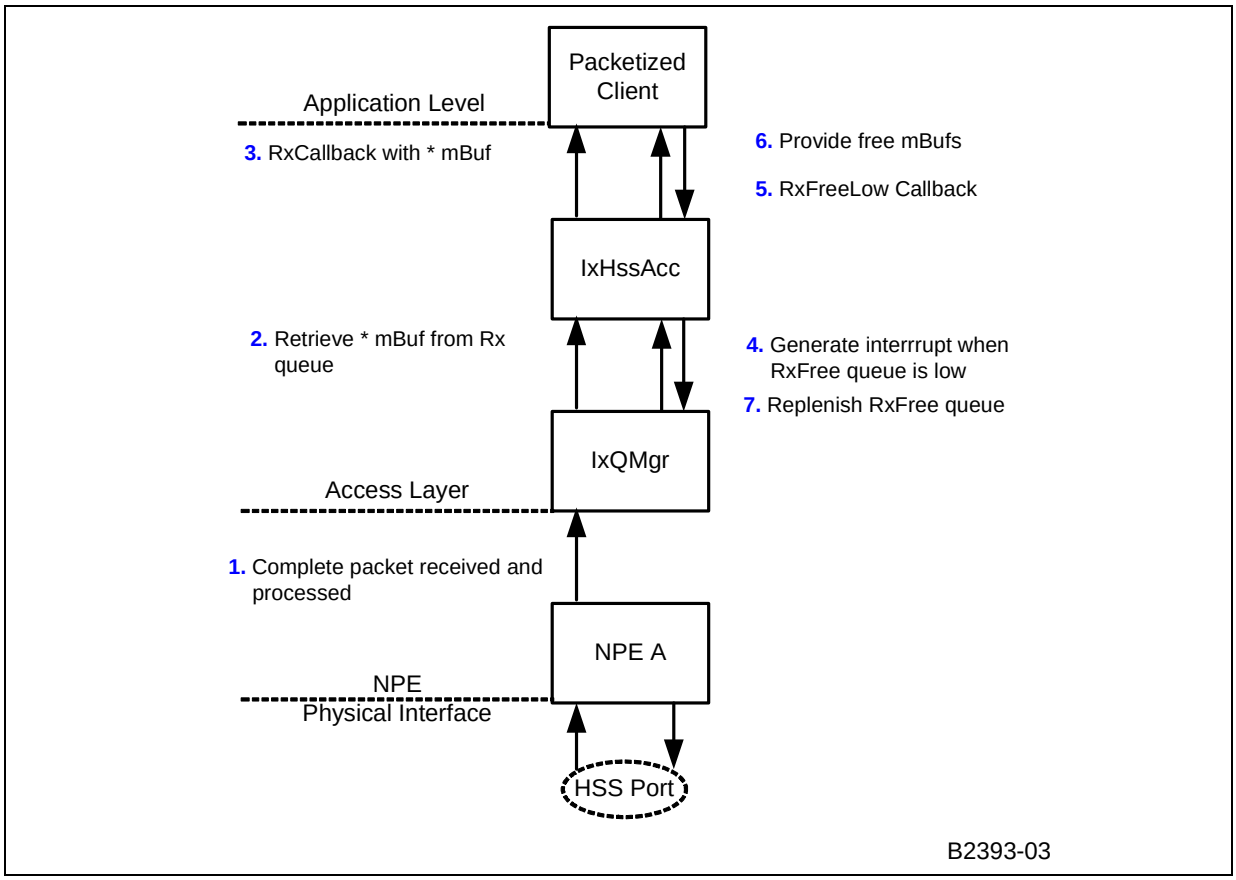
rxCallback() also passes the packet status and the number of errors that NPE receives. The packet status is used to determine if the packet received is good or bad, and the client can call function *ixHssAccLastErrorRetrievalInitiate*() to initiate the retrieval of the last HSS error.

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 suppling mBufs to the queue. This is the case if the pointer for *rxFreeLowCallback*() passed to *ixHssAccPktPortConnect*() is *NULL*.

The process is show in Figure 57.

Figure 57. Packetized Receive



1. When a complete packet is received, the Rx queue call-back function is invoked in an interrupt.
2. The descriptor is pulled from the Rx queue and the callback for this channel is invoked with the descriptor. The descriptor gets recycled.
3. The mBuf is transmitted to the client.
4. When the RxFree queue is low, IxQMgr triggers an interrupt to IxHssAcc.
5. IxHssAcc triggers the client rxFreeLowCallback function, which was registered during the client connection process.
6. The client provides free mBufs for specific packetized channels.
7. Free mBufs are stored in the RxFree queue, and listed within the IxHssAcc Rx descriptors.

14.6.4 Packetized Disconnect

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

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

14.6.5 56-Kbps, Packetized Raw Mode

When a packet service channel is configured for 56-Kbps, packetized Raw mode, byte alignment of the transmitted data is not preserved. All raw data that is transmitted by a device using IxHssAcc in this manner will be received in proper order by the receiver (the external PHY device, for example). However, the first bit of the packet may be seen at any offset within a byte. All subsequent bytes will have the same offset for the duration of the packet. The same offset also applies to all subsequent packets received on the service channel as well.

The receive data path is identical to the scenario described above.

While this behavior will also occur for 56-Kbps, packetized HDLC mode, the HDLC encoding/ decoding will preserve the original byte alignment at the receiver end.

14.7 Buffer Allocation Data-Flow Overview

Prior to connecting and enabling ports in IxHssAcc, a client must allocate buffers to the IxHssAcc component. IxHssAcc provides two services, packetized and channelized, and the clients exchange data with IxHssAcc differently for transmitting and receiving, depending on which service is chosen.

Note: The IxHssAcc component addressing space for physical memory is limited to 28 bits. Therefore mBuf headers should be located in the first 256 Mbytes of physical memory.

14.7.1 Data Flow in Packetized Service

Data in the time slots configured for HDLC or raw services will form packets for packetized service. IxHssAcc supports up to 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");
```

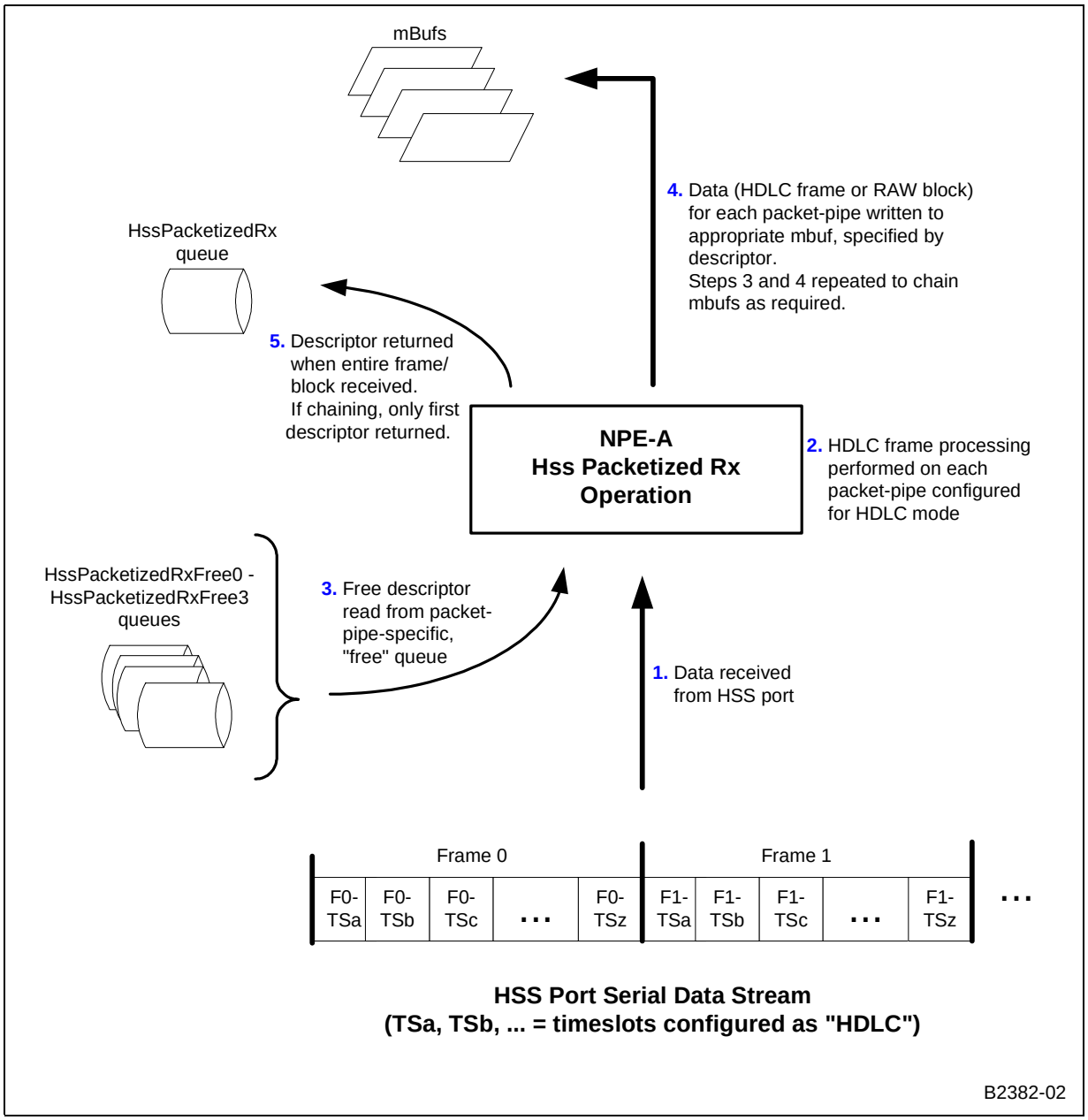
A mBuf can be obtained from the pool by calling *IX_MBUF_POOL_GET*(). This mBuf pool is shared by the Tx and Rx processes.

For Rx, before the packetized service is enabled, the Rx mBuf queue in IxHssAcc 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 IxHssAcc packetized service when *ixHssAccPktPortConnect*() is called. Alternatively, the client can use its own timer to regularly supply 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. The Rx packetized data flow is shown in Figure 58 on page 181.

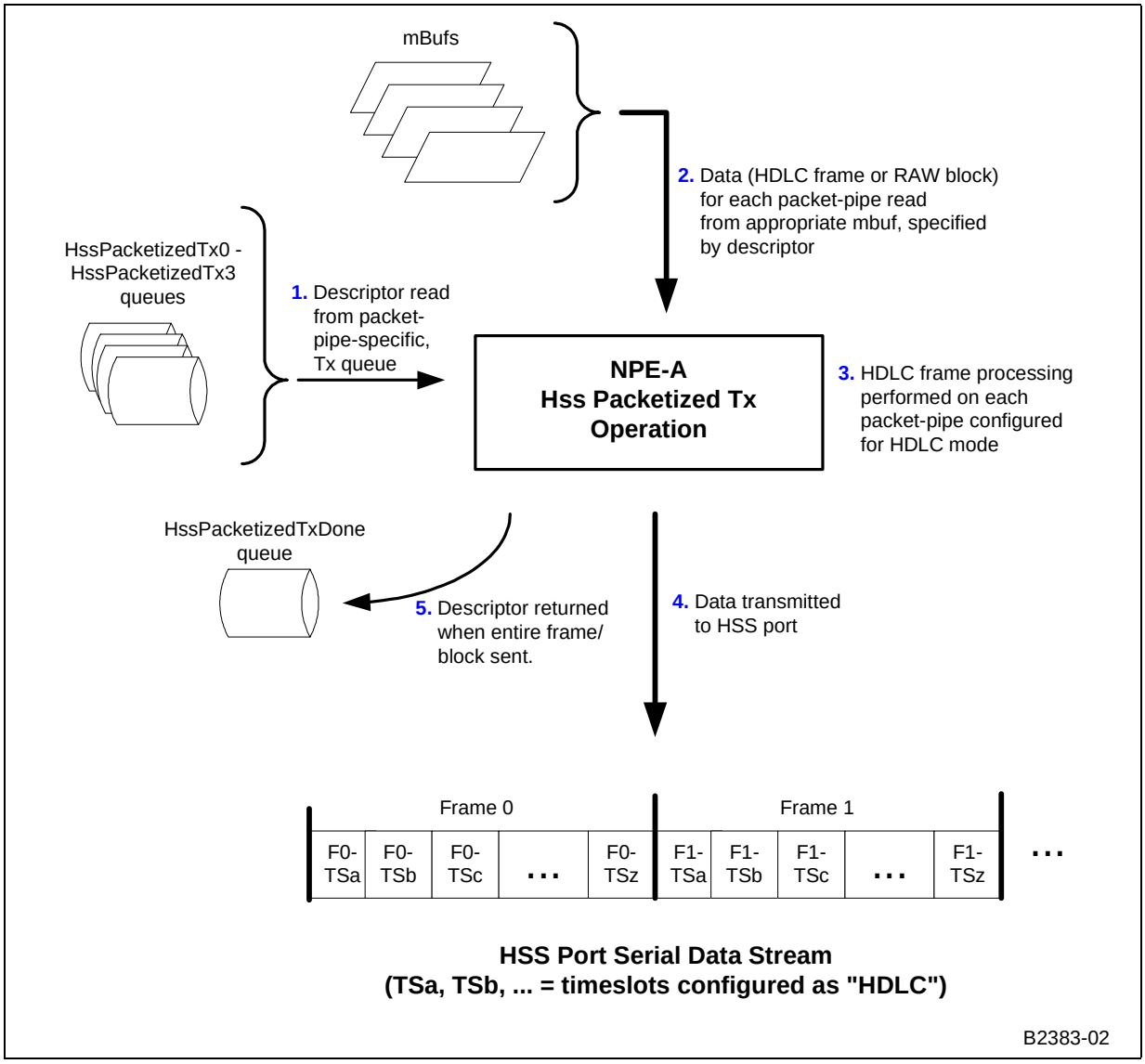
Figure 58. HSS Packetized Receive Buffering



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 IxHssAcc for transmission by calling *ixHssAccPktPortTx*().

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 mBufs can be returned to the mBuf pool. The packetized Transmit data flow is described in Figure 59.

Figure 59. HSS Packetized Transmit Buffering



14.7.2 Data Flow in Channelized Service

Data in the time slots configured as Voice64K/Voice56K types will be provided to the client via the IxHssAcc 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 IxHssAcc to transmit and receive data through the HSS port.

For receive, *ixOsServCacheDmaAlloc()* of the IxOSCacheMMU 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.

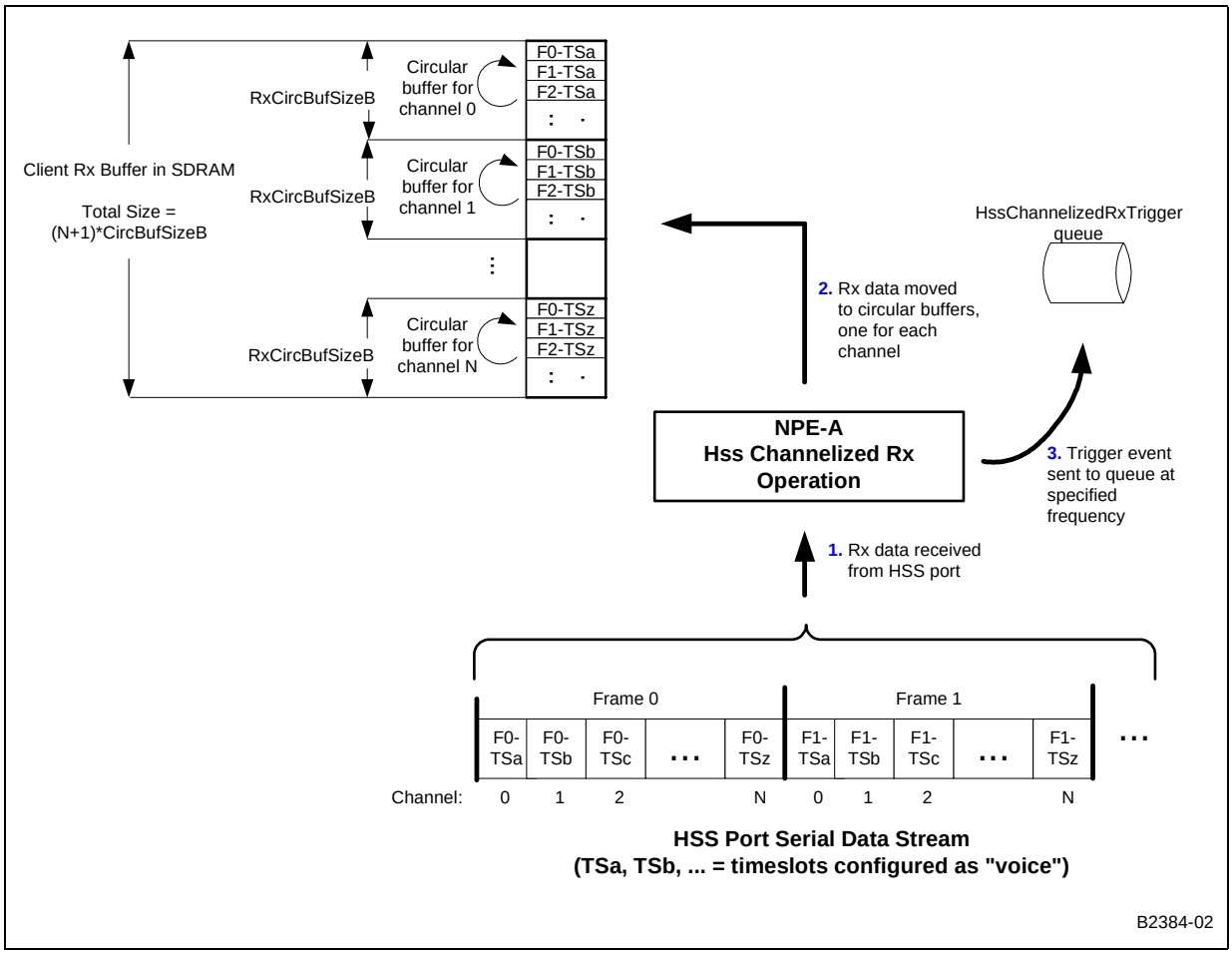
All the buffers have the same length. When the channelized service is initialized by *ixHssAccChanConnect*(), the pointer to the pool, the length of the circular buffers, and a parameter *bytesPerTStrigger* are passed to IxHssAcc, as well as a pointer to the an ixHssAccChanRxCallback() Rx callback function.

Figure 60 shows how the circular buffers are filled with data received though the HSS ports. 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 and complete any possible in-place processing that would need to occur before the NPE rewrites over that memory. Understanding the client application's read and processing latency, the size of the data unit needed by the client application for processing, and the rate at which the NPE writes data to a buffer at a given channel rate, are useful in making this calculation.

Figure 60 on page 184 shows the data flow of the channelized Receive service.

Figure 60. HSS Channelized Receive Operation



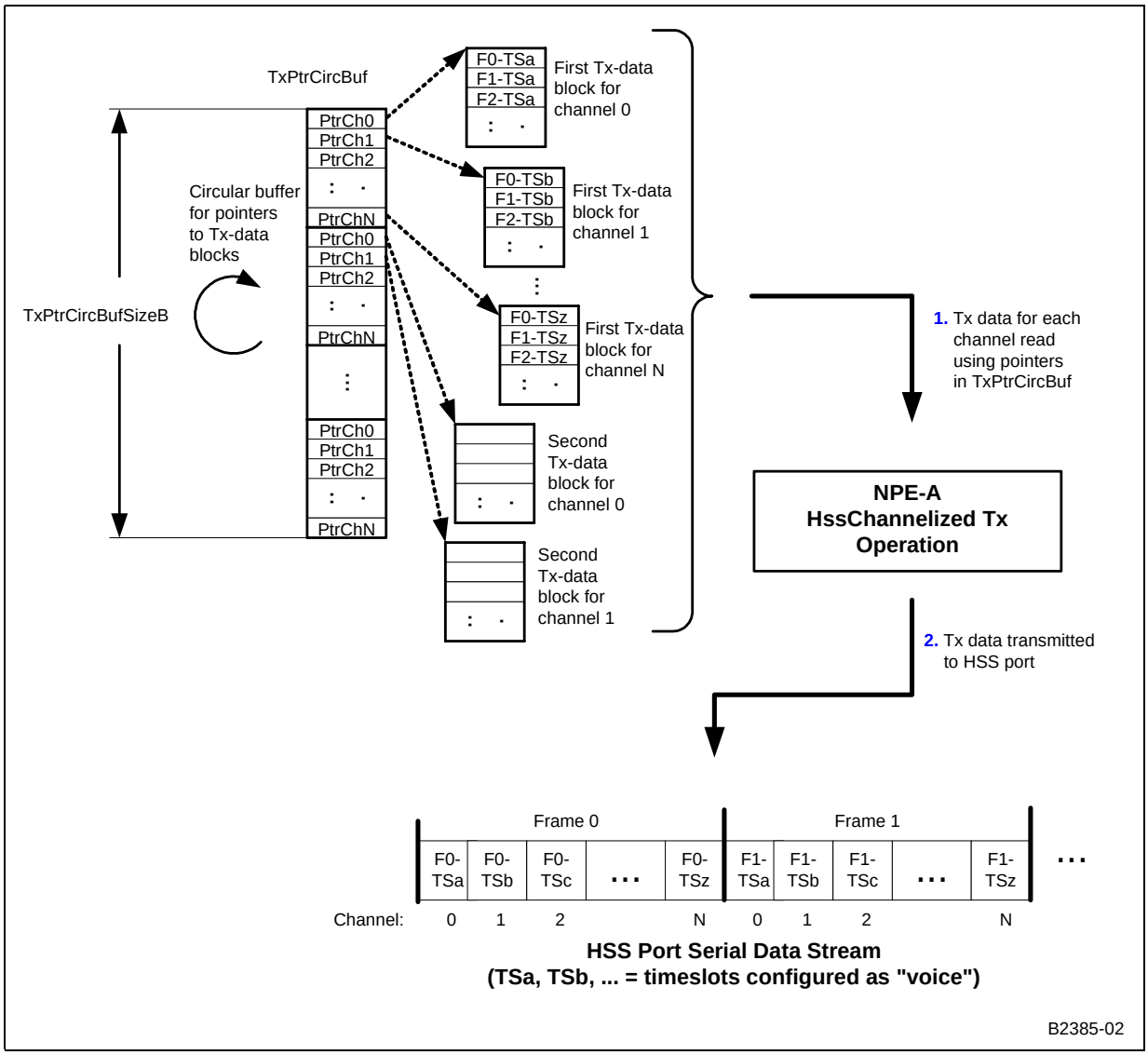
For transmission, *ixOsServCacheDmaMalloc*() is used to allocated two pools: a data buffer pool and a pointer list pool. The data buffer pool has *N* buffers — one for each channel. Each buffer is divided into *K* sections and each section has *L* bytes. The pointer list pool has *K* pointer lists. Each 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.

The client can check the current location of data being transmitted by using the registered *ixHssAccChanRxCallback()* function. When the 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 61. HSS Channelized Transmit Operation



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

This chapter describes the Intel® IXP400 Software v1.4's "NPE-Downloader API" access-layer component.

15.1 What's New

The following changes and enhancements were made to this component in software release 1.4:

- New NPE microcode images for NPE A and NPE C were added to support WEP services in the IxCryptoAcc component.
- The IX_NPEDL_NPEIMAGE_NPEB_ETH_FPATH image has been removed as Fast Path support has been discontinued.

15.2 Overview

The NPE downloader (ixNpeDl) component is a stand-alone component providing a facility to download a microcode image to NPE A, NPE B, or NPE C in the system. The IxNpeDl 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 microcode images 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.

NPE Image Compatibility

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 1.4 library. The NPE images are provided in the form of a single header file incorporated within the software release package, and those NPE microcode images are assumed compatible for that specific release.

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 C with Ethernet and Crypto functionality:

```
ixNpeDlNpeInitAndStart(IX_NPEDL_NPEIMAGE_NPEC_CRYPTO_ETH);
```

The parameter is a UINT32 that is defined in the IXP425 NPE image ID definition. Table 28 lists the parameters for the standard images.

Table 28. NPE-A 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 AAL 5, AAL 0 and OAM for UTOPIA SPHY, 1 logical port, 32 VCs.
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 AAL 5, AAL 0 and OAM for UTOPIA MPHY, 1 logical port, 32 VCs.
IX_NPEDL_NPEIMAGE_NPEA_ATM_MPHY_12_PORT	NPE Image ID for NPE-A with ATM-Only feature. It supports AAL 5, AAL 0 and OAM for UTOPIA MPHY, 12 logical ports, 32 VCs.
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_NPEA_WEP	NPE Image ID for NPE-A with ARC4 and WEP CRC engines.
IX_NPEDL_NPEIMAGE_NPEC_CRYPTO_AES_ETH	NPE Image ID for NPE-C with Crypto and Eth features. For Crypto, it supports AES, DES, SHA-1, MD 5. AES-CCM mode is not supported.

Table 29. NPE-B Images

Image Name	Description
IX_NPEDL_NPEIMAGE_NPEB_ETH	NPE Image ID for NPE-B with Ethernet-Only feature.
IX_NPEDL_NPEIMAGE_NPEB_DMA	NPE Image ID for NPE-B with DMA-Only feature.

Table 30. NPE-C Images

Image Name	Description
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. AES-CCM mode is not supported.
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. AES-CCM mode is not supported.
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. AES-CCM mode is not supported.
IX_NPEDL_NPEIMAGE_NPEC_CRYPTO_AES_CCM	NPE Image ID for NPE-C with Crypto-Only feature. For Crypto, it supports AES, CCM, DES, SHA-1, MD5.
IX_NPEDL_NPEIMAGE_NPEC_CRYPTO_AES_CCM_ETH	NPE Image ID for NPE-C with Crypto and Eth feature. For Crypto, it supports AES, CCM, DES, SHA-1, MD5. For Ethernet, MAC address learning disabled (but filtering is still enabled).
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. External libraries come in the form of a header file. The header file defines the image library as a single array of type UINT32, and it is that array symbol that should be used as the *imageLibrary* parameter for that function.

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 IxNpeDl.) If possible, IxNpeDl also should be uninitialized before a soft reboot.

Here is a sample function call to uninitialize IxNpeDl:

```
ixNpeDlUnload();
```

Note: Calling ixNpeDlUnload twice or more in succession will cause all subsequent calls after the first one to exit harmlessly but return a FAIL status.

15.7 Deprecated APIs

As of software release 1.3, 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

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

This chapter describes the Intel® IXP400 Software v1.4's "NPE Message Handler API" access-layer component.

16.1 What's New

There are no changes or enhancements to this component in software release 1.4.

16.2 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 IxEthAcc. In the sections that describe how the messaging works, the "client" is an access component such as IxEthAcc. 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 in-FIFOs, while messages are received from the NPEs out-FIFOs. Both the in-FIFO and out-FIFO have a depth of two messages, and each messages is two words in length.

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.

16.3 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.

16.3.1 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.

16.3.2 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 from inside an interrupt service routine as it will use resource protection mechanisms.

16.4 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.

16.5 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.

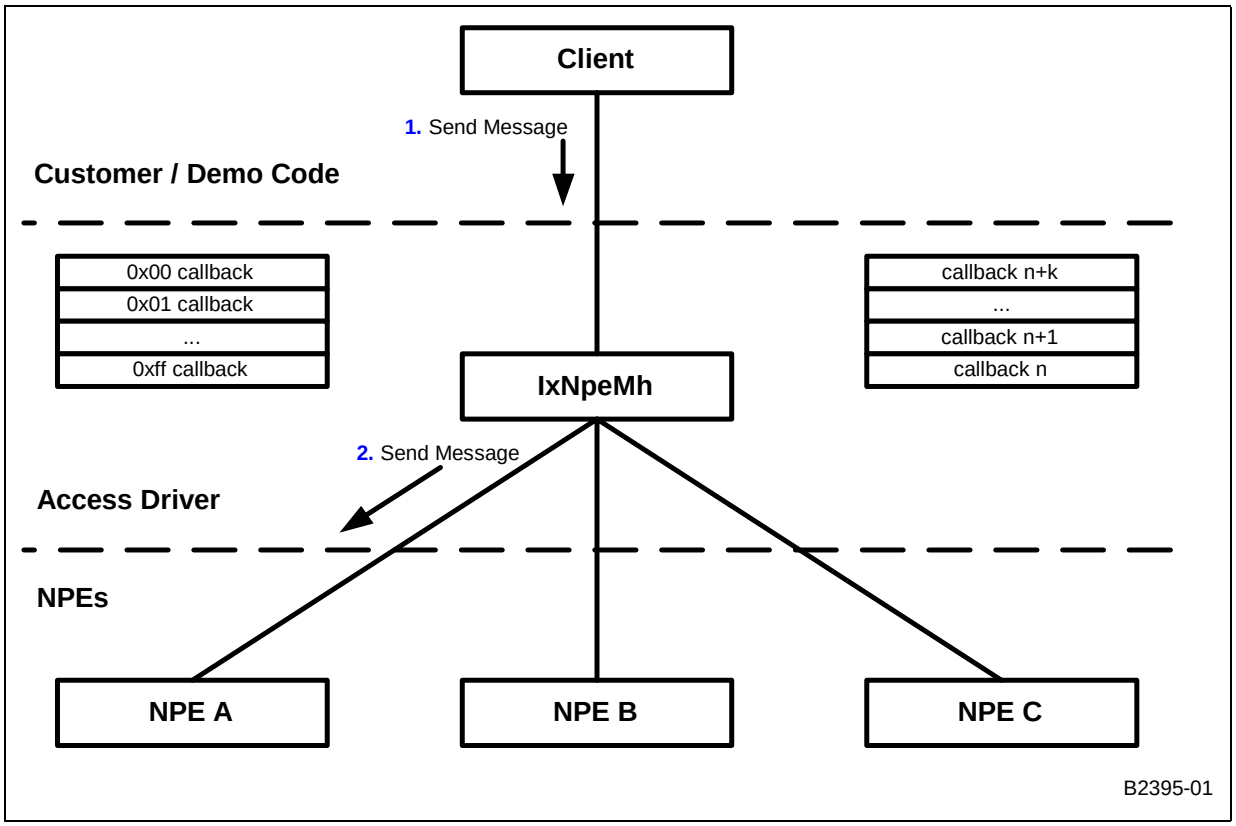
16.5.1 Sending an NPE Message

The scenario of sending a messages from an Intel XScale core software client to an NPE (as shown in Figure 62) 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 62. Message from Intel XScale® Core Software Client to an NPE



16.5.2 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 63) 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.

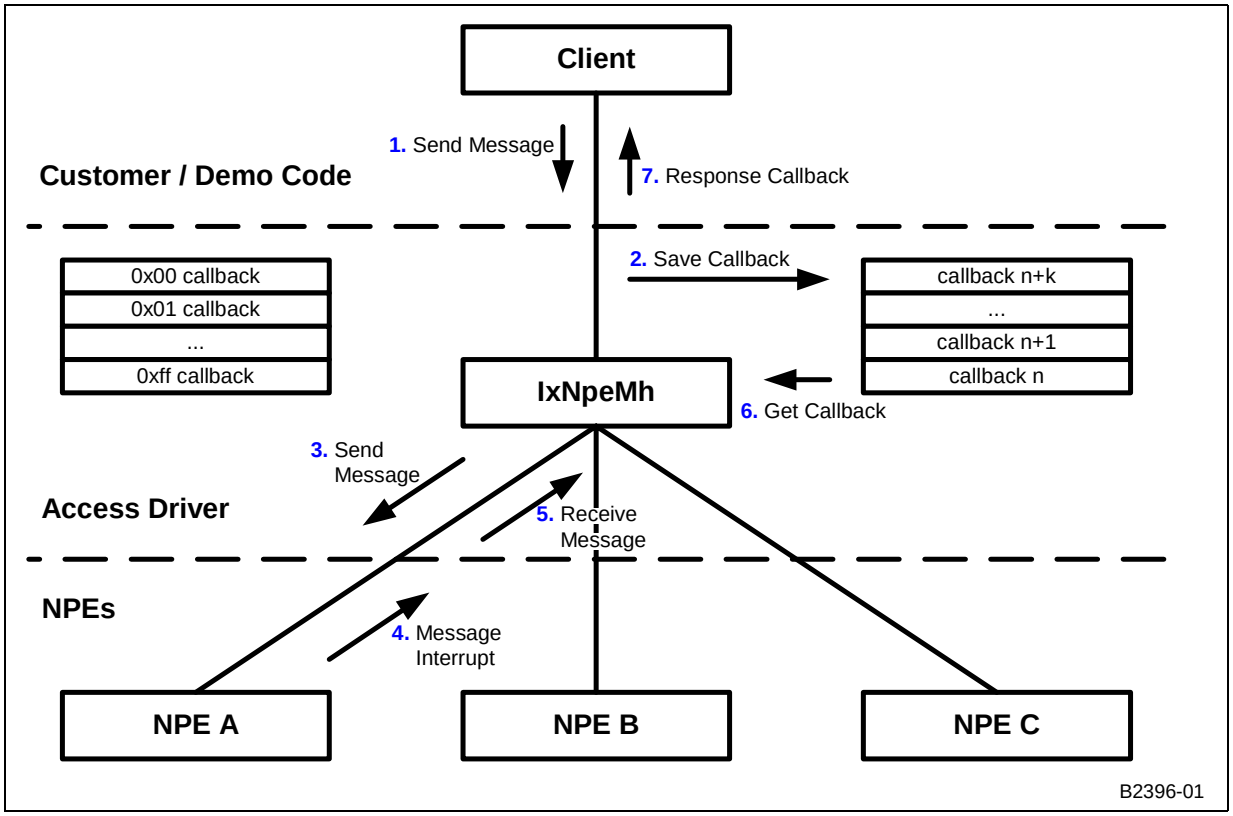
   If the component cannot accept a message, the send fails.
3. The IxNpeMh component adds the response callback to the end of the solicited callback list and sends the message to the NPE.
4. After some time, the NPEs "outFIFO not empty" interrupt invokes the IxNpeMh component's ISR.
5. Within the ISR, 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 195.)

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



16.6 Receiving Unsolicited Messages from an NPE to Client Software

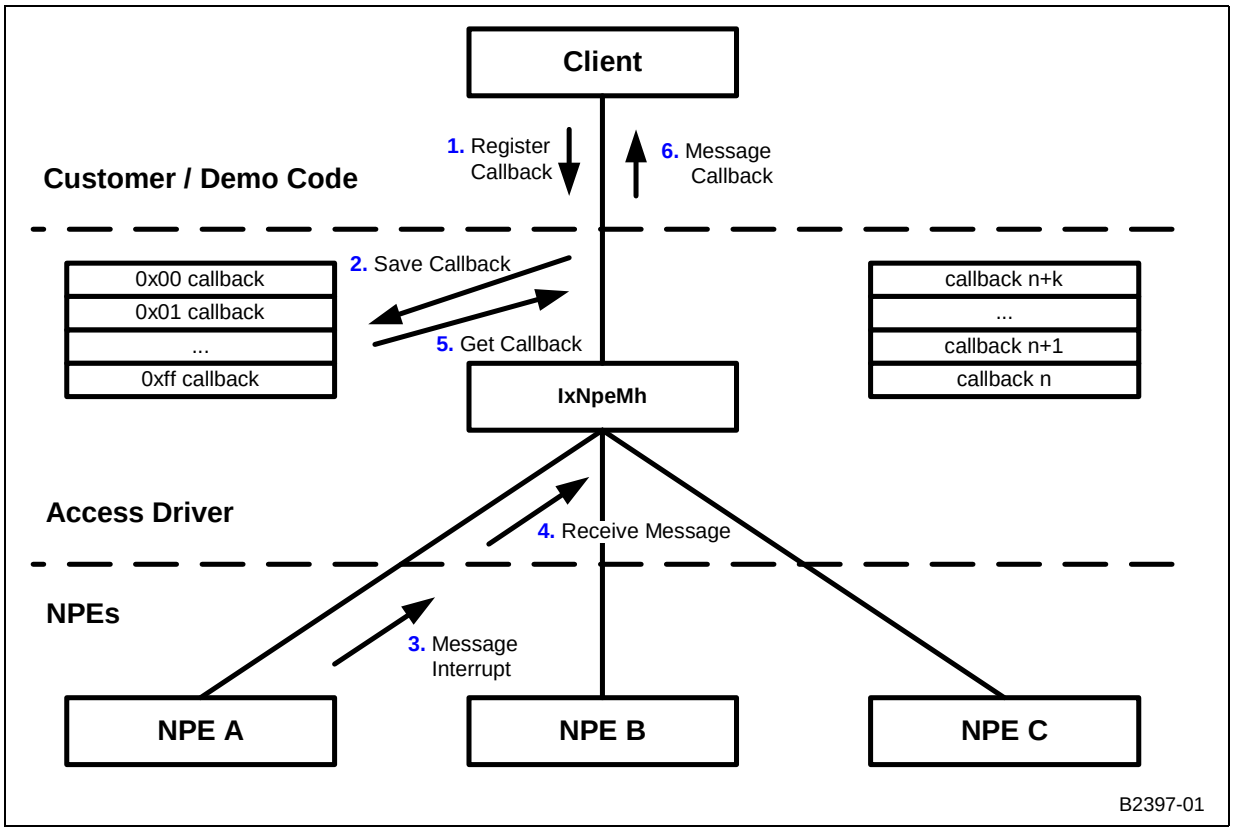
The scenario of receiving unsolicited messages from an NPE to client software (as shown in Figure 64) is:

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 194.)

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

Figure 64. 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.)

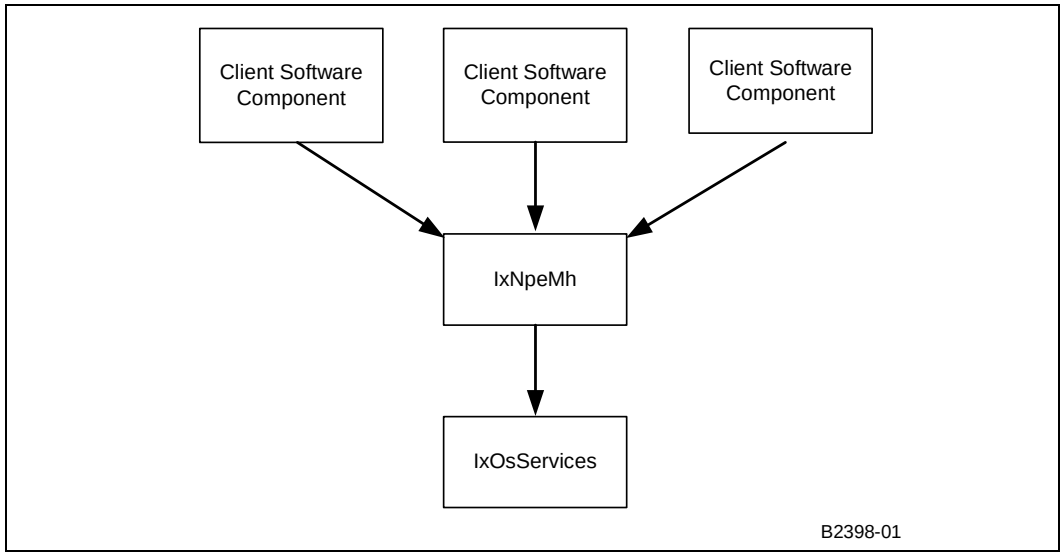
The IxNpeMh component handles messaging for the three NPEs independently. A problem or delay in interacting with one NPE will not impact interaction with the other NPEs.

16.7 Dependencies

The IxNpeMh component's dependencies (as shown in Figure 65) 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 65. ixNpeMh Component Dependencies



16.8 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.

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

This chapter describes the Intel® IXP400 Software v1.4's "Performance Profiling API" access-layer component.

17.1 What's New

The following changes and enhancements were made to this component in software release 1.4:

- Support for loading of symbols was added, making it more convenient to determine the function located at a given program counter location.
- Result output for the Intel XScale core's PMU Time and Event sampling functions has been improved and the ixPerfProfAccXscalePmuTimeSampStop() and ixPerfProfAccXscalePmuEventCountStop() functions now write results into a file.
- The IxPerfProfAcc component is now supported in little-endian mode.

17.2 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. For the Event and Time sampling features of the Intel XScale core's PMU, the results will also be printed to an output file.

17.3 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 half 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.

17.3.1 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.

17.4 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 eight 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.

17.5 Idle-Cycle Counter Utilities ('Xcycle')

The idle-cycle counter utilities (called "Xcycle," in this document) calculate the percentage of cycles that have been idle (not performing any processing) for a period of time.

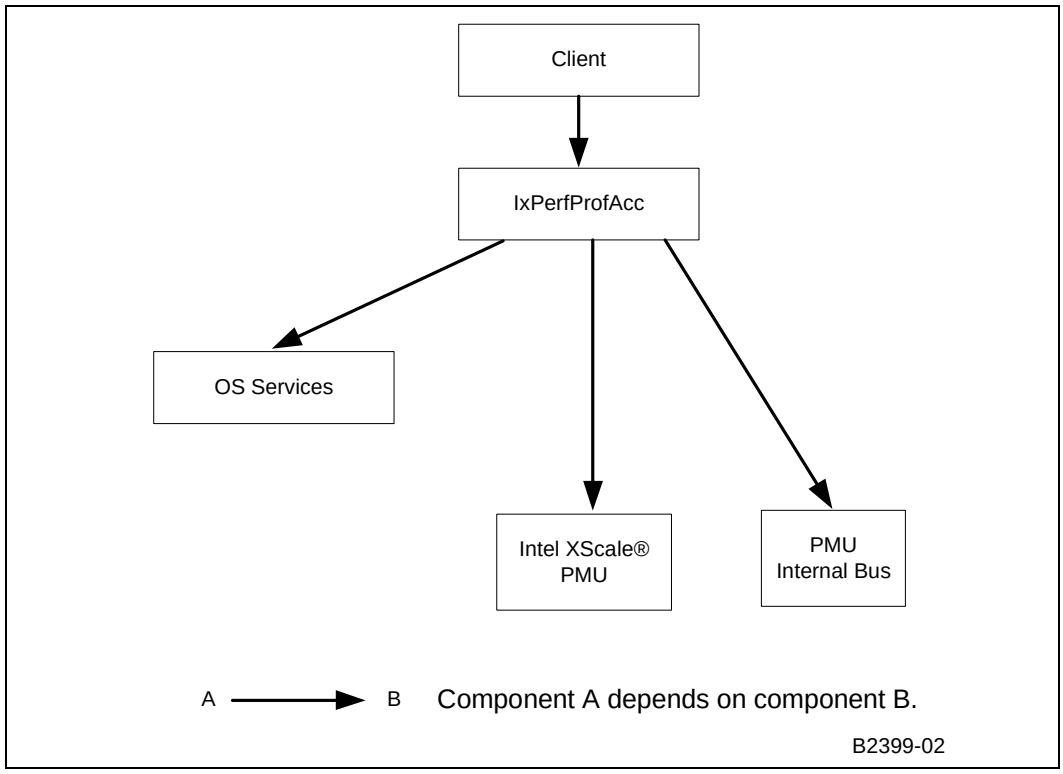
The client needs to calibrate the program by running *ixPerfProfAccXcycleBaselineRun*() when system is under low utilization. The client then starts the program it wants to measure. The *ixPerfProfAccXcycleStart*() API kicks off the idle cycle measurements. The client can select continuous Xcycle calculations, 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.

17.6 Dependencies

Figure 66 shows the functional dependencies of the IxPerfProfAcc component.

Figure 66. 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.

17.7 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.

17.8 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.

17.9 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.

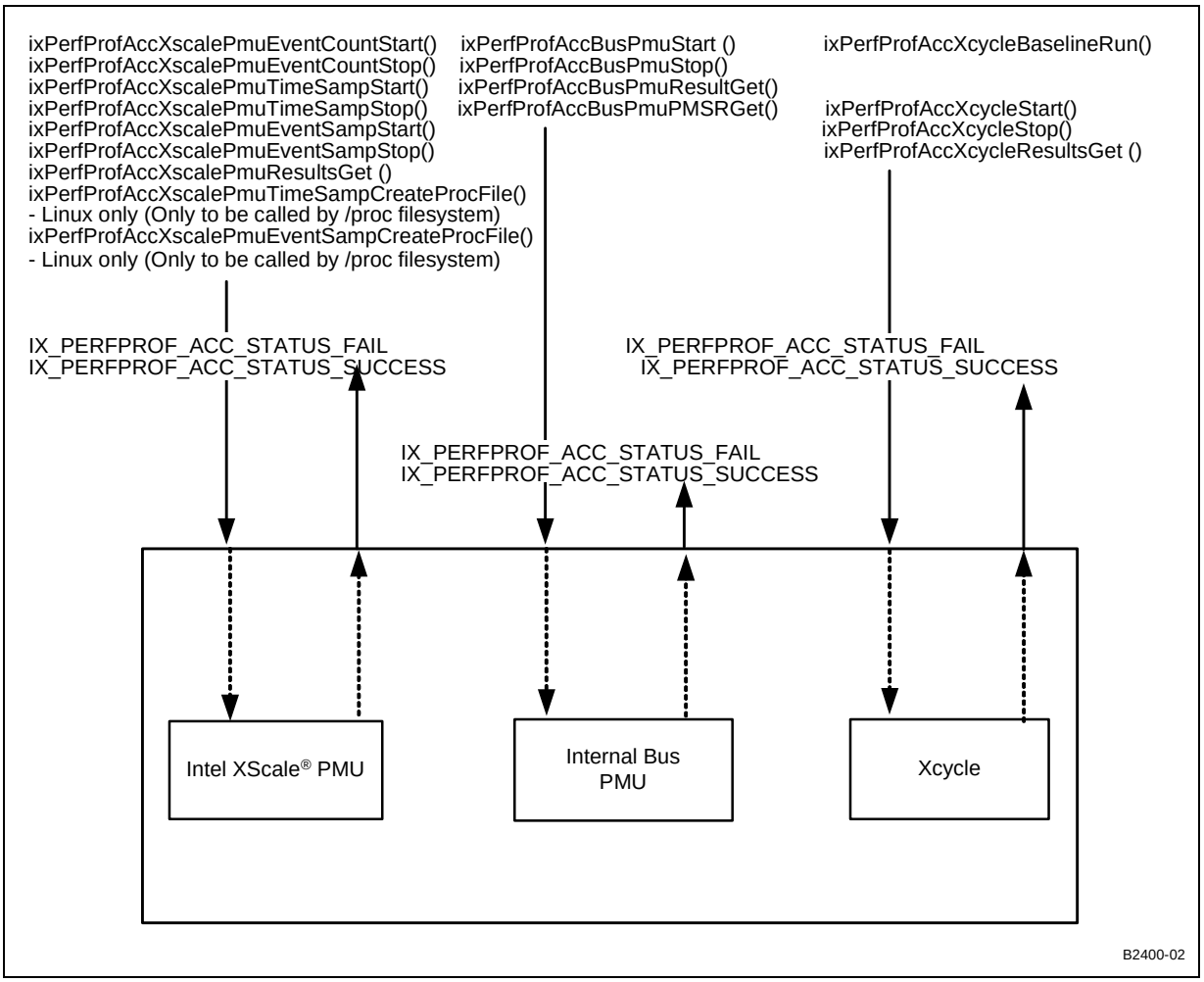
17.10 Using the API

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 67 shows all of the APIs.

Figure 67. IxPerfProfAcc Component API



17.10.1 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.

17.10.1.1 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 zero to four 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 68.

Figure 68. 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 68.

17.10.1.2 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 results are presented to the calling function in a sorted form from the pc address with the highest frequency to the pc address with the lowest frequency of hits.

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 32 bits 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 69. 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. 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.

The results for time sampling are also automatically written to a file when the "Stop" functions for these features are called. For vxWorks, this file is stored in the location pointed by the FTP server where the image for the system is downloaded from. In Linux, the file is stored in the /proc filesystem. As this filesystem is temporary, the user is required to copy the output file into a permanent location or else the results will be lost when a new round of sampling is done or when the system is stopped or rebooted. Sample file output for vxWorks is as follows:

Hits	Percent	PC Address	Symbol Address	Offset	ClosestRoutine
65451	99.8718	49a88	49914	174	reschedule
14	0.0214	47938	47924	14	intUnlock
10	0.0153	49a84	49914	170	reschedule
10	0.0153	49a8c	49914	178	reschedule
1	0.0015	54ab8	54ab8	0	__div32

Linux output is identical with the exception of the Percent column. In Linux, the user is also able to change the accuracy of matching the PC Address to the Symbol Address. The greater the accuracy required, the longer it takes to find a match. The recommended accuracy is 0xffff which means the module will reduce the PC Address by up to 0xffff until it can find a match. Else, a message is logged and "No symbol found" is written to the file. The accuracy can be changed by modifying the #define IX_PERFPROF_ACC_XSCALE_PMU_SYMBOL_ACCURACY.

The output filename is defined in IxPerfProfAcc. To use a different filename, the user is required to change the filename in the stop function for vxWorks or the ixPerfProfAccXscalePmuTimeSampCreateProcFile() function in Linux.

Note: The Linux proc file create API is declared public so that it can be called by the `/proc` file sytem. It should never be called directly by the user.

17.10.1.3 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*(). The results are presented to the calling function in a sorted form from the PC address with the highest frequency to the PC address with the lowest frequency of hits.

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

The results for event sampling are also automatically written to a file when the "Stop" functions for these features are called. For vxWorks, this file is stored in the location pointed by the FTP server where the image for the system is downloaded from. In Linux, the file is stored in the `/proc` file system.

As this file system is temporary, the user is required to copy the output file into a permanent location or else the results will be lost when a new round of sampling is done or when the system is stopped or rebooted.

Sample file output for Linux is as follows:

```
Total Number of Samples for Event1 = 0

Hits    PC Address Symbol Address Offset Routine
------- ---------- -------------- ------ ------------------

Total Number of Samples for Event2 = 0

Hits    PC Address Symbol Address Offset Routine
------- ---------- -------------- ------ ------------------

Total Number of Samples for Event3 = 65535

Hits       PC Address     Symbol Address  Offset  Routine
-------    -----------    -----------    ------  ----------------------------------------
21814      c004dd38       c004dac4       274     schedule [Module - kernel]
21381      c0058cac       c0058c7c       30      add_timer [Module - kernel]
21329      c00594a4       c005949c       8       schedule_timeout [Module - kernel]
189        c0058c84       c0058c7c       8       add_timer [Module - kernel]
124        c0059520       c005949c       84      schedule_timeout [Module - kernel]
95         c00594a8       c005949c       c       schedule_timeout [Module - kernel]

Total Number of Samples for Event4 = 6

Hits  PC Address  Symbol      Address Offset    Routine
----- ----------- -------     ---------------   -------------------
5     c004dd38    c004dac4    274               schedule [Module - kernel]
1     c0112788    0 c0112788  No lower symbol found. [Module - kernel]
```

VxWorks output is identical, but also includes a Percent column. In Linux, the user is also able to change the accuracy of matching the PC Address to the Symbol Address. The greater the accuracy required, the longer it takes to find a match. The recommended accuracy is 0xffff which means the module will reduce the PC Address by up to 0xffff until it can find a match. Else, a message is logged and “No symbol found” is written to the file. The accuracy can be changed by modifying the #define IX_PERFPROF_ACC_XSCALE_PMU_SYMBOL_ACCURACY.

The output filename is defined in IxPerfProfAcc. To use a different filename, the user is required to change the filename in the stop function for vxWorks or the ixPerfProfAccXscalePmuTimeSampCreateProcFile() function in Linux.

Note: The Linux `proc` file create API is declared public so that it can be called by the `/proc` file sytem. It should never be called directly by the user.

17.10.1.4 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
```

17.10.2 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 hexadecimal.

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

17.10.2.1 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.

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 PECs.

```
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%
```

17.10.3 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 70.

Figure 70. 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 v1.4's "Queue Manager API" access-layer component.

18.1 What's New

There are no changes or enhancements to this component in software release 1.4.

18.2 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.

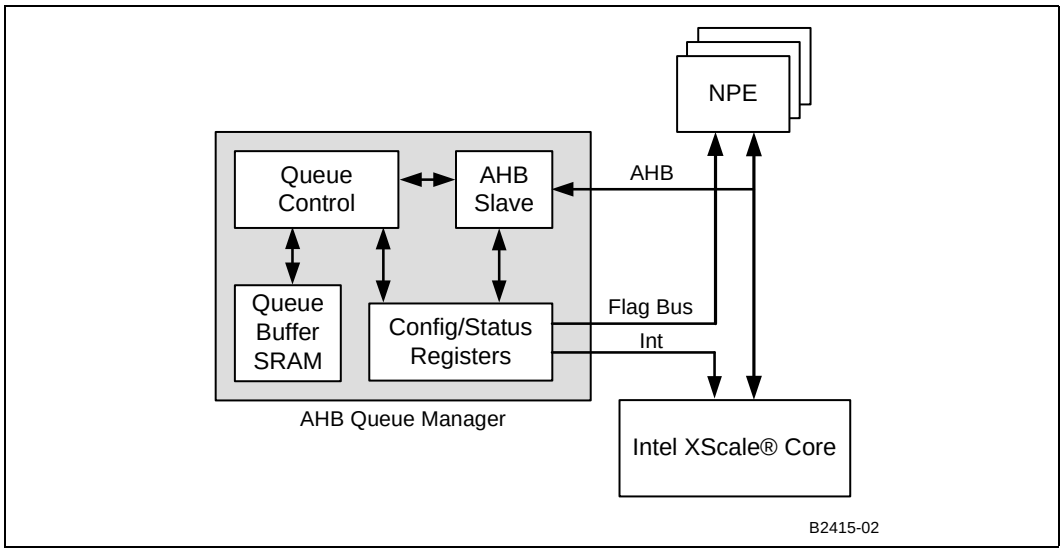
This chapter contains the necessary steps to start the IxQMgr component. Additionally, information has been included about how the AHB Queue Manager functions from a high-level view. The IxQMgr component acts a pseudo service layer to other access components such as IxEthAcc.

In the sections that describe how the queue manager works, the "client" is an access component such as IxEthAcc. 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 IxEthAcc, IxEthAcc would make the appropriate function calls to the IxQMgr component.

18.3 Features and Hardware Interface

Figure 71. 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.

18.4 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 73 on page 222. 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.

18.5 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.

18.6 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
```

18.7 Configuration Values

Table 31 details the attributes of a queue that can be configured and the possible values that these attributes can take on (word = 32 bits).

Table 31. 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

18.8 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 72. AHB Queue Manager Dispatcher Operation

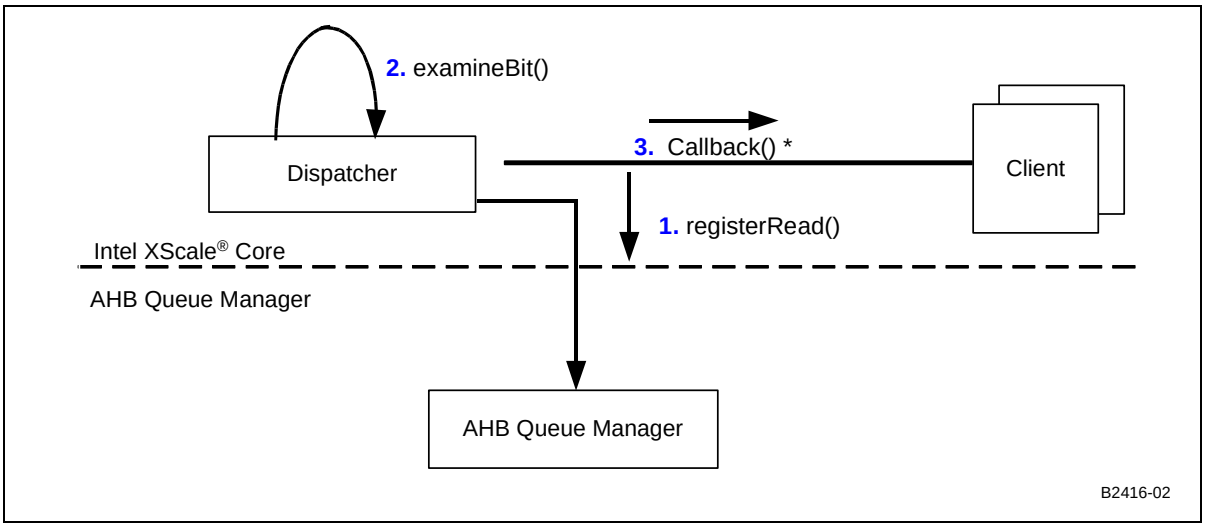


Figure 72 shows the operation of the dispatcher called in the context of an ISR. The interrupt register indicates which queues fired the interrupt. Each bit in this register is examined and, if set, the dispatcher will call each callback registered with this queue.

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 73. 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 73 is taken from the ixEthAcc codelet.

18.9 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.

18.10 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 73.

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

This chapter describes the Intel® IXP400 Software v1.4's "UART-Access API" access-layer component.

19.1 What's New

There are no changes or enhancements to this component in software release 1.4.

19.2 Overview

The UARTs of the Intel® IXP42X Product Line of Network Processors and IXC1100 Control Plane Processor' 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 and IXC1100 control plane processors, the same low-level routines will be used by both. The default configuration for IXP42X product line and IXC1100 control plane 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.

19.3 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

19.4 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.

19.4.1 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 and IXC1100 control plane 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).

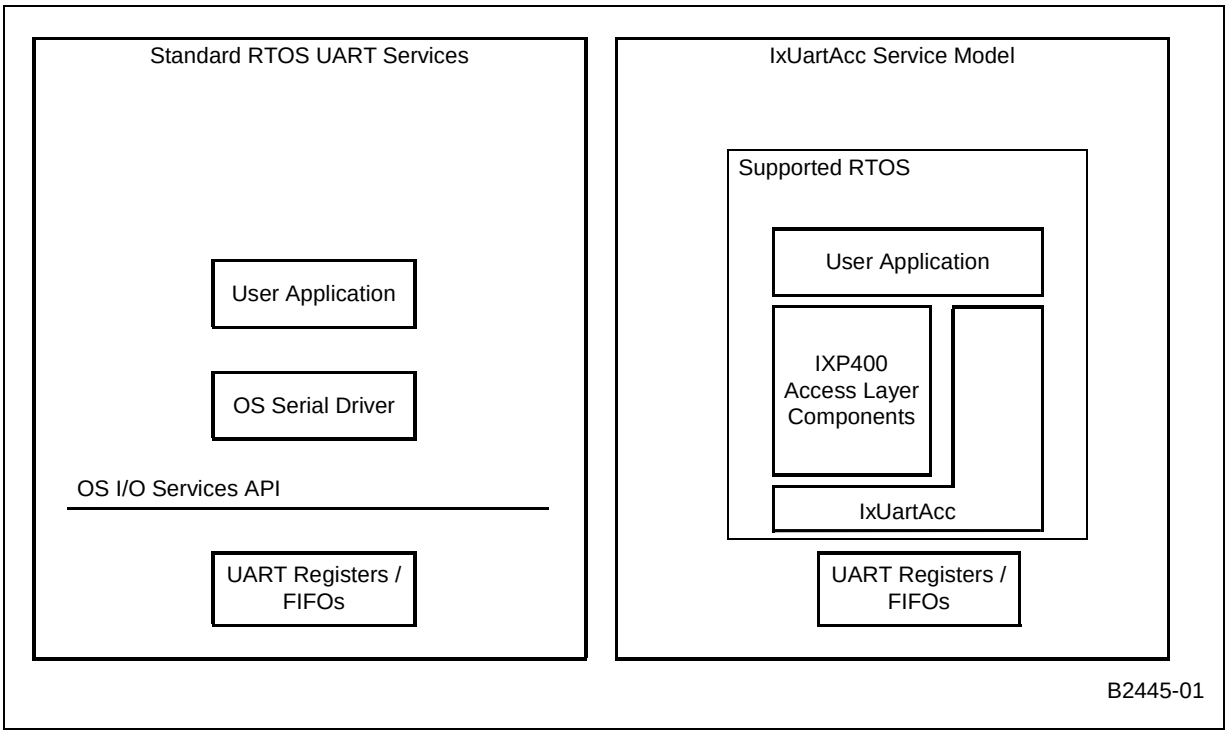
Interrupts can occur in one of two ways. One is when the FIFO has reached its programmed trigger level (set by the FIFO Control Register [FCR]). The other is when a character timeout has occurred (also set in the FCR). The driver will implement both modes of operation.

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)

19.5 Dependencies

Figure 74. UART Services Models



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

This chapter describes the Intel® IXP400 Software v1.4's "USB Access API" access-layer component.

20.1 What's New

There are no changes or enhancements to this component in software release 1.4.

20.2 Overview

The Intel® IXP42X Product Line of Network Processors and IXC1100 Control Plane Processor' USB hardware components comply with the 1.1 version of the Universal Serial Bus (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 Systems.

20.3 USB Controller Background

The IXP42X product line and IXC1100 control plane processors' 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
- Control
- Interrupt
- 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
- Address assignment
- Endpoint configuration
- Bus enumeration
- Disconnect

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.

20.3.1 Packet Formats

USB supports four packet types:

- Token
- Data
- Handshake
- 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 32. 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 33. The presence of SOF packets every 1ms prevents the UDC from going into suspend mode.

Table 33. 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 34. 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 34. 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 35 shows the format of a handshake packet.

Table 35. Handshake Packet Format

8 Bits	8 Bits
Sync	PID

20.3.2 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 36. (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 36. 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 1 ms, 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 37.

Table 37. 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 38 shows the four possible types of control transactions.

Table 38. 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 39. 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 40 shows the eight possible types of interrupt transactions.

Table 40. 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**.

20.4 ixUSB API Interfaces

Table 41. 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.

20.4.1 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 Figure 75 is passed to the setup-callback function so the software can be configured properly.

Figure 75. USBSetupPacket

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

Table 42 shows a summary of the setup device requests.

Table 42. 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 42. 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.

20.4.1.1 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.

20.4.1.2 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).

20.4.2 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.

20.4.3 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

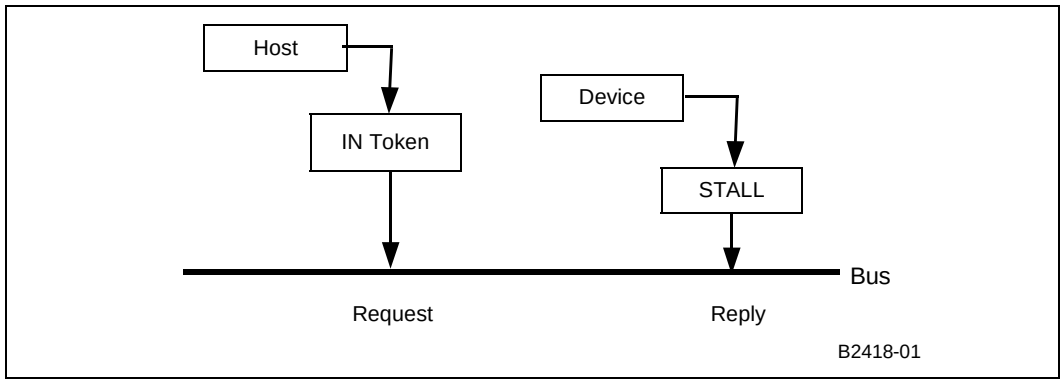
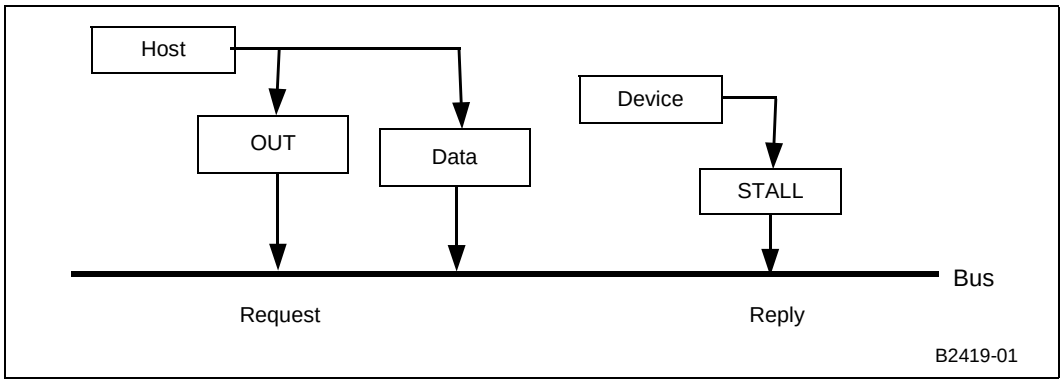


Figure 77. STALL on OUT Transactions



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.

20.4.4 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 43. Detailed Error Codes

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

/* error due to unknown reasons */

#define IX_USB_ERROR(IX_USB_ERROR_BASE + 0)

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

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

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

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

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

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

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

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

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

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

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

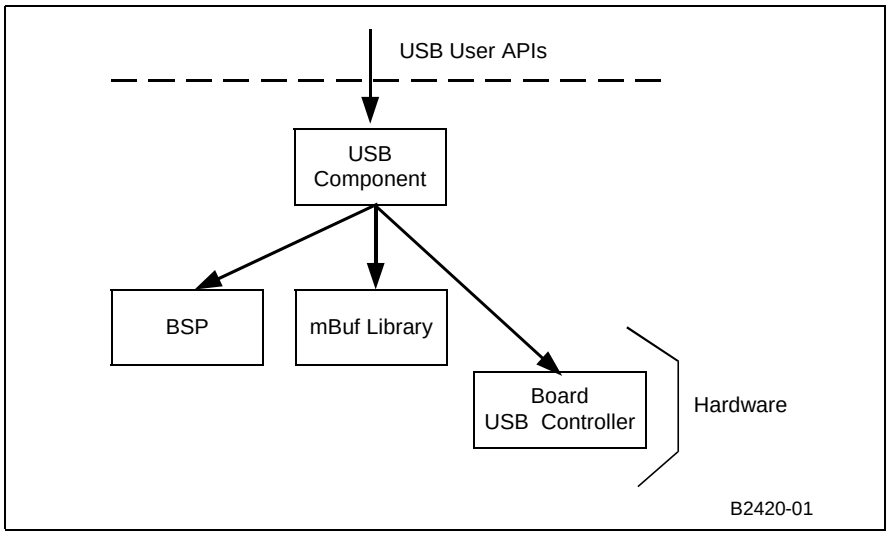
20.5 USB Data Flow

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.

20.6 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 v1.4 codelets.

21.1 What's New

The following changes and enhancements were made to this component in software release 1.4:

- IxFpathAccCodelet was removed as support for the IxFpathAcc Access Layer Component has been discontinued.
- Little-endian support has been added the IxAtmCodelet, IxPerfProfAccCodelet, and IxEthAal5Codelet.
- IxCryptoAccCodelet has been enhanced to demonstrate the new WEP services of IxCryptoAcc.
- IxPerfProfAccCodlet has been enhanced in several ways
  - The codelet now operates in little-endian mode.
  - The underlying IxPerfProfAcc component now supports symbol loading support, expanded task listings, and supports storing results to a file. The codelet has been enhanced to use these new capabilities.

21.2 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, see the header and source files provided with software release 1.4 in the xscale_sw/src/codelets directory.

21.3 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.

IxAtmCodelet makes use of the following access-layer components:

- IxAtmdAcc
- IxAtmm
- IxAtmSch

21.4 Crypto Access Codelet (IxCryptoAccCodelet)

This codelet demonstrates how to use the IxCrypto access layer component and the underlying security features in the Intel® IXP42X Product Line of Network Processors and IXC1100 Control Plane Processor. IxCryptoAccCodelet runs through the scenarios of initializing the NPEs and Queue Manager, context registration, and performing a variety of encryption (3DES, AES, ARC4), decryption, and authentication (SHA1, MD5) operations. This codelet demonstrates both IPSec and WEP service types.

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

21.5 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.

21.6 Ethernet AAL-5 Codelet (IxEthAal5App)

IxEthAal5App codelet is a mini-bridge application which bridges traffic between Ethernet and UTOPIA ports or Ethernet and an ADSL port. Two Ethernet ports and up to eight UTOPIA ports 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.

IxEthAal5App makes use of the following access layer components:

- IxEthAcc
- IxAtmdAcc
- IxAtmm
- IxAtmSch
- IxQMgr

21.7 Ethernet Access Codelet (IxEthAccCodelet)

This codelet demonstrates both Ethernet data and control plane services and Ethernet management services. The features can be selectively executed at run-time via the menu interface of the codelet.

- 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.

21.8 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.

21.10 Timers Codelet (IxTimersCodelet)

The Timers Codelet is a utility that provides support for enable, triggering and disabling timers. The timers defined in this codelet are two general purpose timers, a watch-dog timer, a time-stamp timer, and an Intel XScale core PMU timer.

21.11 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 and IXC1100 control plane processors running VxWorks to communicate IP traffic over USB to a Microsoft* Windows* system.

Operating System Abstraction Layers22

22.1 What's New

The following changes and enhancements were made to this component in software release 1.4:

- The ixOsServYield function was added to the IxOsServices API

22.2 Overview

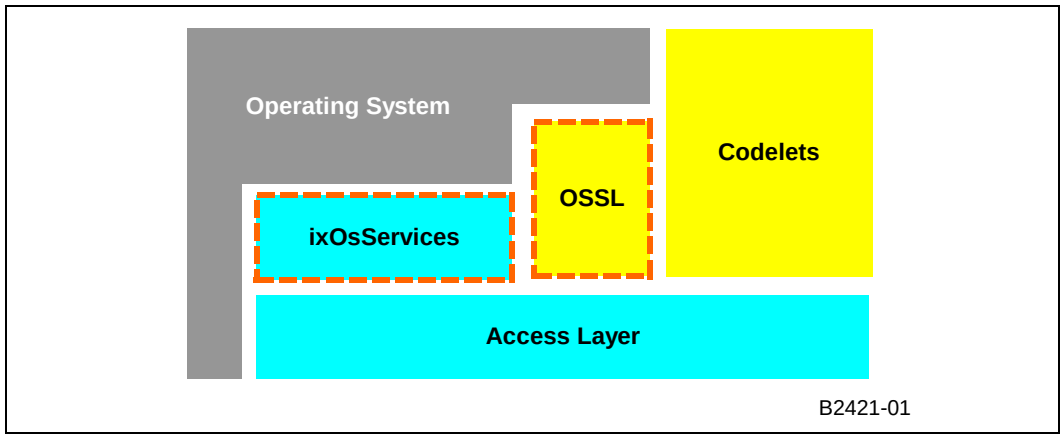
There are two operating-system abstraction layers provided as part of the Intel® IXP400 Software v1.4 architecture.

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. Though primarily intended for use by the software release 1.4 access layer, these services are also made available to the codelets and to application layer software.

The Operating System Service Layer (OSSL) defines an extended, more fully featured interface for operating system services.

There is some overlap in the functionality provided in the OsServices layer and OSSL layer. Where possible, it is recommended that the software release 1.4 developer use an OsServices layer API as opposed to a corresponding, similar OSSL function.

Figure 79. Intel® IXP400 Software v1.4 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.

22.3 IxOsServices and Related APIs

A primary focus of the IXP400 software access-layer component development was to make the APIs as OS-independent as possible. This OS-independent abstraction layer is represented by several APIs from the header files listed below:

- IxOServices.h — Contains the API to the functions that would require porting to a particular OS.
- IxOsBuffMgt.h — Includes the OS-dependant mbuf header files.
- IxOsBuffLib.h — Contains the mbuf pool initialization entry point.
- IxOsCacheMMU.h — Configures the memory management unit, performs address translations, and manages caches entries.
- IxOsServicesEndianess.h — Determines system hardware endianess requirements
- IxOsServicesMemAccess.h — Defines macros for memory mapped I/O access.
- IxOsServicesMemMap.h — Abstracts memory mapping functions for different OSes
- IxOsServicesComponents.h — Defines access coherency for software modules, when used in little-endian mode. Defines dynamic or static I/O mapping.

Information about the IxOsServices API is further described in this section. For information on other related APIs, refer to the source code documentation. (For information on this documentation, see "Access Library Source Code Documentation" on page 27.)

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" Mutex—** This will use the Intel XScale core 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 APB peripheral clock (66Mhz).

22.3.5 IxOsServices Functions

The ixOsServices layer implements 17 functions. Basic ixOsServices provided are shown in Table 44.

Table 44. Intel® IXP400 Software v1.4 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.

Table 44. Intel® IXP400 Software v1.4 ixOsServices Function Descriptions (Sheet 2 of 2)

	Function	Description
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.
Thread	ixOsServYield	Yield execution of the current thread.

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. A mutex is also sometimes known as a “binary semaphore.”

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 45.

Table 45. Intel® IXP400 Software v1.4 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 46. Access Layer Component OS Service Dependencies (Sheet 1 of 2)

		Access Layer Components															
		ixNpeMh	ixNpeDl	ixQmgr	ixAtmdAcc	ixHssAcc	ixEthAcc	ixEthDB	ixEthMii	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						
	ixOsServMutexLock	X		X	X	X	X	X									
	ixOsServMutexUnlock	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											
	ixOsServSleep	X										X					
	ixOsServTaskSleep						X	X	X	X			X				
	ixOsServTimestampGet																
	ixOsServCacheDmaAlloc				X	X		X			X						
	ixOsServCacheDmaFree																
	ixOsServYield																
	IX_OSSERV_MEM_MAP	X	X	X			X						X				X
	IX_OSSERV_MEM_UNMAP	X		X			X						X				

Table 46. Access Layer Component OS Service Dependencies (Sheet 2 of 2)

		Access Layer Components														
		ixNpeMh	ixNpeDl	ixQmgr	ixAtmdAcc	ixHssAcc	ixEthAcc	ixEthDB	ixEthMii	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 Intel® IXDP425 / IXCDP1100 Development 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.

23.1 What's New

The following changes and enhancements have been implemented since software release 1.4:

- The driver now supports little-endian mode

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 and IXC1100 control plane processors' expansion bus
- The ATM UTOPIA data path interface — via the IXP42X product line and IXC1100 control plane processors' 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.

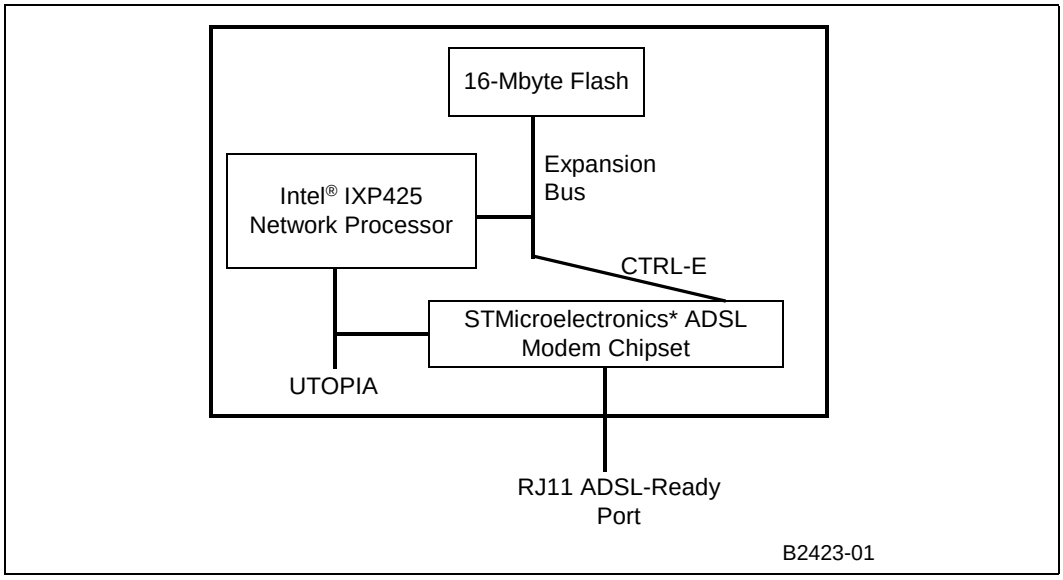
There is very little configuration information required to open an ATU-R line. Almost all line configuration parameters are supplied by the ATU-C side.

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. APIs are provided in the following areas:

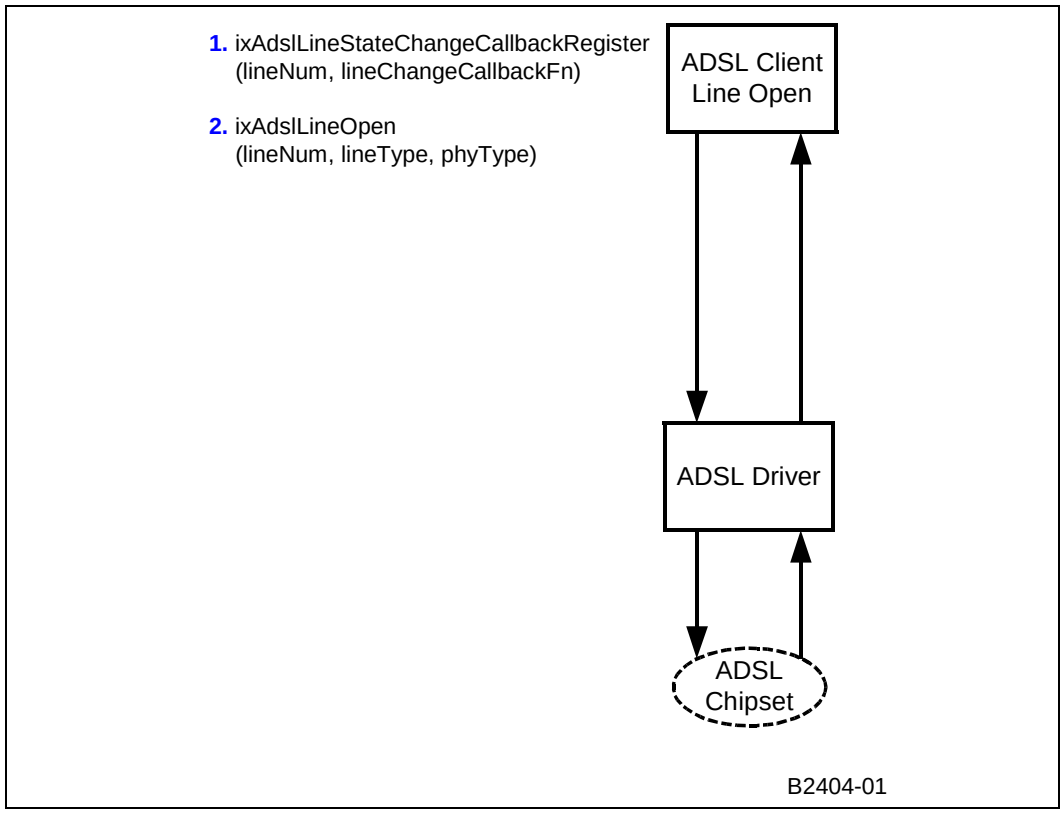
- Firmware download to the ADSL chipset
- Initialization of the ADSL devices
- Opening, closing and monitoring an ADSL line.
- Soft reset

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 255 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 only supports the ATU-R mode of operation.
- The driver can operate in single PHY mode only.