Intel® Internet Exchange Architecture Portability Framework

Developer's Manual

November 2003

Document Number: 278662-005

Contents

Figures

Tables

Revision History

Date	Revision	Description
May 2002	001	SDK 3.0 Pre-Release 3
August 2002	002	SDK 3.0 Pre-Release 4
October 2002	003	SDK 3.0 Pre-Release 5
February 2003	004	SDK 3.0 Pre-Release 6
April 2003	005	SDK 3.0 Release 6 FCS Removed chapters describing the Optimized Data Plane Libraries and added a pointer to the the Software Reference Manual (part of the IXA SDK Tools Release). Removed chapter describing the OSSL Libraries and added a pointer to the Software Reference Manual (part of the IXA SDK Tools Release).
July 2003	006	SDK 3.1 Release Added new chapter for TCAM Lookup Libraries.
November 2003	007	SDK 3.5 Release In Dispatch Loop chapter: • Added new APIs for handling extended meta data. In Resource Manager chapter: • Added new APIs for Microengine Services and Debug Support. • Significantly modified Services API to include 12 new APIs. Added Appendix A describing Memory and CPU Usage. Changed multiple mentions of “Intel XScale™ core” to “Intel XScale® core” due to trademark registration.

Introduction 1

1.1 About this Document

This Developer's Manual introduces you to the Intel Exchange Architecture (IXA) Portability Framework, which is a part of the IXA Software Development Kit (IXA SDK). This manual provides guidelines for using the Portability Framework to develop applications. This is a companion guide to the *Intel® Internet Exchange Architecture Portability Framework Reference Manual*, which provides details on functions and parameters contained in the Portability Framework libraries.

The IXA Portability Framework is a network application framework and infrastructure for writing modular and portable code, which:

- Save time by providing robust infrastructure software and APIs
- Save time by providing re-configurable building blocks
- Permit portability across IXA network processors
- Provide an ideal structure for third-party plug-in application modules

1.2 Audience

This guide is intended for software developers who will design, develop, and deliver network applications that must process packets at high speed. It assumes that you are familiar with the following:

- C Programming
- Realtime network applications

1.3 In This Manual

This manual includes the following chapters:

- Chapter 1, "Introduction," (this chapter) presents the organization of this manual and a brief overview of the IXA Portability Framework.
- Chapter 2, "IXA Portability Framework Overview," provides a brief overview of the IXA Portability Framework components.
- Chapter 3, "Microengine Programming Models," discusses two types of programming models and their advantages and disadvantages.
- Chapter 4, "Microblocks," describes the architecture and function of microblocks.
- Chapter 5, "Dispatch Loop," explains the function of a dispatch loop.
- Chapter 6, "Optimized Data Plane Libraries Support," introduces the optimized data plane libraries and provides a cross-reference to further documentation.

- Chapter 7, “Resource Manager,”RM introduces the different components of the Resource Manager.
- Chapter 8, “Core Components,” provides an overview of the core components.
- Chapter 9, “TCAM Lookup Libraries,” describes libraries used for managing and searching tables on the Intel XScale® core and on the microengines for Intel® IXP2400 and IXP2800 Network Processors.
- Chapter 10, “Core Component Infrastructure,” introduces the infrastructure APIs used by core components.
- Chapter 11, “Operating System Service Layer (OSSL) Support,” introduces APIs that provide portability across the operating systems and provides a cross-reference to further documentation.
- Chapter 12, “Intel XScale® Core Support,” describes two Intel XScale® core support libraries, one supporting microengine microcode loading and the other providing a hardware-independent services layer.
- Chapter 13, “Control Plane PDK,” explains the high-level architecture of the Control Plane Platform Development Kit (CP PDK).
- Appendix A, “Framework Memory and CPU Usage Summary,” lists code size and memory consumption data for key components of the portability framework.
- Appendix B, “Glossary,” defines key terms used in the IXA SDK documentation set.

1.4 Other Sources of Information

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

- *Intel® Internet Exchange Architecture Software Development Kit Software Framework Getting Started Guide*
- *Intel® Internet Exchange Architecture Portability Framework Reference Manual*
- *Intel® Internet Exchange Architecture Software Building Blocks Developer’s Manual*
- *Intel® Internet Exchange Architecture Software Building Blocks Reference Manual*
- *Intel® Internet Exchange Architecture Software Building Blocks Applications Design Guide*

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

- *Intel® Internet Exchange Architecture (IXA) Software Reference Manual*
- *Intel® Internet Exchange Architecture Optimized Data Plane Libraries Reference Manual*
- *IXP2400/IXP2800 Development Tools User’s Guide*
- *Help Topics: Developer Workbench*
- *Intel® IXP2400/IXP2800 Network Processor Microengine C Compiler Language Support Reference*
- *Intel® IXP2400/IXP2800 Network Processor Microengine C Compiler LIBC Reference*
- *Intel® IXP2400/IXP2800 Network Processor Programmer’s Reference Manual*

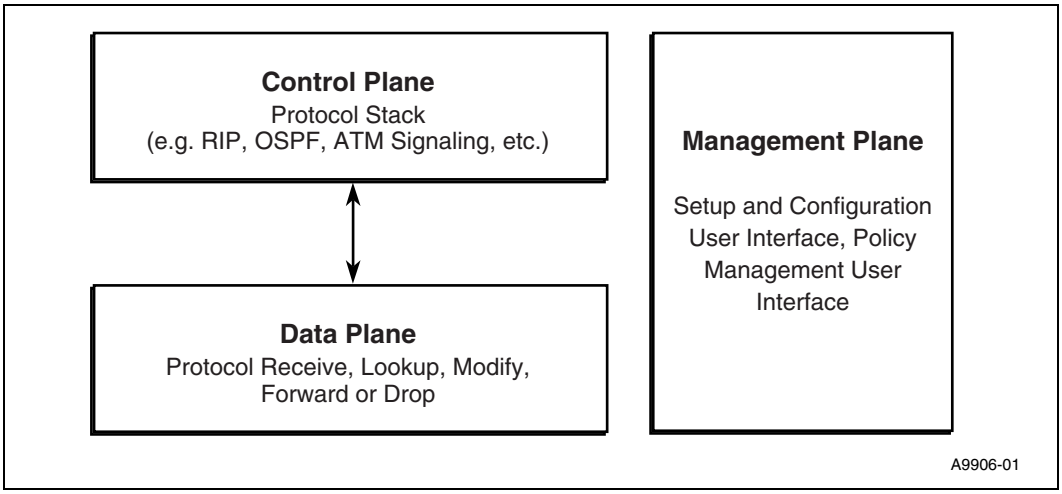
- *Intel® IXP2800 Network Processor Hardware Reference Manual*
- *Intel® IXP2400 Network Processor Hardware Reference Manual*

IXA Portability Framework Overview 2

2.1 Network Application Structure

As shown in Figure 2-1, a networking application typically operates on three logical planes. This document focuses primarily on writing data plane components utilizing MEv2 microengines on the IXP2400 Network Processor and IXP2800 Network Processor.

Figure 2-1. Functional Planes of a Networking Application



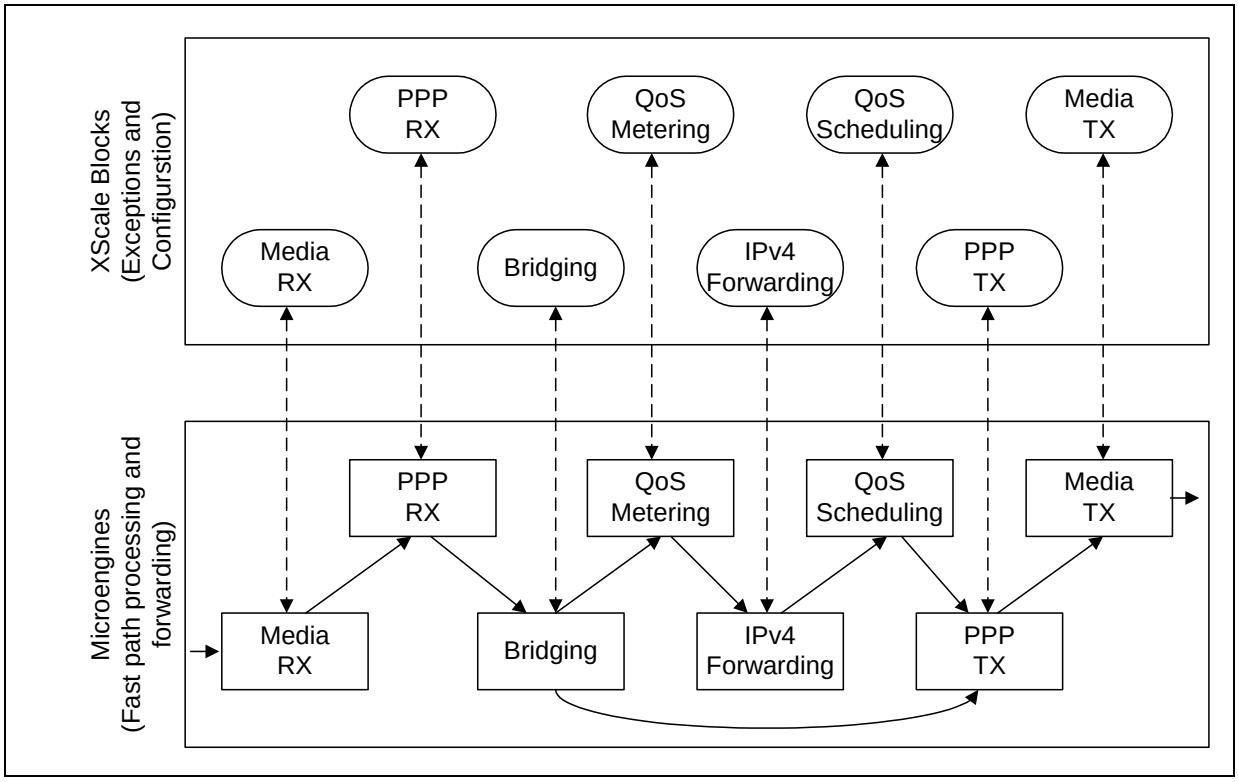
- The *data plane* processes and forwards packets at high speed. It is typically the most performance-sensitive since all packets processed by the device must pass through the data plane. For the IXP2400 Network Processor and IXP2800 Network Processor, the data plane consists of:
  - The *fast path*, which refers to the MEv2 microengines, which handle most of the packets. For example, the fast path handles the simple forwarding of IPv4 packets.
  - The *slow path*, which refers to the Intel XScale® core. This handles a few packets that cannot be handled on the fast path because of the complexity of the processing involved. These packets are called exception packets. Examples include forwarding IP packets with options in the header, handling fragmented packets, etc.
- The *control plane* handles protocol messages and is responsible for the setup, configuration, and update of tables and data sets used by the data plane for lookups. For example, the control plane processes RIP, OSPF packets containing routing information, and updates the IPv4 forwarding table used by the data plane.
- The *management plane* is responsible for system configuration, gathering and reporting statistics, and stopping or starting applications in response to user input or messages from other applications. The management plane typically implements a GUI for getting and displaying information from the user.

2.2 Data Plane Software Structure

Figure 2-2 shows software components on the data plane for a sample application. The bottom half of the figure shows code blocks running on the microengines that handle data plane processing. These code blocks are referred to as *microblocks*. To get the highest level of performance, the microengines should process as many of the packets as possible.

Once a microblock has processed a packet, it passes it to another microblock for processing. The solid arrows in Figure 2-2 represent the passing of packets. In some instances, there is only one microblock that is run after the current microblock, but in other cases it is useful for a microblock to select which block processes the packet next. In the example shown in Figure 2-2, the bridging microblock has two choices for further processing. If the packet is not bridged (it is routed) then the packet is sent to the output that is connected to the QoS metering block. If the packet is bridged, then we don't want to perform IPv4 forwarding, therefore the packet is passed directly to the PPP transmit stage. (Note: other possible ordering of the blocks can be specified if the user wants to perform QoS for bridged and routed packets.)

Figure 2-2. Example of Data Plane Software Components



An Intel XScale® core component is code that runs on the Intel XScale® core. Each of the blocks on the microengines has a Core Component associated with it. The dotted lines in Figure 2-2 show the relationship between the microblocks and the core components. A core component works in conjunction with its associated microblock to perform tasks. A core component may perform such tasks as:

- Provide a mechanism for configuring and managing the microblocks. This allows the control plane software to control the behavior of the blocks as well as get statistics and other state

information. As an example, the IPv4 forwarding Core Component provides interfaces to add and delete route entries, etc.

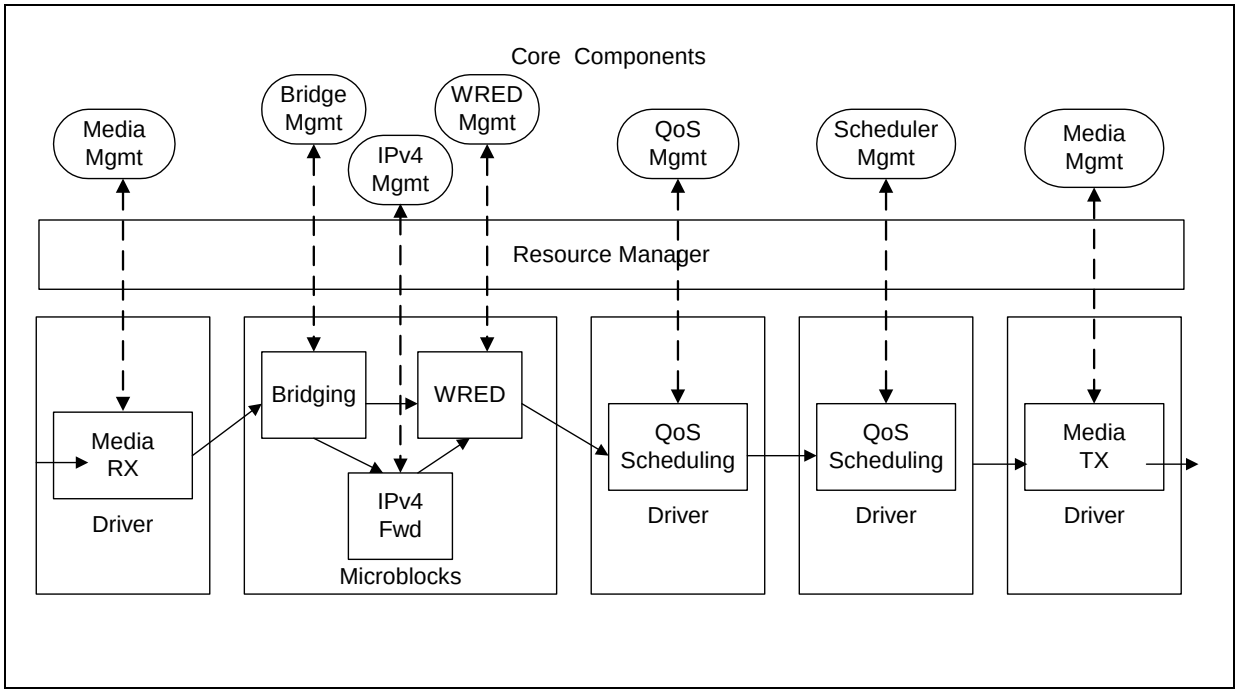
- Perform exception packet handling. Some of the microblocks may not be able to handle a packet for a variety of reasons, for example, the programmer doesn't want to dedicate the code space to rare events, the processing may be complicated and it isn't worth the effort of supporting it on the microengines, insufficient time budget, etc.

  In this case, the exception packet can be passed to the Intel XScale® core block to perform the work. For example, the IPv4 block may send packets that require fragmentation to the Intel XScale® core block instead of performing fragmentation on the microengine.

- Manage data structures that are shared between the microengine and the Intel XScale® core. These data structures may be modified in reaction to arriving network data, or modified in response to configuration requests that are made on the Intel XScale® core. For example, the IPv4 Core Component updates a shared route table in response to route update requests.

Figure 2-3 shows a more detailed representation of the data plane portion of the application. At the bottom, we see the fast path building blocks, which are either packet processing microblocks or hardware-specific driver microblocks. A dispatch loop combines packet processing microblocks running on the same microengine thread. It can be instantiated on one or more threads on multiple microengines and implements the data flow between the microblocks. Driver blocks are typically not combined with other blocks on the same thread of execution. They usually run on a separate microengine thread (or threads), which implies that a dispatch loop is not needed.

Figure 2-3. Detailed Example of Data Plane Software Components



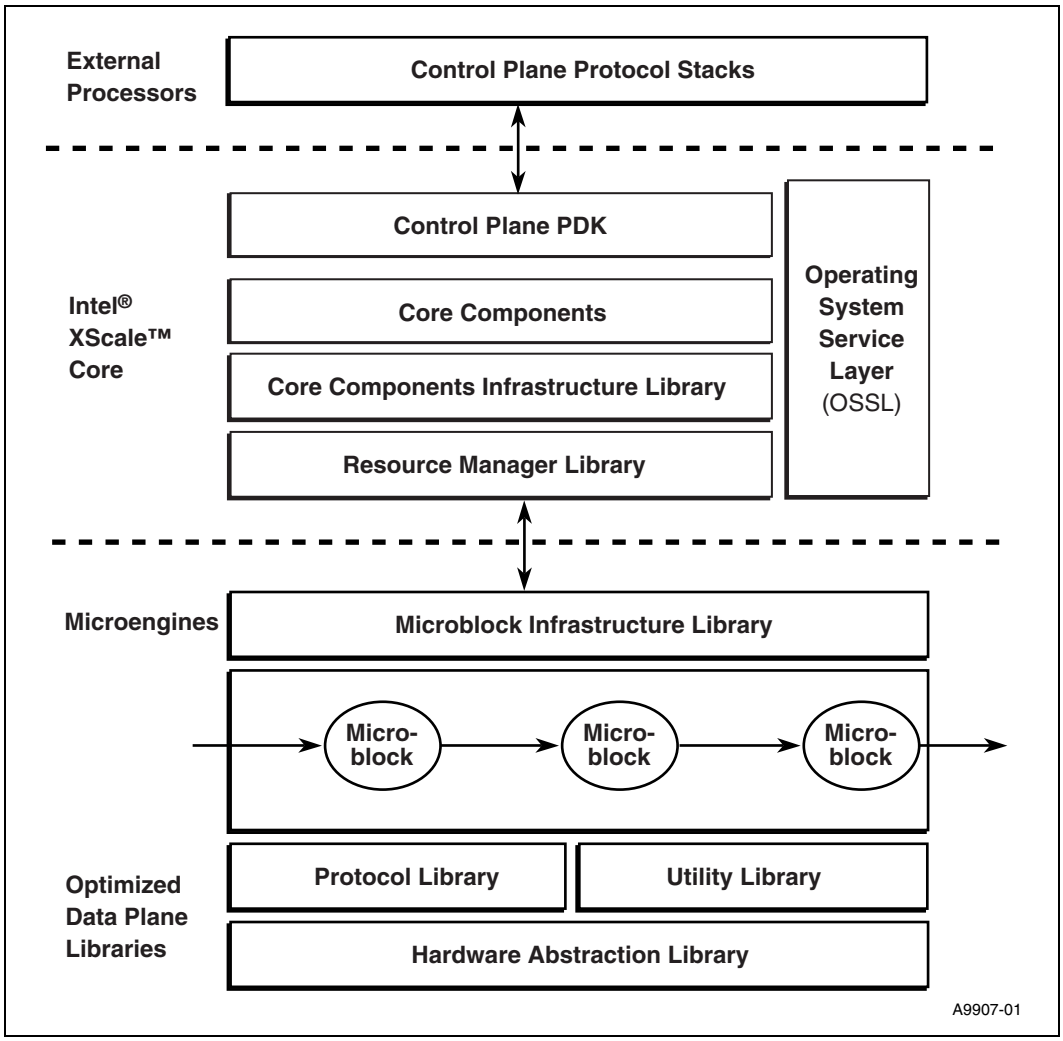
At the top of Figure 2-3, we see the Intel XScale® core components, which interface with control plane software via the Control Plane PDK. The Intel XScale® core components are written using the Resource Manager APIs and the Core Component Infrastructure Library (CCI). The Resource Manager and the CCI provides a library and support infrastructure that allows the microblock and the Intel XScale® core component to work together.

2.3 Logical Elements of an IXA Application

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

The IXA Portability Framework is implemented using a layered architecture on both the microengines and the Intel XScale® core, as shown in Figure 2-4. Using this layered architecture, you have the flexibility to use the entire IXA Portability Framework or to use it only up to a specific level. For example, you can choose to write microblocks on the microengines but use only the Resource Manager API on the Intel XScale® core.

Figure 2-4. Elements of an IXA Application



The remainder of this chapter contains a brief overview of the following components of the IXA portability framework:

2.3.1 Optimized Data Plane Libraries and Tools

The optimized data plane libraries consist of low-level macros or microengine C functions used to write microblocks or any other microengine code. These libraries are optimized for high performance and minimal code space utilization. Programs written with optimized data plane libraries are structured assembly directives or microengine C functions that facilitate the creation of software programs that are easy to read, understand, and maintain without sacrificing real-time performance. For details, see the *Intel® Internet Exchange Architecture Optimized Data Plane Libraries Reference Manual* on the IXA SDK Tools CD.

Some examples of what these libraries provide include:

- Hardware-specific functions for CAM, local memory, critical sections, etc.
- Protocol header parsing functions for IPv4, IPv6, etc.
- Utility functions for hash table lookup, CRC, etc.

2.3.2 Microblocks

The fast path processing on the microengines is divided into logical networking functions called *microblocks*. Each microblock is a macro or microengine C function written using underlying low-level libraries provided by the dispatch loop infrastructure and the optimized data plane libraries.

The intent is to allow microblocks to be written so that each microblock is independent of the others. This improves reusability and allows developers to combine microblocks in different ways to create a meaningful application. The net benefit is that the task of writing fast path code is simplified and time to market is accelerated.

A microblock has an associated management component on the Intel XScale® core. The application is typically written so that the microblock will process the most common cases and the exception cases are passed to the Intel XScale® core component for further processing.

There are two types of microblocks:

- Packet processing microblocks, which perform high-level protocol-specific functions on a packet, for example IPv4 forwarding, IPv6 forwarding, Bridging, Network Address Translation (NAT), etc. Note that as opposed to a low level macro/function (such as IP checksum), a microblock is coarse-grained, has state and associated data structures typically shared with the Intel XScale® core (for example, IPv4 forwarding).
- Driver microblocks, which are hardware- or media-specific code blocks which may be implemented in hardware in future revisions of the IXP processor. Examples include Receive/Transmit blocks that interact closely with the MSF interface on the IXP2xxx, the Queue Manager microblock, which manages the queuing hardware (Q-Array) etc. A developer will write software as driver microblocks when there is specific hardware that needs to be managed, (such as receive or transmit hardware) or when the software processing model doesn't match the packet-processing model (for example, schedulers that don't operate on packets, but on queues).

Chapter 4, "Microblocks," provides detailed information about writing microblocks.

2.3.3 Dispatch Loop and Microblock Infrastructure Library

A *dispatch loop* combines microblocks running in a single microengine thread into a microblock group. It is instantiated on one or more threads on multiple microengines and implements the data flow between the microblocks. The dispatch loop provides an infrastructure for microblocks to efficiently access commonly used fields in the packet descriptor and header. It also provides a mechanism to send and receive packets to and from other dispatch loops and the Intel XScale® core.

The dispatch loop is implemented using support libraries called the Microblock Infrastructure Library, which provides APIs to access the cached packet descriptor.

Chapter 5, "Dispatch Loop," in this manual provides detailed information about writing a dispatch loop. The dispatch loop API is detailed in the *Intel® Internet Exchange Architecture Portability Framework Reference Manual*.

2.3.4 Resource Manager Library

The resource manager library is a software component on the Intel XScale® core, which provides an API for:

- Hardware initialization, configuration, and resource management
- Communication between the microblocks and their associated core components

The resource manager API simplifies the task of hardware initialization, configuration, and resource sharing. It also isolates the developer from the communication details between the microengine and Intel XScale® core.

Chapter 7, "Resource Manager," provides detailed information on the Resource Manager. The Resource Manager API is detailed in the *IXA Portability Framework Reference Manual Volume I*.

2.3.5 Intel XScale® Core Components

An Intel XScale® core component implements the configuration, management and exception processing code for an associated microblock. A core component may manage more than one microblock. In the extreme case, there may be a single core component for all the microblocks. A core component performs the following functions:

- Configures its microblock (static configuration by means of imported variables and dynamic configuration through control blocks)
- Initializes and maintains common data structures that may be updated by other applications
- Provides exception as well as control message handler to process packets/messages sent by the microblock

There are two ways to implement a core component. One way is to use the IXA Core Component Infrastructure Library (see Chapter 8, "Core Components"). The core component infrastructure specifies the design of the component and the mechanisms used for configuration, and for packet and message processing.

The other way is to implement it as a software entity that directly uses the Resource Manager API (see Chapter 7, "Resource Manager"). The design of this entity (whether it is a shared library, driver, thread, process, etc.) and how it processes packets and messages is left entirely to the developer.

This provides developers with considerable flexibility in integrating applications written using the IXA Software Framework with existing legacy applications and protocol stacks running on the Intel XScale® core. If you are writing mostly new code on the Intel XScale® core, you may prefer the accelerated development time provided by the IXA Portability Framework. Customers with a substantial legacy code base may prefer to write the Intel XScale® core component using the core component infrastructure to ease integration with existing code. You can still use the microblock infrastructure on the microengines and use the Resource Manager API to interface with them.

For more information on core components, refer to Chapter 8, "Core Components". To view details on core components supplied and supported by Intel, see the *Intel® Internet Exchange Architecture Software Building Blocks Developer's Manual*.

2.3.6 Additional Intel XScale® Core Supporting Libraries

Two additional Intel XScale® core libraries are described in Chapter 12, "Intel XScale® Core Support." This chapter includes the following sections:

- Section 12.1, "Microengine Loader for the Intel XScale® Core" on page 99 describes support for loading microengine microcode from the Intel XScale® core.
- Section 12.2, "Hardware Abstraction Layer for the Intel XScale® Core" on page 99 describes support for a hardware abstraction layer (HAL) for the Intel XScale® core. The HAL generates code to interface either to the Intel XScale® core-based hardware or the Transactor. An Intel XScale® core application that uses HAL can run in hardware mode or simulation mode without changes to the code that accesses the functional units. This increases portability of the application code when moving between the simulation environment of the Transactor and the Intel XScale® core-based hardware.

2.3.7 Core Component Infrastructure

The core component infrastructure specifies the design and structure of Intel XScale® core components and provides underlying APIs and infrastructure to pass messages and packets between them. A core component written to the core component infrastructure is typically a library of code with the following entry points:

- An initialization function
- A termination function
- One or more packet handlers
- One or more message handlers

For more information, refer to Chapter 10, "Core Component Infrastructure."

2.3.8 Control Plane PDK

The Control Plane PDK provides support for interfacing Intel XScale® core components with software running on the control plane. The Control Plane PDK APIs, based on the Network Processor Forum (NPF) APIs, present a flexible and well-known programming interface.

The Control Plane PDK also makes the existence of multiple forwarding planes, as well as vendor-specific details, transparent to control-plane applications. Furthermore, the hardware properties and nature of the interconnect used between the control and the forwarding planes are isolated. Thus, the protocol stacks and network processors available from different vendors can be easily integrated with the NPF APIs.

For more information, see Chapter 13, "Control Plane PDK," or the Control Plane PDK documentation set.

2.3.9 Operating System Service Layer (OSSL)

The Operating System Services Layer (OSSL) provides an abstraction layer for all code running on the Intel XScale® core. It is used by the Resource Manager, core components infrastructure, and other code on the Intel XScale® core to enhance their portability across multiple operating systems. Application developers writing Intel XScale® core code should also use this library instead of directly using operating system-specific APIs. The OSSL provides the following type of services:

- Thread management
- Synchronization primitives
- Mutual exclusion
- Timers
- Memory management
- Message logging

2.3.10 System Application

This management application is responsible for initializing the Intel XScale® core components, loading the microengines, and performing any configuration required for startup. It may perform additional tasks as needed. The application needs to know how the microblocks are combined in the dispatch loop on the microengines. Some of the tasks performed by a system application include:

- Initializing the Resource Manager (which in turn initializes the hardware)
- Setting the microcode image using the Resource Manager APIs
- Launching the Intel XScale® core components associated with the microblocks and any other Intel XScale® core applications that are required. The application needs to wait for each of the core components to initialize, configure the microblock data structures, etc.
- Configuring the core components appropriately with an initial set of parameters
- Setting up the packet flow between the Intel XScale® core components to match the data flow in the microengines
- Writing the microcode image into the microengine microstore
- Enabling the microengines

The *Intel® Internet Exchange Architecture Software Building Blocks Developer's Manual* provides more detail on writing a System Application using the core component infrastructure and the Resource Manager.

Microengine Programming Models 3

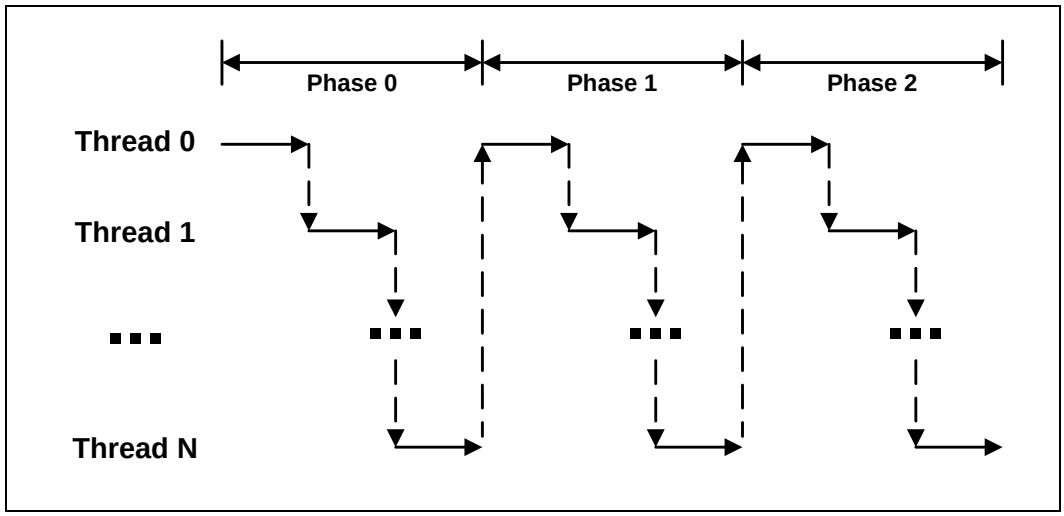
In many cases, software running on the microengines will run on more than one thread, and in some cases, many threads will run on many microengines. When writing this software, the programmer needs to be aware of the parallelism and associated challenges of writing parallel processing programs.

This chapter identifies two basic models for writing parallel processing blocks: the ordered thread model and the unordered thread model. In the ordered thread model, the programmer assumes that the threads execute in some application-defined order in critical sections. In the unordered thread model, the programmer assumes that threads can run in any arbitrary order. The differences between these two models affect how the programmer handles different problems that arise when writing applications. The following sections discuss these programming models in more detail.

3.1 Ordered Thread Model

The first programming model is where thread order is carefully controlled by the programmer. This model is also referred to as the Hypertask Chaining Model (HTC). In this model, a critical task is broken into different phases and each thread takes a turn executing a phase and then passes control to the next thread, as shown in Figure 3-1. The horizontal solid lines show the execution of instructions and the vertical, dashed lines show a signal sent from one thread to another.

Figure 3-1. Thread to Thread Synchronization Using Signals



In this model, each thread waits for a signal from the previous thread before it begins execution. When a thread receives a signal from the previous thread, it will run its task and signal the next thread. The last thread in a group will signal the first thread when it is done executing. This ensures that all threads continue to execute code in the same order.

Within each phase the programmer controls execution order. The advantage of this programming model is that it maintains packet order throughout the processing. For some packet processing tasks, packet ordering is required and this programming model simplifies the task.

Another characteristic of the ordered programming model is mutual exclusion can be handled implicitly. Because the programmer controls the order in which threads execute, then the programmer can ensure that only one thread is in the critical section at a time. This removes the need for explicit locking.

While the ordered thread model offers the above benefits, it also has some drawbacks. One potential disadvantage is when the processing varies per packet. Because of the ordering, all threads end up running at the rate of the slowest thread. Ordered processing is more appropriate when the amount of effort is roughly constant per thread.

Another challenge associated with ordered thread model comes from the ordering operations. In the Intel® IXP2400 Network Processor and Intel® IXP2800 Network Processor family, thread ordering is maintained by a form of token passing (through a signal). Consequently, each of the threads must execute each of the ordered phases even if they have no work to perform. This is because they need to consume the signal and pass it to the next thread. If a thread were to skip a phase, the token would be lost.

3.1.1 Unordered Thread Model

Using the unordered thread model, the thread ordering is not controlled as it is in the ordered thread model. Instead, threads work until their current task is done, get more work and continue. This is the programming model referred to as Pool of Threads (POTs).

This model is efficient when there is a large variation in the time necessary for each thread to complete its processing or when the latency of operations is highly variable (for example, memory, TCAM, etc.). Because each thread is working independently, processing happens at the average processing rate instead of the slowest processing rate as in the ordered thread model. The unordered model also works well when the packet arrival time varies, such as a mix of small and large packets, because the application can use the slow periods to catch up on work that piled up during the fast arrival periods.

A disadvantage of the unordered thread model is that mutual exclusion must be performed explicitly. When all threads working on the same tasks are on the same microengine, then the locking can be implemented with local resources and is not very expensive. When the threads executing the same tasks span microengines, then the locking must be through a shared resource such as memory. (It is expected that most unordered thread programs will span microengines.) This increases the cost of doing the synchronization and can make it difficult to get high performance when there are multiple cases of lock contention.

Another challenge with the unordered thread model is that any ordering constraints must be explicitly satisfied. This can lead to more overhead and lose some of the potential performance gains obtained by removing the strict ordering requirements.

3.1.2 Selecting the Appropriate Model

Typically, driver code such as Receive and Transmit blocks and Queue Manager will use the ordered threading model.

Packet processing microengines may use the ordered or the unordered model. If the ordered threaded model is chosen, then the thread ordering is typically enforced at any point in the processing where critical sections (sections of code which only one thread must execute at any given time) are present. Critical sections are always used at the beginning and end of the packet processing to get packets from the previous stage and send packets to the next stage in order. A

disadvantage of using the ordered thread model is that each block must be processed which is wasteful. The ordered thread model is also not appropriate for applications that contain loops in the packet processing.

Packet processing microengines may use the unordered model, in which case packets may get reordered during processing and the ordering will need to be restored at a later stage. The next section describes mechanisms to restore packet ordering.

3.1.3 Maintaining Packet Order

A key with a parallel processing architecture is that packet order must be maintained for packets as they are processed. For example, a simple router must not reorder packets that belong to the same flow because of its effect on end-station performance.

For applications that use the ordered thread model, packet order is maintained at the start and end of every packet processing stage using critical sections.

For the unordered thread model, the IXA Portability Framework provides an application support library that allows the developer to maintain packet order. On the Intel® IXP2400 Network Processor, this library is used in the Queue Manager microblock. The mechanics of this library are described in the following subsection.

3.1.3.1 Packet Ordering for the Unordered Thread Model

The packet-ordering problem can be subdivided into end-to-end packet order and partial order at modules. We will first describe end-to-end packet ordering support in unordered thread model and later describe support for maintaining partial order at blocks requiring packet sequencing.

3.1.3.1.1 Maintaining End-to-End Packet Order

The end-to-end packet ordering criteria is the packets of a flow should leave the system in the same order as they arrived in the system. The problem of end-to-end packet ordering is solved using an Asynchronous Insert, Synchronous Remove (AISR) array.

Every packet is assigned a sequence number before it exits the receive block. The sequence number can be globally maintained for all packets arriving in the system or it can be maintained separately for each port or flow. From now onwards, we will call the unit of classification as the sequence ID, for example, input port or flowid.

The Sequence ID must be defined at a granularity such that all packets that need to be ordered relative to each other belong to the same sequence ID. The sequence ID can be as coarse-grained as all packets belonging to one global sequence ID or it can be as fine-grained as each micro-flow (identified by tuple of src and dst IP addresses, layer 4 port numbers and layer 4 protocol) has a separate sequence ID. If you use fine-grained sequence IDs, then you end up having a lot of state to maintain and it may be expensive. On the other hand, if you have coarse-grained sequence IDs you get false blocking by trying to order the packets that need not be ordered.

The sequence number of a packet arrived in the system should be one greater than the sequence number of the previous packet of the same sequence ID arrived in the system. The sequence number wraps back to zero after Max_seq_num (size of AISR array) is reached.

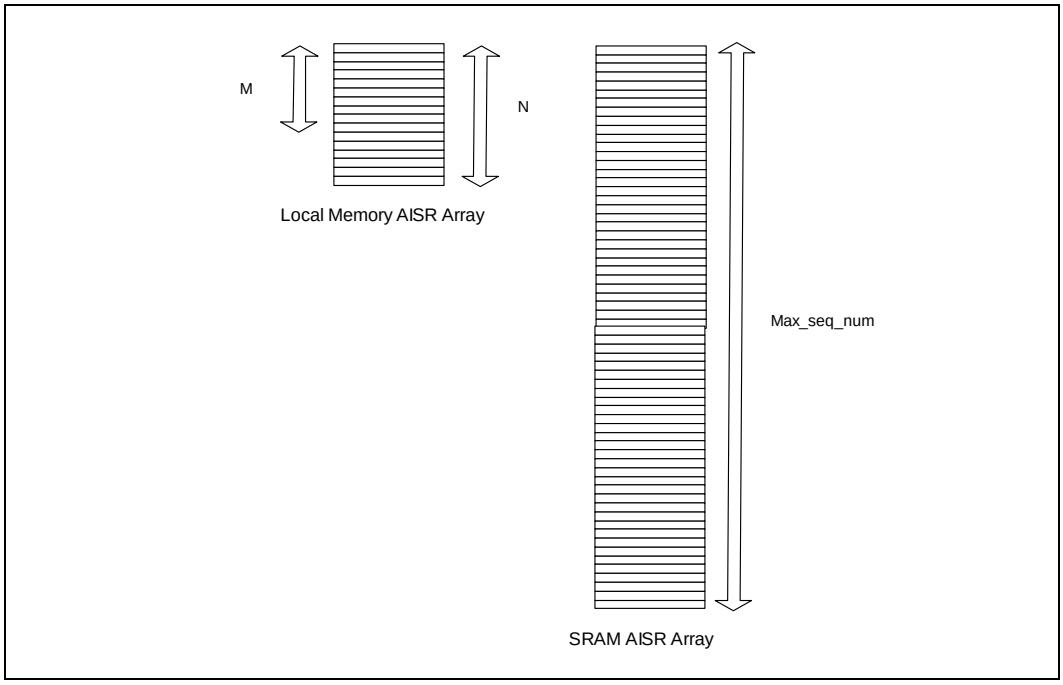
The AISR array is maintained in shared memory (such as SRAM) and is indexed by the packet sequence number. For each sequence ID, there is a separate AISR array. When the packet processing pipeline has completed the processing on a particular packet, it passes the packet to the

next stage, which we will call a reordering block. The reordering block uses the AISR array to store out-of-order packets and to pick packet in the order of the sequence number assigned. Ideally, the AISR array would be maintained in the local memory of the reordering block, however, that limits the size of the AISR array. Therefore, a small AISR array is maintained in local memory and a much larger AISR array is maintained in SRAM. Both of these AISR arrays are implemented using circular buffers as shown in Figure 3-2. When a packet has completed processing, it is sent to the reordering block (normally, by a scratch ring or next neighbor ring).

The reordering block maintains an expected_seq_num for each sequence_ID. The reordering block extracts out the sequence number and sequence_ID of the packet. If the packet's sequence number matches the expected_seq_num for the sequence_ID, it is processed as an "in-order" packet and the expected_seq_num is incremented. In the case when a sequence number is not expected_seq_num (for example, the packet is out of order), the reordering block finds out whether the packet will fit in the "local memory AISR Array" of size N entries. If the (packet's sequence num - expected_seq_num < N), then the packet's detail is stored in the "local memory AISR Array" to be processed later.

Otherwise, the packet is stored in the appropriate position in the "SRAM AISR Array". The reordering block also maintains the max_seq_num_in_SRAM, which is the maximum sequence number that has been stored in the "SRAM AISR Array". A value of -1 for max_seq_num_in_SRAM indicates no packet, that will be required to be processed later, is stored in the "SRAM AISR Array". The reordering block processes the first element in the "Local Memory AISR Array" when it becomes available. When packets from the "Local Memory AISR Array" are processed, the reordering block looks for the packet it has previously stored in the "SRAM AISR Array" to copy it to the "Local Memory AISR Array". The value for max_seq_num_in_SRAM indicates whether such a read from SRAM is necessary or not. If necessary, the entries of "SRAM AISR Array" are read using block read of B entries. Typical values for B is 4 and for N is 32.

Figure 3-2. AISR Array Implementation



The modified scheme has a two-fold benefit compared to simply maintaining the AISR table in SRAM. The reordering block does not poll while waiting for the head of the AISR array. In addition, the local memory cache can be used, for example, when there is a backing up of some packets due to their out of order arrival, the reordering block can schedule them quickly and catch up with rest of the pipeline.

The following pseudo-code shows an example for the reordering block when it receives a packet from a packet processing pipeline. The code assumes that there is only one sequence_ID.

```
Function: receive_packet()
seq_num = Extract sequence number from the packet;
if (seq_num == expected_seq_num)
    {
    process packet;
    expected_seq_num++;
    clear entry corresponding to seq_num from local memory and SRAM AISR Array;
    }
else
    {
    if (seq_num < (expected_seq_num + N))
        {
        store seq_num in corresponding local memory AISR Array;
        look_for_head();
        }
    else
        {
        store seq_num in corresponding SRAM AISR Array;
        if ( seq_num > max_seq_num_in_SRAM)
            max_seq_num_in_SRAM = seq_num;
        look_for_head();
        }
    }

Function: look_for_head()
if (entry at expected_seq_num is not NULL)
    {
    process expected_seq_num;
    expected_seq_num++;
    clear entry corresponding to seq_num from local memory and SRAM AISR Array;
    if (expected_seq_num % B == 0)
        {
        // perform block_read_if necessary
        if ((max_seq_num_in_SRAM != -1) & (max_seq_num_in_SRAM >
            (expected_seq_num + N)))_
                block read from SRAM AISR Array from (expected_seq_num + N)
                to (expected_seq_num + N + B);
        else
            max_seq_num_in_SRAM = -1;
        }
    }
```

3.1.3.2 Maintaining Partial Order per Block

In some applications, certain packet processing microblocks need to process packets in order. This is a difficult problem since such a microblock may not receive all the packets and need to order with respect to only the packets that flow through it.

Note: Partial Packet Ordering will be discussed in subequent revisions of this document.

3.2 Packet Descriptor or Metadata

The packet state is passed from one data plane block to the next via the packet metadata. Shared memory, next_neighbor registers, or a reflector operation may be used to pass metadata between blocks that execute on separate microengines. When multiple blocks run on the same microengine, the metadata is cached locally (for example, in local memory, GPRs, transfer registers, or next neighbor registers in loopback mode) in the beginning of the dispatch loop and flushed out at the end. The caching scheme has several benefits - it reduces the memory bandwidth utilization when multiple blocks need to access the same information and it allows for efficient packing of data useful for individual blocks without the burden of wasted bandwidth.

To facilitate the interoperability of microblocks, all fields in the metadata must be accessed via accessor functions. If the layout of the metadata is changed, these accessor functions need to be updated. In microcode assembler, the accessor functions are implemented as macros that are called by the driver or microblock. When microC is used (or on the Intel XScale® core), the packet metadata is defined as structures and accessed via structure references. Thus, a recompile of software with the new structure will ensure correctness. It is recommended that packet processing microblocks should not have any dependencies on the relative layout of the metadata or the exact size of metadata fields - these should be hidden by the accessor functions.

Details on packet metadata accessor functions are provided in Chapter 5, “Dispatch Loop.”

3.2.1 Packet Header Caching

When processing a packet using microblocks, it is likely that more than one of the blocks may access the same regions of the packet header. For example, a microblock that is performing DiffServ classification may need to both read and write some of the packet's IP header. The IPv4 forwarding block will also need to read and write portions of the IP header. We do not want both of these blocks to read and write the same regions of memory because this will incur extra memory operations (consuming bandwidth) as well as adding additional latency that must be hidden for efficient operation.

To get around this problem, the IXA Portability Framework provides a library that caches a portion of the packet header within the microengine. If the microblock authors use this library, then redundant accesses to the same memory regions will hit the cache.

Note: Partial Packet Ordering will be discussed in subequent revisions of this document.

3.2.2 Packet Source and Packet Sink Libraries

The Microblock Infrastructure provides a set of libraries that allow the dispatch loop to read and write packets from the various queuing devices. These are provided as a convenience to the developer and a developer can write their own mechanisms if these do not meet their requirements.

These library calls provide functions that:

- Source/sink packets from/to a different microengine using SRAM or scratch rings.
- Source/sink packets from/to the Intel XScale® core for further processing.
- Source/sink packets from/to a different microengine using the next neighbor registers.
- Source/sink packets using the SRAM rings.
- Put packets into an AISR ring for re-ordering

The source library calls get the next packet to work on. The sink calls dispose of the current packet. Examples of these are the dl_xxx_source[] and dl_xxx_sink[] macros provided in the Intel® IXA SDK software.

Microblocks 4

The data plane processing on the microengines is divided into logical networking functions called *microblocks*. Several microblocks can be combined into a *microblock group*. A microblock group has a *dispatch loop* that defines the dataflow for packets between different microACEs within the group. A microblock group can be instantiated on one or more microengines, but two microblock groups cannot share the same microengine.

Microblocks can send packets to an associated Intel XScale® core component. The dispatch loop handles packets that come from the Intel XScale® core component and steers them to the appropriate microblock.

Microblocks are coarse grain, stateful entities that perform major functions, such as Ipv4 forwarding, Ethernet layer 2 filtering, 5-Tuple lookup, MPLS label insertion, PPP header termination, and so on.

It is the intent that each microblock should be written independently of the other microblocks. By providing clean boundaries between these blocks, it makes it possible to modify, add, or remove more microblocks without affecting the behavior of the other blocks. If microblocks are written assuming another block is run before or after this block, then the modularity that they provide is lost.

4.1 Microblock Types

There are two types of microblocks:

- Packet Processing Microblocks perform high-level protocol-specific functions on a packet, for example, IPv4 forwarding, IPv6 forwarding, Bridging, and Network Address Translation (NAT). Note that as opposed to a low level macro/function (IP checksum), a microblock is coarse-grained, and has state and associated data structures typically shared with the Intel XScale® core (for example, IPv4 forwarding).
- Driver Microblocks are hardware or media specific code blocks which may be implemented in hardware in future revisions of the IXP processor. Examples include Receive/Transmit blocks that interact closely with the MSF interface on the IXP2xxx, and the Queue Manager microblock, which manages the queuing hardware (Q-Array). A developer will write software as driver microblocks when there is specific hardware that needs to be managed, (such as receive or transmit hardware) or the software processing model does not match the "get a packet, process it, and pass the packet on" model (for example, schedulers that do not operate on packets, but queues).

4.2 Structure of a Microblock

A microblock written in microcode consists of two macros:

- An initialization macro which is called only once by the dispatch loop during the startup sequence.
- A packet processing macro, which is called for every packet received.

For example, an IPv4 microblock would have two associated macros: `Ipv4_Init[]` and `Ipv4[]`.

A microblock written in microC consists of two functions:

- An initialization function
- A packet processing function

4.3 Microblock Name and ID

Each microblock in the system has a globally unique name and an 8-bit ID associated with it. The microblock IDs are allocated at compile time and are maintained in a system header file `dl_system.h`, which is used to set up the data flow bindings as described below.

Note: The globally unique ID does not imply that the ID does not change from application to application. The microblock ID only needs to be unique among all the microblocks running in the system. Its value may change from design to design even for the same microblock.

4.4 Outputs for a Microblock

One of the primary goals of structuring applications into microblocks is to allow developers to write the functionality without knowledge of which microblocks are upstream and downstream. Thus, the microblocks need to be written in such a way that the values of the block ID as well as setting the `dl_next_block` can be defined at compile time instead of when the microblock is written.

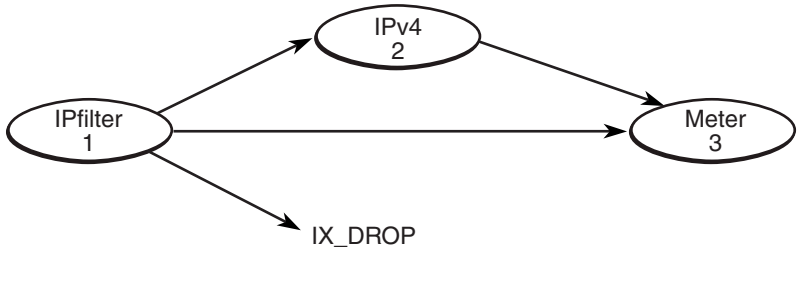
Each microblock can have one or more logical outputs that indicate which path the buffer should follow next. The logical output is passed along by setting the dispatch loop variable `dl_next_block` to a specific value. To make the microblock generic, symbolic names will be used for these outputs.

For example, a microblock called `IPFilter` may have three logical outputs `PASS_1`, `PASS_2`, and `DENY`. These would be defined in the `dl_system.h` file as `IPFILTER_PASS_1`, `IPFILTER_PASS_2`, and `IPFILTER_DENY` and set to the appropriate block IDs. It is assumed that the documentation for each microblock will include the number of logical outputs it has and also the semantic meaning associated with those outputs.

The following values are reserved:

- `IX_EXCEPTION`: Indicates that the packet should be sent to the core.
- `IX_EXCEPTION_PRIORITY`: Indicates that the packet should be sent to the core on the high priority queue.
- `IX_DROP`: Indicates that the packet should be dropped.
- `IX_NULL`: Indicates that a NULL packet is being handled.

Figure 4-1. Example Packet Flow



A sample `system.h` file is shown below:

```
#define IPFILTER_BLOCKID        1
#define IPV4_BLOCKID            2
#define METER_BLOCKID           3
#define IPV4_NEXT1              METER_BLOCKID
#define IPFILTER_PASS_1         IPV4_BLOCKID
#define IPFILTER_PASS_2         METER_BLOCKID
#define IPFILTER_DENY           IX_DROP
```

In this case, the `IPFilter` block would send packets on the `PASS_1` logical output to the IPV4 forwarding block, `PASS_2` output to the meter block, and the packets on the `DENY` output would be dropped.

4.5 Configuring a Microblock

There are three ways to initialize and configure a microblock.

- *Control block*: Each microblock may have an area of SRAM that it uses for communication with its associated Intel XScale® core component. This stores parameters that may change at run-time e.g., MAC filters.
- *Imported variables*: These are used for values that can be determined during load-time of the microcode and do not change subsequently. For example, the location of the control block allocated in SRAM from the Resource Manager.
- *Tables and other data structures in SRAM, DRAM, or Scratch*: These are shared between the microengine code and the code running on the Intel XScale® core. An example of such a shared data structure is the IPv4 forwarding table.

4.6 Critical Sections and Folding in a Microblock

A *critical section* is a section of code in which only one microengine thread has exclusive modification privileges for a global resource (such a memory location) at any one time. On the IXP2400, in the ordered threading model, a critical section of code is implemented in the following situations:

- Between microengines, by making only one microengine at a time execute the critical section. This is done by making the microengines run in order and using microengine to microengine signaling.
- Within microengines, the critical section is typically implemented by making sure the thread does not swap out while performing the modification and write operations on the shared data.

Typically the critical section contains a read-modify write, which can be optimized by using the folding technique. In this case the threads must execute in strict order and use local inter-thread signaling to ensure the order.

Note: Under this implementation of a critical section, it is quite likely that other microengines/threads will stall waiting for a signal from the current microengine/thread. This implementation has been chosen over memory locks used in the IXP1200, due to the high latency of memory accesses.

An example of the pseudo-code for a microblock that implements a critical section via thread ordering and using folding is shown below:

```
// Wait for previous microengine or previous thread
If (Context 0 of the microengine)
Wait for Signal from previous microengine
Else
    Wait for Signal from previous thread
// Enter first phase of critical section
Look up the CAM to see if data is in local memory
If (CAM Miss) {
    Issue Read to read data into local memory
      Signal next thread
Increment reference count for this entry in LM
      // Exit first phase of the critical section
      Swap out waiting for the read to finish and a
signal from the previous thread (thread 0 waits on thread 7)
}
else {
// exit first phase of critical section
Increment reference count for this entry in LM
Send Signal to next thread
Swap out waiting for a signal
From the previous thread (thread 0 waits on thread 7)
}
```

```
// Enter second phase of critical section
If (CAM miss)
    Move data read in into local memory
Modify the data read in.
Decrement reference count for this entry
If (reference count == 0)
Write back the data
If (last thread for the microengine)
    Send signal to next microengine
Else
Signal the next thread
// Exit the critical section
Swap out and wait for the write to finish if one was issued.
```

In the example above, the processing in the microblock is broken into two phases. In the first phase, each thread waits for its turn (thread 0 waits for the previous microengine) and then checks the CAM to see if its data is in local memory. If not, it issues a read to get the data into local memory and sets up the CAM entry accordingly. It then signals the next thread (thread 7 signals thread 0). In the next phase, each thread waits for its turn and modifies the data read in. The last thread using an entry writes it out to memory. Thread 7 signals the next microengine. In this way, multiple read-modify-writes on the same entry are "folded" into a single read, multiple modifies and a single write.

One issue is that the microblock needs to know which microengines are participating in the microengine to microengine signaling. The microblock will need to use imported variables to make this happen. For example, a microblock can use an imported variable `NEXT_ME` to determine which microengine to signal next. This variable needs to be patched in by the building block core component on the Intel XScale® core (using the microengine mask that is passed in to the core component indicating which microengines it runs on).

A significant consequence of having microblocks where the threads run in strict order (and the order is imposed by thread signaling) is that all threads must go through all microblocks on the microengine. Even if a packet is dropped at any stage of the pipeline, the thread must continue through the rest of the pipeline on the microengine so it may signal the next thread in an ordered critical section. The same applies if the packet goes to the core or if it skips stages of a pipeline. The implication of this is that the data flow in the dispatch loop is no longer a graph but a sequential pipeline.

The data plane library components `ixp_sig` and `ixp_critsect` interfaces may be used to implement the critical section in microcode. For more information, see the *Intel® Internet Exchange Architecture Portability Framework Reference Manual*.

In the unordered thread model, critical sections may be implemented via locks in memory using the SRAM or Scratch atomic operations. The folding technique is not applicable. Atomic operations in scratch are recommended over atomic operations in SRAM which have a performance penalty on IXP2400 and IXP2800 A1 silicon.

4.7 Exception Packets

If a microblock throws an exception and sends the packet to the core, it calls the dispatch loop macro `dl_set_exception[]` (or microC function `dl_set_exception()`). This takes as arguments:

- The microblock ID for this block. This allows the Resource Manager on the core to send this packet to the appropriate Intel XScale® core component that has registered an exception packet handler for this block.
- The exception code. This is an 8-bit value that the microblock uses to indicate to the Intel XScale® core component why this packet was marked as an exception packet. The values for the exception code and how they are interpreted are decided between the microblock and the associated Intel XScale® core component.
- The value for `dl_next_block`. Currently two priorities are supported for exception packets and `dl_next_block` may be either `IX_EXCEPTION` or `IX_EXCEPTION_PRIORITY`.

All of the above are stored and passed along in the cached dispatch loop state.

In the ordered threading mode, the packet still goes through all the other blocks in the pipeline (because some of these blocks may be strictly ordered critical sections). The other blocks check the value of `dl_next_block` and simply pass the packet along. Finally the packet gets to the system sink block (`dl_sink[]`). This block queues the packet in a scratch (or SRAM) ring going to the core. The framework supports two scratch (or SRAM) rings going to the core–one for high-priority packets and the other for low priority packets. These rings will be shared by all microengines.

4.8 Receiving Packets from the Intel XScale® Core

On both the Intel® IXP2400 Network Processor and the Intel® IXP2800 Network Processor, scratch rings are used to pass packet descriptors from one microengine to the next. To receive packets from the Intel XScale® core, a separate scratch ring is used. The dl_source[] macro needs to poll this scratch ring in addition to the ring from the previous microengine. This can be done in two ways - one is to dedicate a single microengine thread to handle only packets from the Intel XScale® core, the other is to poll the ring from the Intel XScale® core periodically (e.g. once in 64 packets). The method chosen depends on the application and may be modified by the developer in the dl_source[] implementation.

Note that the Intel XScale® core can atomically write to a scratch ring shared by the microengines. However, it can only write a single word atomically into the scratch ring. Since most messages on the ring are multiword, this feature is not used to write directly to the microengine to microengine scratch ring.

4.9 Dropping Packets

If a microblock decides to drop a packet, it sets `dl_next_block` to `IX_DROP`. The packet still has to go through the rest of the pipeline. The other microblocks in the pipeline will check for the value of `dl_next_block` to see if they need to handle the packet.

4.10 Handling Null Packets

A microblock may receive NULL packets in the ordered thread model. For example, if a thread polls a scratch ring and no packet is available it receives NULL from the ring. This thread still needs to execute the entire pipeline with this NULL packet. This implies that a microblock must support bypass of NULL m-packets (the inter-thread signaling must be done if it contains a critical section).

If a NULL packet is received, then `dl_next_block` will be set to `IX_NULL`. Microblocks can check for the value of `dl_next_block` to determine if they need to handle the packet or not.

4.11 Ordered vs. Unordered Model

For packet processing microblocks without critical sections (for example, IPv4/v6 forwarding, MPLS marking, and switching) there is little difference in the code for the ordered and unordered models. One change is that in the unordered model, it is not required to check if the packet has the dl_next_block set to the block id for the current microblock. This check should be compiled out of the code with a build switch #ifdef POTS.

For blocks with critical sections, the algorithm for locking and accessing the shared data structure is different. Future revisions of the document will explore how to hide this via a library API.

Driver blocks specially the receive and transmit code will typically be written to the ordered thread model. Note that in typical applications, even if the packet processing code runs in the unordered thread model, the driver code for Receive and Transmit may run in the ordered thread model.

Dispatch Loop 5

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

Note: A dispatch loop is specific to the application being targeted. The intent is for the microblocks to be as reusable as possible. The dispatch loop and the internal implementation of its associated helper macros or functions may be optimized for a specific application.

5.1 Dispatch Loop for the Ordered Thread Model

The dispatch loop implements the execution of the flow among the microblocks. Conceptually this can be a graph, but because of the restrictions discussed in Section 4.6, "Critical Sections and Folding in a Microblock," this graph must be mapped into a linear execution of the microblocks. This linear execution mapping is derived by performing a topological sort of the microblocks.

To implement the behavior of the graphs, an integer value is assigned to each of the microblocks. At the exit to the microblock, the integer value will set the `dl_next_block` value associated with the current packet. When a microblock receives control of the packet, the receiving microblock must compare the packet's `dl_next_block` number to the receiving microblock's number. If the `dl_next_block` number on the packet is not equal to the current microblock's number, the microblock must perform the bypass operation; otherwise it processes the packet as normal.

Figure 5-1. A Flowchart of the Logical Call Order for the Example Microblocks

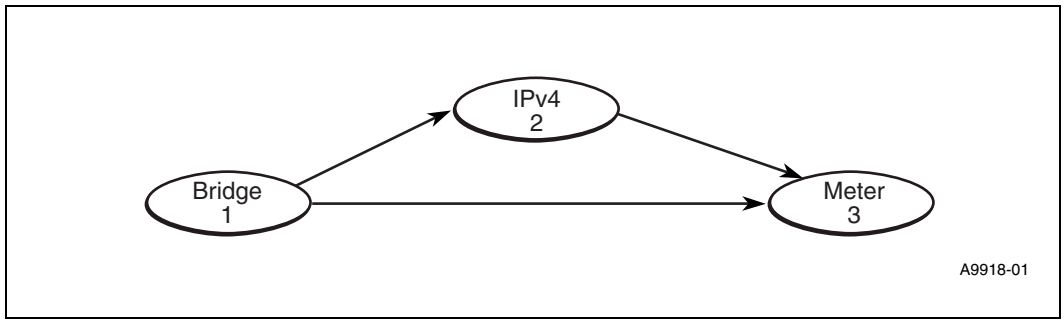


Figure 5-1 shows a simple case with three microblocks. Block 1 is a `Bridge` microblock performing layer 2 bridging. It has two logical outputs; one for bridged traffic and one for non-bridged traffic. Packets that are bridged go directly to block 3, a `Meter` microblock, while non-bridged data is sent to block 2, an `IPv4` forwarding microblock. Traffic from block 2, the `IPv4` forwarding microblock, is then sent to block 3, the metering microblock.

Figure 5-2. Flattened Microblock Call Graph.

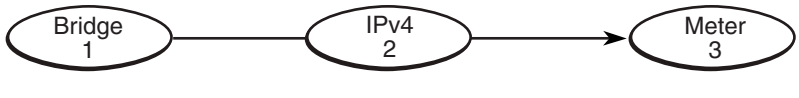


Figure 5-2 transforms the graph in Figure 5-1 into a linear sequence of microblocks. Each microblock will be called in turn. For example, at the `Bridge` microblock, the `dl_next_block` value of the dispatch loop will be set to `2` or `3` based on whether the packet is non-bridged or bridged, respectively. At the `IPv4` microblock, the `dl_next_block` value will be examined. If it is `2`, then forwarding operations are performed. If the value is `3`, bypass operations are performed.

The details of setting the `dl_next_block` are covered in later sections.

5.2 Dispatch Loop Structure

As described in the previous section, the dispatch loop is a sequential pipeline of microblocks. Below is an example of a dispatch loop for the ingress functional pipeline for the OC-48 POS IPv4 forwarding application described in the *Intel® Internet Exchange Architecture Software Building Blocks Developer's Manual*.

```
// include header files
#include dispatch_loop.h // generic dispatch loop structures
#include dl_pos.h        // pos specific dispatch loop variables
#include IPFwd.h         // IP Longest Prefix Match forwarding block
#include Meter.h         // Metering block
#include WRED.h          // WRED block

extern DL_PosRx dl;  // instantiate dispatch loop control structure

void main()
{
    DL_Source_Init(); // initialize DL_Source block
    IPFwd_Init();     // initialize the Ipv4 Forwarding block
    Meter_Init();     // initialize the meter block
    WRED_Init();      // initialize the WRED block
    DL_Sink_Init();   // initialize DL_MESink block

    while(TRUE) {     // Run the dispatch loop

    // Get a packet from the scratch ring
        DL_Source();
        // call the IP forwarding block
        IPFwd();
        // call the metering block
        Meter();
        // call the WRED block
```

```
        WRED();
        // pass packet to another microengine
        DL_Sink();        //
    }   // end dispatch loop while
}   // end main
```

The dispatch loop initializes each of the microblocks within its scope by calling the `<BlockName>_Init()` function, as shown above.

The dispatch loop then executes a loop in which it sequentially calls all the microblocks in the pipeline. The first microblock of the dispatch loop is a system source block that dequeues packets from a scratch ring. These packets may come either from the Intel XScale® core or a previous microengine. Each microblock will set the `dl_next_block` variable indicating which microblock needs to handle the packet next.

5.3 Dispatch Loops for the Unordered Thread Model (POTs)

Most of the concepts introduced in the previous section apply to writing dispatch loops for the unordered thread model. The buffer metadata is still maintained in global structures, and a dl_sink library function still handles exception and dropped packets. The differences between the dispatch loop styles are the result of actual programming models. In the unordered thread model, the dispatch loop does not need to execute every microblock in order. It may instead skip completely over the microblocks that do not need to process a particular packet, and it may also loop back and re-execute some microblocks. This change affects two aspects of dispatch loop writing: dispatch loop control flow and handling packets from the Intel XScale® core.

5.3.1 Dispatch Loop Control Flow

In the dispatch loop for the unordered thread model, the dl_next_block value is checked after each microblock is called. Based on the value of dl_next_block, the code determines which packet the block goes to next.

There are two reserved values for exception packets and packets to be dropped. These reserved values are `IX_EXCEPTION` and `IX_DROP`, just as in the ordered thread execution dispatch loops. When the `dl_sink[]` block receives a packet with a `next_block` value matching one of these return values, it performs the correct action. If the value is neither of these, it passes the packet to the next microengine. It is very important that even dropped packets are sent to the sink block, since it manages packet reordering and skipped sequence numbers can cause the system to lock up.

When a dispatch loop writer is writing code to test the `dl_next_block` value to decide which microblock to execute next, they have a few options. Developers can use *if* statements to construct the entire control flow. For example:

```
// run the bridge microblock and test the return variable
bridge()
.if (dl_next_block == BRIDGE_OUTPUT0)
    ipv4()
    .if (dl_next_block == IX_EXCEPTION)
        // if this is an exception packet, send to the system
        // driver
        dl_sink()
```

```
    .else
       meter()
       dl_sink()
    .endif
.else
    meter()
    dl_sink()
.endif
```

Although this example is in microengine assembly, the microengine C variant of this code would look almost the same. The only difference would be that the microengine C microblock would actually return the next- block code.

The advantage of using an *if* statement to implement the microblock flow is that the flow of packets through the microblocks is readily apparent from just the dispatch loop. There are, however, some disadvantages. For example, in the assembly code above, the code for the meter microblock is duplicated in the control store since it is a microengine assembly macro. This reduces the amount of control store available and may actually make the code too large to fit on a microengine. Non-inlined function calls in microengine C eliminate this drawback but reduce performance and reduce the ability of the microengine C compiler to perform optimizations on otherwise inlined code. Although the example may work well for implementing simple flows, more complicated flows with loops may require a branch or a *goto* statement. Therefore, this dispatch loop style is recommended only for simple graphs without loops.

Another way to implement the microblock flow in a dispatch loop is to use *if* statements coupled with branches in microengine assembly or *goto* statements in microengine C. Using this style, the above code could be rewritten in microengine C as follows:

```
// run the bridge microblock and test the return variable

bridge:
Bridge();
if (dl_next_block == BRIDGE_OUTPUT0)
{
    goto ipv4;
}
else {
    goto meter;
}

ipv4:
Ipv4();

if((dl_next_block == IX_EXCEPTION)
{
    goto dl_sink;
}

meter:
Meter();

dl_sink:
dl_sink();
```

Another option available for implementing the microblock flow is the `br=byte` instruction. This option results in only one branch per microblock. For example:

```
// run the bridge microblock and test the return variable
bridge#:
bridge();
br=byte[dl_next_block, 0, BRIDGE_OUTPUT0, ipv4#]
br[meter#]

(etc)
```

5.3.2 Packets from the Intel XScale® Core

As in the dispatch loops for ordered thread model, the framework macro dl_source may produce packets that are from the Intel XScale® core and could be destined for any of the microblocks in the microblock group. In order for this to work, the Intel XScale® core blocks need a way to reference the correct microblocks. The Intel XScale® core blocks use bindings.h file to do this. The dl_source function then sets the dl_next_block to one of these unique microblock IDs.

In the dispatch loop, the dl_next_block value set by dl_source could be the microblock ID for any of the microblocks in the microblock group. To test this return value, the dispatch loop code could use a series of if statements with associated branches/*goto*s, or it could use a case statement/jump instruction. Here is a simple example illustrating this style:

```
while(1)
{
    if (dl_buffer_handle)
    {
        dl_source();
        switch (dl_next_block)
        {
        case BRIDGE_BLOCKID:
            goto bridge;
            break;
        case IPV4_BLOCKID:
            goto ipv4;
            break;
        case METER_BLOCKID:
            goto meter;
            break;
        default:
               // This is a catastrophic error
               goto drop;
               break;
        }
    }
    else
    {
        continue;
    }
    ...
```

This style could be inefficient in some cases. For example, if the number of packets coming from the Intel XScale® core is small, and if the remainder of the packets all go to the same microblock, executing the case statement each time a packet enters the microblock group could be very inefficient. To address this problem, the framework has additional dl_source functions/macros that allow the developer to specify the scratch ring from which to dequeue packets. This could then be used to balance the rate at which packets are dequeued from the core with the rate at which packets are dequeued from other sources, and reduces the number of times the case statement/jump instruction must be executed. The code below shows this:

```
int count = 0;
while(1)
{
       if (count++ & 0xf)
       {
           dl_source(RING_FROM_PREV_MICROENGINE);
           if (dl_buffer_handle)
       {
           // All packets from the previous microengine go to the
           // bridging microblock
           goto bridge;
       }
       else
       {
           continue;
       }
   }
   else
   {
       dl_source(RING_FROM_CORE);
       if (dl_buffer_handle)
       {
           // Packets from the core could go to any microblock
           // in the microblock group
           switch (dl_next_block)
           {
           case BRIDGE_BLOCKID:
               goto bridge;
               break;
       case IPV4_BLOCKID:
               goto ipv4;
               break;
       case METER_BLOCKID:
               goto meter;
               break;
       default:
                   // This is a catastrophic error
                   goto drop;
                   break;
               }
       }
       else
       {
           continue;
```

```
        }
}
    ...
```

5.4 Dispatch Loop Variables

The dispatch loop maintains some global state, which may be cached in registers or local memory. In the case of microC, this state will be maintained in a global C structure. The compiler decides whether this structure can be cached in registers or if some the data structure elements will be cached in local memory.

5.4.1 Microengine Assembler Dispatch Variables

Table 5-1 lists variables that may be cached by a dispatch loop (the actual variables cached depend on the nature of the application and can customized by the developer).

Table 5-1. Microengine Assembler Dispatch Loop Variables[1]

Field Name	Size	Use
`exception_id`	8 bits	This is used by microblocks when sending packets to the Intel XScale® core. The microblock must set the `exception_id` to the microblock ID when indicating an exception.
`exception_code`	8 bits	The microblock sets an 8-bit exception code when a buffer is sent to the Intel XScale® core component. This exception code is treated as opaque data by the dispatch loop and Resource Manager.
`dl_next_block`	8 bits	Identifies the next logical block to process after the current block. The current block sets this value after it is done processing.
`dl_buf_handle`	32 bits	The buffer handle containing the start of the packet.
`dl_eop_buf_handle`	32 bits	The buffer handle containing the end of the packet.
`buffer_size`	16 bits	This is the length of the buffer containing the start of the current packet. This is the total length of the buffer including all of the headers. If the buffer is not complete, this is the amount of data currently in the buffer.
`packet_size`	16 bits	This is the total length of the packet across multiple buffers.
`buffer_offset`	16 bits	This is the offset from the start of the buffer to the buffer data. Transform blocks that pack or unpack the buffer must change this offset.
`input_port`	16 bits	This contains the logical port number on which the packet was received. Port numbers should be in the range of 0 to 255.

Table 5-1. Microengine Assembler Dispatch Loop Variables[1] (Continued)

Field Name	Size	Use
rx_stat	4 bits	These are receive status flags for a buffer. Supported flags are IX_RXSTAT_UCAST, IX_RXSTAT_BCAST, IX_RXSTAT_MCAST, and IX_RXSTAT_PROMISC.
output_port_egress	24 bits	This is the number of the port interface on which the packet is to be transmitted in a given blade.
output_port_fabric	8 bits	When multiple blades are connected to the fabric this is the blade ID.
output_port_type	4 bits	The type of interface on which the packet is to be transmitted—for example, POS, ATM, Ethernet, and so on.
cache_flags	4 bits	This is used for caching packet headers. Each bit represents 32 bytes of the packet header—the cache line. Two bits are used to detect if a cache line is in local memory. Two bits are used to check if a cache line is dirty and needs to be written out.
next_hop_id	32 bits	Next hop IP ID
flow_id (QoS only)	32 bits	The flow identifier that is used for metering and other QoS functions.
queue_id (QoS only)	16 bits	The output queue identifier. This is set when classifying packets for quality of service processing.

1. These variables may be cached.

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

5.4.2 Microengine C Loop Data Structure

For microblocks written in Microengine C, the variables described in Section 5.4.1 and the packet headers are stored in global data structures. This section describes the data structure used to store this data. Microblocks written in Microengine C access data in this structure by directly referencing its member fields. Consequently there are no get or set functions for this data structure.

Microengine C Data Structure Members

buffer_next	The next buffer in the chain.
buffer_size	The amount of data currently in the buffer.
offset	The offset in DRAM where the data begins.
packet_size	The amount of data in the buffer chain.
free_list_id	The free list to which this buffer belongs.
rx_stat	The receive status.
header_type	The header type—IPv4, IPv6, and so on.

Microengine C Data Structure Members (Continued)

`input_port`	The input port on which this packet was received.
`output_port`	The output port on which this packet is to be transmitted.
`next_hop_id`	The next hop ID.
`fabric_port`	The blade port.
`reserved`	Reserved for future use.
`nhid_type`	The next hop ID type.
`flow_id`	The flow ID.
`class_id`	The class ID.
`reserved_2`	Reserved for future use.
`packet_next`	The next packet in the chain—used only in *Hierarchical Queuing*.

5.5 Dispatch Loop Macros

Of the variables described in Section 5.4, "Dispatch Loop Variables," the buffer handles and the next block will be stored in global variables (dl_buf_handle, dl_eop_buf_handle, and dl_next_block). The remaining variables are packed into registers.

The IXA SDK provides macros for buffer allocation, buffer freeing, the return or modification of various IP header fields, and so on. There are two categories of helper macros that support Dispatch Loops. Full details on these macros are contained in Section 2.2, "Dispatch Loop Interface" of the *Intel® Internet Exchange Architecture Portability Framework Reference Manual*.

Table 5-2 summarizes the Dispatch Loop macros supporting meta data and Table 5-3 summarizes the Dispatch Loop macros supporting extended meta data.

Table 5-2. Dispatch Loop API Functions for Meta Data

Name	Description
`dl_buf_init[]`	Initializes the Buffer API.
`dl_buf_alloc[]`	Allocates a buffer.
`dl_buf_free[]`	Frees a buffer.
`dl_buf_get_desc[]`	Returns the SRAM pointer to the meta data given a buffer handle.
`dl_buf_get_data[]`	Returns the DRAM pointer to the buffer data given a buffer handle.
`dl_buf_get_data_from_meta[]`	Returns the DRAM pointer using the SRAM base as input.
`dl_meta_init_cache[]`	Populates a meta data cache.
`dl_meta_flush_cache[]`	Flushes meta data to SRAM.
`dl_meta_load_cache[]`	Loads meta data from SRAM into registers.
`dl_meta_get_buffer_next[]`	Returns the handle of next buffer in the buffer chain—for large packets.

Table 5-2. Dispatch Loop API Functions for Meta Data (Continued)

Name	Description
dl_meta_set_buffer_next[]	Sets the handle of next buffer in the buffer chain—for large packets.
dl_meta_get_hw_next[]	Gets the hardware next field in the handle
dl_meta_set_hw_next[]	Sets the hardware next field in the handle
dl_meta_get_offset[]	Returns the offset at which data begins within a buffer.
dl_meta_set_offset[]	Sets the offset at which data begins within a buffer.
dl_meta_get_free_list[]	Returns the free list from which the current buffer—that is, the buffer pointed to by dl_buf_handle—was allocated. There may be multiple free lists—that is, buffer pools—but only one is in use at any point in time.
dl_meta_set_free_list[]	Sets the free list to which the current buffer belongs. There may be multiple free lists—that is, buffer pools—but only one is currently used.
dl_meta_get_rx_stat[]	Returns the receive status.
dl_meta_set_rx_stat[]	Sets the receive status.
dl_meta_get_buffer_size[]	Returns the buffer size of the current buffer in the packet.
dl_meta_set_buffer_size[]	Sets the buffer size of the current buffer in the packet.
dl_meta_get_input_port[]	Returns the input port over which the packet came in.
dl_meta_set_input_port[]	Sets the input port.
dl_meta_get_packet_size[]	Returns the total packet size.
dl_meta_set_packet_size[]	Sets the total packet size.
dl_meta_get_nexthop_id[]	Returns the ID for the next hop.
dl_meta_set_nexthop_id[]	Sets the ID for the next hop.
dl_meta_get_output_port[]	Returns the output port. This is the port on the egress IXP2400 out of which the packet is transmitted.
dl_meta_set_output_port[]	Sets the output port. This is the port on the egress IXP2400 out of which the packet is transmitted.
dl_meta_get_fabric_port[]	Returns the output blade (when multiple blades are connected to the fabric) from which the packet is transmitted.
dl_meta_set_fabric_port[]	Sets the output blade (when multiple blades are connected to the fabric) from which the packet is transmitted out.
dl_meta_get_flow_id[]	Returns the flow ID.
dl_meta_set_flow_id[]	Sets the flow ID.
dl_meta_get_class_id[]	Returns the class ID.
dl_meta_set_class_id[]	Sets the class ID.
dl_buf_set_SOP[]	Sets the SOP bit in the buffer handle. This indicates that the buffer contains the start-of-packet.
dl_buf_set_EOP[]	Sets the EOP bit in the buffer handle. This indicates that the buffer contains the end-of-packet.
dl_buf_get_cell_count[]	Gets cell count from the buffer handle.

Table 5-2. Dispatch Loop API Functions for Meta Data (Continued)

Name	Description
dl_buf_set_cell_count[]	Sets the cell count in the buffer handle.
dl_set_exception[]	Sets the exception code.
dl_meta_get_nexthop_id_type[]	Returns the next hop ID type—IPv4, IPv6, and so on.

.

Table 5-3. Dispatch Loop API Functions for Extended Meta Data

Name	Description
dl_meta_parent_get_ref_cnt[]	Obtains the reference count value.
dl_meta_child_get_child_offset[]	Obtains the child buffer data offset in bytes.
dl_meta_child_set_child_offset[]	Sets the child buffer data offset in bytes.
dl_meta_child_get_child_buffer_size[]	Obtains the child buffer data size in bytes.
dl_meta_child_set_child_buffer_size[]	Sets the child buffer data size in bytes.
dl_meta_child_get_child_freelist_id[]	Obtains the freelist ID of the child buffer.
dl_meta_child_set_child_freelist_id[]	Sets the freelist ID of the child buffer.
dl_meta_child_get_parent_offset[]	Obtains the data offset of the parent buffer.
dl_meta_child_set_parent_offset[]	Sets the data offset of the parent buffer.
dl_meta_child_get_parent_buffer_size[]	Obtains the data size of the parent buffer.
dl_meta_child_set_parent_buffer_size[]	Sets the data size of the parent buffer.
dl_meta_child_get_header_type[]	Obtains the header type of the packet.
dl_meta_child_set_header_type[]	Sets the header type of the packet.
dl_meta_child_get_parent_free_list[]	Obtains the free list ID of the parent buffer.
dl_meta_child_set_parent_free_list[]	Sets the free list ID of the parent buffer.
dl_meta_child_get_rx_stat[]	Obtains the receive status of the packet.
dl_meta_child_set_rx_stat[]	Sets the receive status of the packet.
dl_meta_child_get_packet_size[]	Obtains the size of the packet across all buffers.
dl_meta_child_set_packet_size[]	Sets the size of the packet across all buffers.
dl_meta_child_get_output_port[]	Obtains the output port number for this packet.
dl_meta_child_set_output_port[]	Sets the output port number for this packet.
dl_meta_child_get_input_port[]	Obtains the input port number for this packet.
dl_meta_child_set_input_port[]	Sets the input port number for this packet.
dl_meta_child_get_nexthop_id[]	Obtains the next hop ID for this packet.
dl_meta_child_set_nexthop_id[]	Sets the next hop ID for this packet.
dl_meta_child_get_fabric_port[]	Obtains the fabric port number for this packet.
dl_meta_child_set_fabric_port[]	Sets the fabric port number for this packet.
dl_meta_child_get_nexthop_id_type[]	Obtains the nexthop ID type for this packet.
dl_meta_child_set_nexthop_id_type[]	Sets the nexthop ID type for this packet.
dl_meta_child_get_flow_id[]	Obtains the flow ID for this packet.

Table 5-3. Dispatch Loop API Functions for Extended Meta Data (Continued)

Name	Description
dl_meta_child_set_flow_id[]	Sets the flow ID for this packet.
dl_meta_child_get_color[]	Obtains the color of this packet.
dl_meta_child_set_color[]	Sets the color of this packet.
dl_meta_child_get_class_id[]	Obtains the class ID of this packet.
dl_meta_child_set_class_id[]	Sets the class ID of this packet.
dl_meta_child_get_parent_buffer_id[]	Obtains the parent buffer ID to which this child buffer is linked.
dl_meta_child_set_parent_buffer_id[]	Sets the parent buffer ID to which this child buffer is linked.
dl_meta_child_get_buffer_next[]	Obtains the next buffer handle for this child buffer.
dl_meta_child_set_buffer_next[]	Sets the next buffer handle for this child buffer.
dl_meta_child_get_packet_next[]	Obtains the next packet handle for this child buffer.
dl_meta_child_set_packet_next[]	Sets the next packet handle for this child buffer.

Optimized Data Plane Libraries Support 6

The IXA SDK Tools CD provides additional library support for the optimized data plane libraries. These libraries consist of generic microengine software building blocks used to construct an application's microengine modules, called microblocks. For more information on microblocks, see Chapter 4, "Microblocks".

The optimized data plane macro libraries are reusable software functions optimized for high performance and minimal executable code size.

For details concerning the use of these libraries and the specific APIs they support, see the *Intel® Internet Exchange Architecture Optimized Data Plane Libraries Reference Manual* on the IXA SDK Tools CD.

Resource Manager 7

7.1 Overview

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

The Resource Manager functionality includes:

- Hardware resource allocation, initialization, and configuration
  - — Memory–SRAM, DRAM, Scratch, and local memory
  - — Hardware queues and rings
- Microengine management
  - — Loading
  - — Patching symbols
  - — Enable
  - — Disable
- Buffer management
- Communication with microblocks

7.2 Changes for IXA SDK 3.x

This section describes the major changes between the Resource Manager functionality available in the previous Intel® IXA SDK 2.0 release and the enhanced functionality in the Intel® IXA SDK 3.x releases. Each sub-section briefly describes the changes. The subsequent internal design section presents the design of Resource Manager incorporating all these changes.

7.2.1 Stand Alone API

In Intel® IXA SDK 2.0, the ACE framework was used to deliver packets to microACEs from microblocks. Also, the Resource Manager supported APIs for creating microACEs, binding static microACE targets, and so on.

For Intel® IXA SDK 3.x, the Resource Manager API supports a single stand-alone API, which may be used by the IXA building blocks or may be directly used by applications outside of that framework.

All APIs and functionality specific to the core component infrastructure have been moved into the Core Component Infrastructure library (see Chapter 10, “Core Component Infrastructure”). Also some APIs that are generic have been moved into the Resource Manager.

7.2.2 Compile Time Allocation of Microengines

The Intel® IXA SDK 2.0 Resource Manager allocated microengines based on the media interfaces (ports) in use. When version 2.0 microcode was compiled one UOF file was generated per microblock group. If a microblock group was instantiated on multiple microengines, then the UOF file for that microblock group was downloaded to each of these microengines.

While this approach improved usability for application developers, it was not possible for the Resource Manager to support every type of media interface. Also loading the same UOF file on multiple microengines produced problems in MicroC, where microengines could export and share variables—the SRAM memory spill area, for example—that are relocated by the linker.

The Intel® IXA SDK 3.x Resource Manager places responsibility for microengine allocation on the application developer at compile time. A single UOF file containing one UOF file per microengine is created using the workbench or command line tools.

7.2.3 Patching Symbols at Debug Time

In Intel® IXA SDK 2.0 microcode was loaded either by an application running on the target using the Resource Manager or by the workbench during debugging. The Intel® IXA SDK 2.0 Resource Manager did not support patching symbols in workbench mode.

The Intel® IXA SDK 3.x Resource Manager supports patching symbols in either workbench or execution mode. In workbench mode, all symbols patched by the application are stored locally until the workbench GUI initiates the microcode download and the UOF file is received.

7.2.4 VxWorks Support

The IXA Portability Framework supports VERITAS Works (VxWorks) and Linux. Even though Operating System Support Layer (OSSL) is used wherever possible, the Resource Manager is a combination of a kernel driver and a user library, which may require some changes to make it portable across these operating systems.

7.2.5 Hardware Resource Management

The Resource Manager manages the hardware resources supported in the IXP2400 and IXP2800 processors. These resources include:

- Scratch rings
- SRAM rings and link-lists
- SRAM, DRAM, scratch, and local memory
- RBUFs and TBUFs

7.2.6 Dispatch Loop Support

The Intel® IXA SDK 2.0 Resource Manager patched symbols that were required to configure the dispatch loop. Since the dispatch loop is very application-specific, these symbols vary depending on the type of application.

The Intel® IXA SDK 3.x Resource Manager defers patching the dispatch loop symbols that are application-specific (not generically applicable to every dispatch loop) to the system application.

7.2.7 MicroC Support

The Intel® IXA SDK 3.x Resource Manager supports microblocks written in MicroC.

7.2.8 Buffer Management

The Intel® IXA SDK 2.0 Resource Manager did not offer buffer-management support as this was handled in the ACE Framework and in microcode. The only Resource Manager processing of the buffer was to translate the packet metadata between the ACE Framework and microcode.

The Intel® IXA SDK 3.0 Resource Manager provides a buffer management API. This interface is used by the IXA Software Framework and by applications using the framework. Both hardware and software buffer pools are supported. The hardware pools are accessible by the core applications and the microcode at the same time, and therefore, they are suitable for core-microblock communication. The buffer management API supports buffer pools of different sizes as well as chaining of packet buffers.

7.2.9 Communication with Microblocks Using Hardware Features

The Intel® IXA SDK 2.0 Resource Manager provided packet transfer between microblocks and core applications using software only by means of polling in the dispatch loop for microblocks and a polling thread in Resource Manager for Intel XScale® core applications. The packet buffer handles were stored in software DRAM rings created by the Resource Manager.

The Intel® IXA SDK 3.x Resource Manager supports both control messages and packet data transfer. Hardware rings and interrupts are used as appropriate (see Section 4.7, “Exception Packets” and Section 4.8, “Receiving Packets from the Intel XScale® Core”).

There are two ways of doing this:

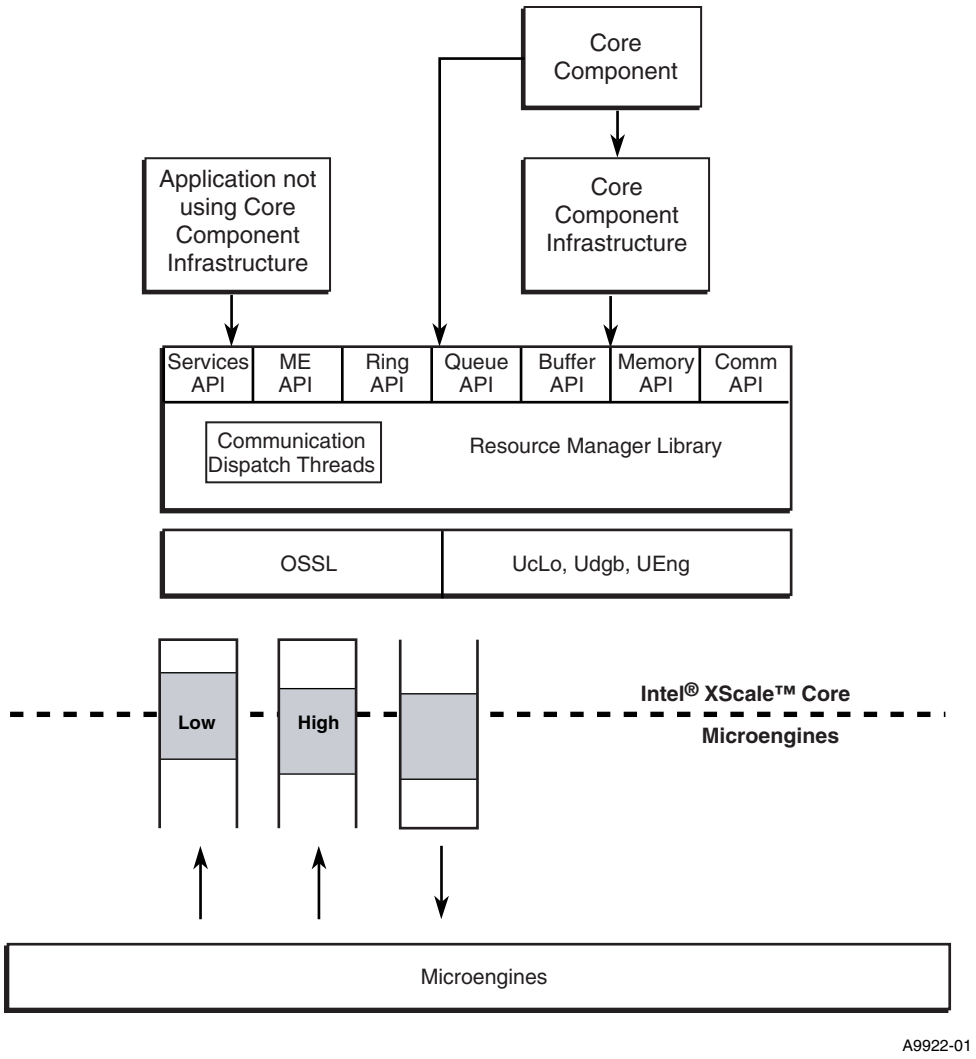
- To limit the number of interrupts going to the core, the dispatch loop sink block can send an interrupt to the Intel XScale® core only when the ring is full or above a particular threshold. The Resource Manager also polls the ring (the polling interval can now be increased). This ensures that packets are being dequeued even if the ring never gets full or reaches the required threshold.
- A specific interrupt (`IRQ_THREAD_A`) is reserved for communicating with the Resource Manager. When the Intel XScale® core receives this interrupt, it signals a task (outside of interrupt context) to dequeue the packets. The `IRQ_THREAD_A` interrupt from the microengine is masked at this point. The task dequeues packets from the ring until it is empty. At that point, it unmasks the `IRQ_THREAD_A` interrupt. The problem with this approach is that the interrupt cannot be used for anything other than communication with the Resource Manager.

The choice of which mechanism to use is dependent on the target implementation requirements.

7.3 Internal Design

Figure 7-1 shows the overall design of the IXA SDK Resource Manager. The Resource Manager implementation is layered on top of the OSSL and the workbench microengine loader (`UcLo`, `Ueng`) libraries. The Resource Manager is implemented as a library, kernel thread, and driver module under Linux. The driver component is needed in the Linux version to support kernel mode building blocks and to handle interrupts. Under VxWorks, the Resource Manager is implemented as a library and a task to support communication with the microengines.

Figure 7-1. Internal Design of the IXA SDK Resource Manager



As shown in Figure 7-1, the building block infrastructure uses the APIs exported by the Resource Manager. Applications written outside of the building block infrastructure can directly access and use the Resource Manager API.

The IXA SDK Resource Manager API may be functionally grouped as shown in Table 7-1.

Table 7-1. Resource Manager API Functional Groups

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

7.4 API

This section provides details on the IXA SDK Resource Manager API. The function prototypes are provided for completeness of the document only. For more details on the Resource Manager API, see Section 3, “Resource Manager,” in the *Intel® Internet Exchange Architecture Portability Framework Reference Manual*.

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

7.4.1 Basic Types

The Resource Manager defines a set of basic types that are used across the entire SDK. These types are defined to ease portability of the API across different operating systems, compilers, etc. These types are listed in Table 7-2.

Table 7-2. Basic Types Supported by the Resource Manager

Basic Types	Description
ix_int8	An 8-bit signed integer.
ix_uint8	An 8-bit unsigned integer.
ix_int16	A 16-bit signed integer.
ix_uint16	A 16-bit unsigned integer.
ix_int32	A 32-bit signed integer.
ix_uint32	A 32-bit unsigned integer.
ix_int64	A 64-bit signed integer.
ix_uint64	A 64-bit unsigned integer.
ix_uint128	A 128-bit unsigned integer. This type does not support full arithmetic operations. However a set of macros has been provided to support this type.
ix_bit_mask8	An 8-bit bit mask.
ix_bit_mask16	A 16-bit bit mask.
ix_bit_mask32	A 32-bit bit mask.
ix_bit_mask64	A 64-bit bit mask.
ix_handle	Generic handle type used throughout the framework API. All handle types are aliases of this handle type.
ix_error	Error token used through out the Intel® IXA SDK 3.x. All Intel® IXA SDK 3.x functions return this type. This is a 32-bit unsigned integer that has packed a 16-bit error code, an 8-bit error group and an 8-bit error level. Macros are provided for creating an error token and for accessing the different fields.
_IX_OS_TYPE_	Preprocessor symbol that indicates for which operating system the SDK is currently compiled. At the moment just four values are defined, but others may be added later: • _IX_OS_VXWORKS_ • _IX_OS_LINUX_KERNEL_ • _IX_OS_LINUX_USER_ • _IX_OS_WIN32_ **NOTE:** _IX_OS_WIN32_ is used for debug with foreign model or for Win32 simulation.

7.4.2 System API

The Resource Manager System API provides functions to initialize and terminate the Resource Manager, to get and set the hardware configuration, and so on. For more details on this API, see Section 3.2, "System API," in the *Intel® Internet Exchange Architecture Portability Framework Reference Manual*. Table 7-3 lists the functions and data structures in this API.

Table 7-3. Resource Manager System API

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

7.4.3 Microengine API

For more details on this API, see Section 3.3, "Microengine API," in the *Intel® Internet Exchange Architecture Portability Framework Reference Manual*. Table 7-4 lists the functions and data structures in the MicroEngine API.

Table 7-4. Resource Manager Microengine API

Function or Data Structure	Description
`ix_imported_symbol`	The structure represents a microcode symbol.
`ix_rm_ueng_set_ucode()`	Sets the microcode image for microengines from a file.
`ix_rm_ueng_map_ucode()`	Sets the microcode image for microengines from a buffer.
`ix_rm_ueng_reset_all()`	Stops and resets all active microengines.
`ix_rm_ueng_patch_symbols()`	Patches symbols.

Table 7-4. Resource Manager Microengine API (Continued)

Function or Data Structure	Description
ix_rm_ueng_load()	Loads the microcode into the microstore of the microengines.
ix_rm_ueng_start()	Starts the specified microengines.
ix_rm_ueng_stop()	Stops the specified microengines.
ix_rm_ueng_reset()	Resets the specified microengines.
ix_rm_ueng_enable()	Enables the specified microengines.
ix_rm_ueng_disable()	Disables the specified microengines.

7.4.4 Hardware Resource Management API

This section discusses API calls to manage MEv2 hardware features including SRAM queues, rings, scratch rings, and so on. This API may be extended in the future to support reservation of RBUFs, TBUFs, and other hardware features. For more details on this API, see Section 3.4, "Hardware Resource Management API," in the *Intel® Internet Exchange Architecture Portability Framework Reference Manual*.

Table 7-5 lists the functions and data structures in the Hardware Resource Management API.

Table 7-5. Resource Manager Hardware API

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

7.4.4.1 SRAM Queues

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

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

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

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

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

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

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

The handle is an alias of the generic `ix_handle` type, and is encoded as an `ix_hw_queue_handle`.

7.4.4.2 SRAM and Scratch Rings

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

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

SRAM rings are supported by the Q-array. The number of rings supported is restricted only by the entries free in the Q-array.

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

Handles

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

Bit-Field Macros

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

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

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

Memory Type Macros

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

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

Ring Index Macros

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

7.4.5 Buffer Management API

The Buffer API has two parts:

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

- An API that is used to access fields in the packet descriptor—the packet metadata. This API is specific to the layout and fields of the packet descriptor.

For more details on this API, see Section 3.5, "Buffer Management API," in the *Intel® Internet Exchange Architecture Portability Framework Reference Manual*.

7.4.5.1 Generic Buffer API

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

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

This section describes the generic buffer API independent of the microblock packet descriptor layout.

Table 7-6 lists the functions and data structures in the Buffer Management API.

Table 7-6. Resource Manager Buffer Management API

Function or Data Structure	Description
`ix_buffer_handle`	A buffer handle.
`ix_buffer_free_list_handle`	A buffer free list handle.
`ix_buffer_free_list_info`	Buffer free list data structure.
`ix_buffer_type`	Enumerated type specifying the type of the buffer—hardware or software.
`ix_rm_hw_buffer_free_list_create()`	Creates a hardware buffer free list.
`ix_rm_sw_buffer_free_list_create()`	Creates a software buffer free list.
`ix_rm_buffer_free_list_delete()`	Deletes a buffer free list.
`ix_rm_buffer_free_list_get_info()`	Retrieves information about a free list.
`ix_rm_buffer_alloc()`	Allocates a buffer.
`ix_rm_buffer_free()`	Frees a buffer.
`ix_rm_buffer_free_chain()`	Frees and returns a buffer in a chain to the correct buffer free list.
`ix_rm_buffer_get_meta()`	Returns the metadata for a buffer.
`ix_rm_buffer_get_data()`	Returns the data associated with a buffer.

Table 7-6. Resource Manager Buffer Management API (Continued)

Function or Data Structure	Description
`ix_rm_buffer_is_eop()`	Determines if the buffer is the last one in a chain.
`ix_rm_buffer_is_sop()`	Determines if the buffer is the first one in a chain.
`ix_rm_buffer_get_type()`	Determines the type of a buffer—either hardware or software.
`ix_rm_buffer_get_next()`	Returns the next buffer in a chain.
`ix_rm_buffer_link()`	Links two buffers into a chain.
`ix_rm_buffer_unlink()`	Breaks a linked list chain.

Note: These functions are for use in the Intel XScale® core. In the microengines, the IXP Microengine Assembler and Microengine C library `xbuf` API is used to allocate and access buffers. Compatibility should be maintained in implementation so that a buffer-free list can be created using the Resource Manager, and still allocate and free buffers using the IXP library in the microengines.

7.4.5.2 IXA Portability Framework Buffer API

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

The IXA Portability Framework also specifies the layout of the packet meta data. If these need to be changed by an application, then the application developer needs to change the associated dispatch loop macros and the Resource Manager library.

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

Table 7-7 lists these packet meta data fields. In this data structure, some of the fields are common to all applications. Some of the fields are specific to a particular category of applications. Users may trim (or add) fields to customize the meta-data to their application category. For further information, see the *Intel® Internet Exchange Architecture Portability Framework Reference Manual*.

Table 7-7. Resource Manager Packet Meta Data Definitions

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

1. The data member, m_BufferInfo, is a packed buffer containing the bit fields free_list_id, rx_stat, and header_type. The appropriate buffer API macros should be used to extract these bit fields from the packed buffer.

7.4.6 Communication API

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

- The calling application polls a communication ID for messages or packets.
- The calling application retrieves messages and packets from a communication ID using a synchronous function that blocks until a message or a packet arrives or the call times out.

- The calling application specifies interest in several communication IDs and the call returns only when one of the specified IDs receives data.
- The calling application registers a callback function that is called whenever new data arrives on a communication ID.

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

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

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

The communication between microblocks and core is handled in the following way. The microcode queues data onto a common hardware ring and signals the Intel XScale® core that data has been sent. For packet communication the scratch ring with the ID of zero is used to queue data, and the core is signaled through `Thread_Interrupt_A_#` (see *Intel® IXP2400 Network Processor Hardware Reference Manual*, *Intel XScale®* core *Gasket* Chapter). For message communication, the scratch ring with ID of one is used to queue data and the Intel XScale® core is signaled through `Thread_Interrupt_B_#`. On the core side, the registered ISRs awaken the corresponding dispatch threads that dequeue the data and send it to the requested destination. The message-dispatch thread has higher priority than the packet-dispatch thread.

For more details on this API, see Section 3.6, "Communication API," in the *Intel® Internet Exchange Architecture Portability Framework Reference Manual*. Table 7-8 lists the functions and data structures in the Communication API.

Table 7-8. Resource Manager Communication API

Communication API and Data Structures	Description
`ix_comm_data_handler`	Generic function type for data communication handler.
`ix_communication_id`	Generic type used for the identification of a communication point.
`ix_comm_select_action_set`	Array of core local communication ID masks that is passed to the select function.
`ix_comm_id_mode`	Enumerated type expressing a communication ID receive mode.
`IX_RM_COMM_ID_GET_LOCAL_ID()`	Returns the local ID for a communication ID.
`IX_RM_COMM_ID_GET_SYSTEM_TYPE()`	Returns the subsystem type for a communication ID.
`IX_RM_COMM_ID_GET_BLADE_ID()`	Returns the blade ID for a communication ID.
`IX_RM_COMM_MAKE_ID()`	Creates a communication ID.
`IX_RM_COMM_MAKE_LOCAL_ID()`	Creates a local subsystem communication ID.
`ix_rm_packet_set_receive_mode()`	Sets the packet receive mode for the communication ID.
`ix_rm_message_set_receive_mode()`	Sets the message receive mode for the communication ID.

Table 7-8. Resource Manager Communication API (Continued)

ix_rm_packet_set_consumer_mode()	Sets the packet consumer mode for the communication ID.
ix_rm_message_set_consumer_mode()	Sets the message consumer mode for the communication ID.
ix_rm_packet_set_producer_mode()	Sets the packet producer mode for the communication ID.
ix_rm_message_set_producer_mode()	Sets the message producer mode for the communication ID.
ix_rm_packet_handler_register()	Registers a packet handler with a core communication ID.
ix_rm_message_handler_register()	Registers a message handler with a core communication ID.
ix_rm_packet_handler_unregister()	Puts the packet processing for the communication ID in default mode—it drops packets.
ix_rm_message_handler_unregister()	Puts the message processing for the communication ID in default mode—it drops packets.
ix_rm_packet_send()	Sends a packet to a destination.
ix_rm_packet_send_wait()	Sends a packet to a destination in a blocking mode.
ix_rm_message_send()	Sends a message to a destination.
ix_rm_message_send_wait()	Sends a message to a destination in a blocking mode.
ix_rm_packet_peek()	Retrieves the number of packets stored in the internal queue for the specified communication ID.
ix_rm_packet_get()	Retrieves a packet from a communication ID in a non-blocking mode.
ix_rm_packet_get_wait()	Retrieves a packet from a communication ID in a blocking mode.
ix_rm_message_peek()	Retrieves the number of messages stored in the internal queue for the specified communication ID.
ix_rm_message_get()	Retrieves a message from a communication ID in a non-blocking mode.
ix_rm_message_get_wait()	Retrieves a message from a communication ID in a blocking mode.
ix_rm_comm_select()	Waits on a set of communication IDs for data to be available.
ix_rm_ublock_packet_comm_init()	Initializes the packet communication to a microblock.
ix_rm_ublock_message_comm_init()	Initializes the message communication to a microblock.

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

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

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

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

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

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

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

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

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

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

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

7.4.7 Remote Communication Extension API

The communication ID mechanism allows for local communication, as well as remote communication with other systems. If the destination is not intended for the local system, then the messages and packets will be forwarded to the active remote communication service that routes the data through PCI or another communication pathway to the remote system. For more details on this API, see Section 3.7, “Remote Communication Extension API,” in the *Intel® Internet Exchange Architecture Portability Framework Reference Manual*.

Table 7-9 summarizes the remote communications API.

Table 7-9. Resource Manager Remote Communication Extension API

Name	Description
`ix_remote_comm_service`	This structure defines a remote communication service.
`ix_remote_comm_data_handler`	This callback function prototype represents a generic remote data handler.
`ix_remote_comm_service_initializer`	Defines the generic remote communication service initializer.
`ix_remote_comm_service_finalizer`	Defines the generic remote communication service finalizer.

Table 7-9. Resource Manager Remote Communication Extension API (Continued)

Name	Description
`ix_rm_remote_comm_service_register()`	Registers a remote communication service.
`ix_rm_remote_comm_service_unregister()`	Unregisters the active remote communication service.
`ix_rm_init_pci_remote_communication()`	Installs the PCI remote communication service between the ingress and egress network processors for a single ingress/egress system.
`ix_rm_register_pci_communication_hw_free_list()`	Registers a hardware free list of choice to be used by the predefined dual single system PCI remote communication service.
`ix_rm_unregister_pci_communication_hw_free_list()`	Reverts to the default hardware free list to be used by the predefined dual single system PCI remote communication service.

7.4.8 Memory Management API

The Resource Manager manages SRAM, DRAM, scratch, and local memory for the IXP2400 and Intel® IXP2800 Network Processors. It exports an interface to allocate, access, and free memory chunks. The Resource Manager owns SRAM and scratch memory completely and DRAM partially. The DRAM is partly owned by the operating system.

The memory managed by the Resource Manager is used to support system-wide data structures such as the buffer-free pools, control blocks for building blocks, route table, trie table, and so on. The difference between this memory and memory allocated from the operating system is that this memory has no MMU protection and is always addressed at the same memory location by all processes. This has meaning only in Linux. In VxWorks, the memory model is a flat memory space shared by every task. Since this memory is shared with the microengines, it is typically uncached.

All applications, including the building block infrastructure, must use this API to allocate memory in order to work in conjunction with the microengines. The operating system memory is not accessible from microcode.

Note: The Resource Manager memory management is designed to handle one-time memory allocation primarily to partition memory among the applications. In other words, do not attempt to re-implement `malloc()`. Applications that require handling a large number of allocation and free operations dynamically need to obtain enough memory from the Resource Manager and manage it themselves.

The Resource Manager allocates memory so that DRAM is always returned aligned at an 8-byte boundary and any request is rounded off to an 8-byte boundary. For SRAM and scratch, the alignment is always at a 4-byte boundary and all requests are rounded off to a 4-byte boundary.

Functions are provided to calculate the physical offset into the specific channel and physical address of the chunk allocated from the virtual address and vice versa. The microengines can only access memory using these offset values into a specific channel. The physical memory is contiguous—on a specific channel—for the IXP2400 and Intel® IXP2800 Network Processors.

Handling SRAM allocation requires supporting multiple channels. The memory management API takes the channel number as one of the inputs. For the Intel® IXP2400 Network Processor, the valid channel numbers are zero and one. For the Intel® IXP2800 Network Processor, the valid channel numbers are zero through three, inclusive.

The Intel® IXP2800 Network Processor, supports three DRAM channels, while Intel® IXP2400 Network Processor only supports one.

At the time of initialization, the Resource Manager can reserve space in SRAM, scratch, SDRAM, and local memory to support Microengine C. The linker (UCLD) requires a base address for these blocks of memory, but no provision exists to limit the size of these memory chunks. The Resource Manager gets information from the loader (UCLO) library about the memory required by the loaded microcode image and checks if that memory has been reserved at Resource Manager initialization. If any of the required areas have not been reserved, then an error is returned along with the memory areas that must be reserved for microcode usage.

For more details on this API, see Section 3.8, "Memory Management API," in the *Intel® Internet Exchange Architecture Portability Framework Reference Manual*. Table 7-10 lists the structures and functions in the Memory Management API. Table 7-11 lists the macros in the Memory Management API.

Table 7-10. Resource Manager Memory Management API

Memory API and Data Structures	Description
`ix_memory_type`	An enumerated type listing the types of memory supported by the Resource Manager.
`ix_memory_info`	Memory information data structure.
`ix_memory_alignment_type`	Alignment types for the aligned memory allocation and reservation calls.
`ix_rm_mem_alloc()`	Allocates memory—SRAM, DRAM, and scratch.
`ix_rm_mem_alloc_aligned()`	Allocates memory with alignment—SRAM, DRAM, and scratch.
`ix_rm_mem_reserve()`	Reserves memory.
`ix_rm_mem_reserve_aligned()`	Reserves memory with alignment.
`ix_rm_mem_free()`	Frees memory.
`ix_rm_mem_info()`	Retrieves memory information for the specified memory type and specified channel.
`ix_rm_mem_local_alloc()`	Allocates local memory.
`ix_rm_mem_local_reserve()`	Reserves local memory
`ix_rm_mem_local_free()`	Frees local memory.
`ix_rm_mem_local_info()`	Retrieves local memory information.
`Read/Write Macros`	Macros to read and write memory locations.
`ix_rm_get_phys_offset()`	Returns the physical offset of a memory block.
`ix_rm_get_virtual_address()`	Returns the virtual address of a memory block.

Table 7-11. Resource Manager Memory Management Macros

Macro Name	Description
`IX_RM_MEM_UINT8_READ`	Returns an `ix_uint8` representing the value at the specified location.
`IX_RM_MEM_UINT16_READ`	Returns an `ix_uint16` representing the value at the specified location.
`IX_RM_MEM_UINT32_READ`	Returns an `ix_uint32` representing the value at the specified location.

Table 7-11. Resource Manager Memory Management Macros (Continued)

Macro Name	Description
IX_RM_MEM_UINT64_READ	Returns an ix_uint64 representing the value at the specified location.
IX_RM_MEM_UINT8_WRITE	Writes an ix_uint8 value to the specified location.
IX_RM_MEM_UINT16_WRITE	Writes an ix_uint16 value to the specified location.
IX_RM_MEM_UINT32_WRITE	Writes an ix_uint32 value to the specified location.
IX_RM_MEM_UINT64_WRITE	Writes an ix_uint64 value to the specified location.

7.4.9 System Repository API

The system repository is designed as a collection of tree constructs that store system properties. The structure and navigation of these tree constructs is similar to a file system structure.

Properties represent name-value entities that can be set and accessed throughout the system in a consistent manner. Configuration properties are defined by property handles that link the name and value pair together. There is a limit on the number of properties that can be created in the system. Each property has a set of attributes associated with it. There could be properties that can act just as nodes in the hierarchy and that have no value associated with them. Once a property has been created, the calling application can register with the property to receive notifications if the property changes. Every application in the system can create properties at a certain node level, and retrieve and modify them.

The core software property tree stores all the properties of the applications residing on the core side. The configuration property handle corresponding to the root of this tree is IX_CP_CORE_PROPERTY_ROOT.

For more details on this API, see Section 3.9, "System Repository API," in the *Intel® Internet Exchange Architecture Portability Framework Reference Manual*. Table 7-12 lists the data structures and functions that make up the System Repository API. I

Table 7-12. Resource Manager System Repository API

System Repository API and Data Structures	Description
ix_configuration_property_handle	Generic handle type for configuration properties.
ix_cp_property_info	Configuration property information structure.
ix_rm_cp_property_create()	Creates a new configuration property at a certain node level.
ix_rm_cp_property_delete()	Deletes a configuration property.
ix_rm_cp_property_open()	Retrieves a configuration property based on a base node and a name.
ix_rm_cp_property_close()	Invalidates a configuration property handle.
ix_rm_cp_property_attach()	A communication ID is registered with the property to receive change notifications.
ix_rm_cp_property_detach()	A communication ID is unregistered from the property notification list.
ix_rm_cp_property_set_value()	A value is associated with a configuration property or the previous value is replaced with the new one.
ix_rm_cp_property_get_value()	Retrieves a property value.

Table 7-12. Resource Manager System Repository API (Continued)

System Repository API and Data Structures	Description
ix_rm_cp_property_set_value_uint32()	A 32-bit unsigned value is associated with a configuration property or the previous value is replaced with the new one.
ix_rm_cp_property_get_value_uint32()	Retrieves a 32-bit unsigned property value.
ix_rm_cp_property_delete_value()	Deletes the value associated with a property.
ix_rm_cp_property_get_info()	Gets information pertaining to a configuration property.
ix_rm_cp_property_get_subproperty()	Navigates a subtree of a configuration property.

7.4.10 64-Bit Counters

The 64-bit counter API is provided as statistics support for the Intel XScale® core applications. *RFC2863* states that 64-bit counters must be supported for interfaces that operate at data rates greater than 650Mbps.

The design of these counters is such that for each 64-bit counter there is a corresponding 32-bit counter residing in SRAM or SCRATCH that can be updated atomically by the microengines. Each counter has an associated overflow time for the internal 32-bit counter. The API has a background thread that monitors all the overflow times for the internal 32-bit counters and do an atomic read/set to 0 operation (atomic swap) for the internal 32-bit counters, and then update the 64-bit counterparts accordingly. The memory necessary for the internal counters is allocated by the callers as they need to know what values need to be patched into the microcode.

For more details on this API, see Section 3.10, "64-Bit Counters API," in the *Intel® Internet Exchange Architecture Portability Framework Reference Manual*. Table 7-13 lists the functions that make up the 64-bit counter API

Table 7-13. Resource Manager 64-Bit Counter API

Name	Description
ix_counter_64bit_handle()	Generic type for a 64-bit counter handle.
ix_rm_counter_64bit_new()	Allocates an array of 64-bit counters.
ix_rm_counter_64bit_delete()	Deletes a 64-bit counter.
ix_rm_counter_64bit_get_internal_overflow_time()	Retrieves the overflow time for the internal 32-bit counter of the specified 64-bit counter.
ix_rm_counter_64bit_set_internal_overflow_time()	Sets the overflow time for the internal 32-bit counter of the specified 64-bit counter.
ix_rm_counter_64bit_get_value()	Returns the 64-bit core value of the counter.
ix_rm_counter_64bit_set_value()	Sets the 64-bit core value of the counter.

7.4.11 Services

The Services API provides a set of functions that allows core applications to take advantage of certain hardware features. The API provides a set of atomic operations and fast memory operations. For more details on this API, see Section 3.11, "Services API," in the *Intel® Internet Exchange Architecture Portability Framework Reference Manual*. Table 7-14 lists data structures and functions included in this API.

Table 7-14. Resource Manager Services API

Functions	Description
`ix_rm_atomic_sram_swap()`	Performs an atomic swap between an SRAM memory location and one arbitrary memory location.
`ix_rm_atomic_sram_add()`	Performs an atomic add to an SRAM memory location.
`ix_rm_atomic_sram_test_and_add()`	Performs an atomic add to an SRAM memory location and returns the value stored at the memory location before add operation.
`ix_rm_atomic_sram_subtract()`	Performs an atomic subtract to an SRAM memory location.
`ix_rm_atomic_sram_test_and_subtract()`	Performs an atomic subtract to an SRAM memory location and returns the value stored at the memory location before the subtract operation.
`ix_rm_atomic_sram_bit_set()`	Performs an atomic bit set operation to an SRAM memory location.
`ix_rm_atomic_sram_bit_test_and_set()`	Performs an atomic bit set operation to an SRAM memory location and returns the value stored at the memory location before the bit set operation.
`ix_rm_atomic_sram_bit_clear()`	Performs an atomic bit clear operation to an SRAM memory location.
`ix_rm_atomic_sram_bit_test_and_clear()`	Performs an atomic bit clear operation to an SRAM memory location and returns the value stored at the memory location before the bit clear operation.
`ix_rm_atomic_scratch_swap()`	Performs an atomic swap between a SCRATCH memory location and one arbitrary memory location.
`ix_rm_atomic_scratch_add()`	Performs an atomic add to a SCRATCH memory location.
`ix_rm_atomic_scratch_test_and_add()`	Performs an atomic add to a SCRATCH memory location and returns the value stored at the memory location before the add operation.
`ix_rm_atomic_scratch_subtract()`	Performs an atomic subtract to a SCRATCH memory location.
`ix_rm_atomic_scratch_test_and_subtract()`	Performs an atomic subtract to a SCRATCH memory location and returns the value stored at the memory location before the subtract operation.
`ix_rm_atomic_scratch_bit_set()`	Performs an atomic bit set operation to a SCRATCH memory location.
`ix_rm_atomic_scratch_bit_test_and_set()`	Performs an atomic bit set operation to a SCRATCH memory location and returns the value stored at the memory location before the bit set operation.
`ix_rm_atomic_scratch_bit_clear()`	Performs an atomic bit clear operation to a SCRATCH memory location.
`ix_rm_atomic_scratch_bit_test_and_clear()`	Performs an atomic bit clear operation to a SCRATCH memory location and returns the value stored at the memory location before the bit clear operation.

Table 7-14. Resource Manager Services API (Continued)

ix_rm_managed_to_os_memory_copy()	Performs a fast memory copy from a managed memory location to an OS memory location.
ix_rm_os_to_managed_memory_copy()	Performs a fast memory copy from an OS memory location to a managed memory location.
ix_rm_managed_to_managed_memory_copy()	Performs a fast memory copy from a managed memory location to another managed memory location.

7.4.12 Hash

The Hash API provides hash operations for Intel XScale® core applications. Programmers can choose to perform the hash operations with hardware support from the chip's hash unit or in software. By default, the resource manager uses hardware support from the hash unit. In order to change this behavior, the _IX_RM_IMPL_SOFTWARE_HASH_ symbol must be defined on the command line at the time the Resource Manager is compiled.

The hash algorithm is explained in great detail in the *Intel® IXP2400/IXP2800 Network Processor Programmer's Reference Manual* on the IXA SDK Tools CD.

For more details on this API, see Section 3.12, "Hash API," in the *Intel® Internet Exchange Architecture Portability Framework Reference Manual*. Table 7-15 summarizes the Hash API.

Table 7-15. Resource Manager Hash API

Name	Description
ix_hash_48	This type represents a 48-bit hash data type.
ix_hash_64	This type represents a 64-bit hash data type.
ix_hash_128	This type defines the 128-bit hash data type.
ix_hash_multiplier_48	This type defines the 48-bit multiplier data type.
ix_hash_multiplier_64	This type defines the 64-bit multiplier data type.
ix_hash_multiplier_128	This type defines the 128-bit multiplier data type.
ix_rm_hash_48_hash()	This function performs a 48-bit hash operation.
ix_rm_hash_48_multiplier_set()	This function sets a new multiplier value for the 48-bit hash operations.
ix_rm_hash_48_multiplier_get()	The function retrieves the current multiplier value for 48-bit hash operations.
ix_rm_hash_64_hash()	This function performs a 64-bit hash operation.
ix_rm_hash_64_multiplier_set()	This function sets a new multiplier value for the 64-bit hash operations.
ix_rm_hash_64_multiplier_get()	The function retrieves the current multiplier value for 64-bit hash operations.
ix_rm_hash_128_hash()	This function performs a 128-bit hash operation.
ix_rm_hash_128_multiplier_set()	This function sets a new multiplier value for the 128-bit hash operations.
ix_rm_hash_128_multiplier_get()	The function retrieves the current multiplier value for 128-bit hash operations.

7.4.13 Microengine Services

The Microengine Services API provides several sets of APIs, which are used to coordinate operations between the Intel XScale® core and the microengines. The following operations are supported:

- Locking and unlocking mechanisms
- Reading and writing of microengine transfer registers
- Sending a notification signal to a microengine

Note: In some places in this document, microengine is abbreviated as ME.

For more details on this API, see Section 3.13, "Microengine Services API," in the *Intel® Internet Exchange Architecture Portability Framework Reference Manual*. Table 7-16 lists data structures and functions included in this API.

Table 7-16. Resource Manager Microengine Services API

Functions and Data Structures	Description
`ix_me_xscale_lock_handle`	A microengine-to-Intel XScale® core lock handle
`ix_me_xscale_lock_status`	Type describing all possible states of a microengine-to-Intel XScale® core lock.
`ix_me_xscale_lock_owner`	Type describing all possible owners of a microengine-to-Intel XScale® core lock.
`ix_me_xscale_lock_info`	Structure providing information about a microengine-to-Intel XScale® core lock.
`ix_me_transfer_register_type`	Enumerated type describing all types of ME transfer registers.
`ix_rm_me_xscale_lock_new()`	Creates a new microengine-to-Intel XScale® core lock object.
`ix_rm_me_xscale_lock_delete()`	Deletes the specified microengine-to-Intel XScale® core lock object.
`ix_rm_me_xscale_lock_acquire()`	Acquires the specified microengine-to-Intel XScale® core lock.
`ix_rm_me_xscale_lock_release()`	Releases the specified microengine-to-Intel XScale® core lock.
`ix_rm_me_xscale_lock_get_info()`	Returns useful information about a microengine-to-Intel XScale® core lock.
`ix_rm_me_transfer_register_read()`	Reads the value of a ME transfer register.
`ix_rm_me_transfer_register_write()`	Writes the value of a ME transfer register.
`ix_rm_me_signal()`	Sends a specified signal to a ME.

7.4.14 Debug Support

The Debug Support API provides a series of functions that provide debugging features to the programmers. In order for debug features to be turned on, the resource manager library must be compiled with the _IX_RM_DEBUG_ preprocessor symbol defined. For the debug builds, this particular symbol is automatically defined. This symbol can be turned off if debug functions are not needed.

For more details on this API, see Section 3.14, "Debug Support API," in the *Intel® Internet Exchange Architecture Portability Framework Reference Manual*. Table 7-17 lists functions included in this API.

Table 7-17. Resource Manager Debug Support API

Functions	Description
`ix_rm_mem_status_print()`	Prints information about a memory manager associated with a certain memory type and channel.
`ix_rm_scratch_ring_print_info()`	Prints information about a SCRATCH ring.
`ix_rm_scratch_ring_print_data()`	Prints the data belonging to a SCRATCH ring.
`ix_rm_sram_ring_print_info()`	Prints information about an SRAM ring.

Table 7-17. Resource Manager Debug Support API (Continued)

Functions	Description
ix_rm_sram_ring_print_data()	Prints the data belonging to an SRAM ring.
ix_rm_free_list_print_available_buffers()	Prints the number of available buffers in a certain free list.
ix_rm_free_list_print_buffers_info()	Prints the allocation information for the buffers belonging to a certain hardware free list.
ix_rm_free_list_print_info()	Prints information related to certain free list.
ix_rm_buffer_print_meta()	Prints the buffer meta information for the passed handle.
ix_rm_buffer_print_data()	Prints the buffer data for the passed handle.
ix_rm_buffer_print_debug_info()	Prints the buffer debug information for the passed handle.

Core Components 8

8.1 Overview

A core component is the slow-path counterpart of the microblock running on the Intel XScale® core. A core component performs the following functions:

- Configures its microblock (static configuration by means of imported variables and dynamic configuration through control blocks).
- Initializes and maintains common data structures that may be updated by other applications.
- Provides exception as well as control message handler to process packets/messages sent by the microblock.

Each core component can have multiple inputs, each of which is associated with a different packet/message handler.

In general, there is a single core component associated with each microblock. The IPv4 Forwarder Core Component services exception packets sent to it by the IPv4 microblock. However, a core component may also manage more than one microblock. In the extreme case there may be a single core component for all the microblocks.

There are two ways to implement a core component. One way is to implement the core component using the IXA Core Component Infrastructure Library (see Chapter 10, “Core Component Infrastructure”). The core component infrastructure provides support for handling messages and packets.

The other way is to implement the core component as a software entity that directly uses the Resource Manager API (see Chapter 7, “Resource Manager”). The design of this entity (whether it is a shared library, driver, thread, process etc.) and how it processes packets and messages is left entirely to the developer.

This provides developers with considerable flexibility in integrating applications written using the IXA Portability Framework with existing legacy applications and protocol stacks running on the Intel XScale® core.

Developers writing mostly new code on the Intel XScale® core would prefer the accelerated development time provided by the IXA Portability Framework software infrastructure. Customers with a substantial legacy code base would probably prefer to ease integration with existing code by not writing the core component as an IXA Building Block. They can still use the infrastructure on the microengines and the use Resource Manager API to interface with it.

TCAM Lookup Libraries 9

This chapter describes a common API used for managing and searching tables on the Intel XScale® core and on the microengines for Intel® IXP2400 and IXP2800 Network Processors.

The lookup library provides a way of managing different search and lookup tables that can be used for many different networking applications. The goal of the search table is to hide the details of both the data structures and the underlying hardware implementation from the application designer. This abstraction allows the addition of different data structures as well as hardware assisted search devices, for example TCAM (Ternary Content Addressable Memory), to be used without rewriting the application itself.

Figure 9-1 shows the different components of the lookup library. The lookup management library runs on the Intel XScale® core. This library is used for creating and deleting tables as well as adding, deleting, and modifying entries to each of the tables. The second component is the microengine lookup library that runs on the microengine. This library is used by the data plane code to perform rapid searches on the data to classify packets and determine how to forward them. Because of the differences in accessing the tables for software and hardware classification, there are separate lookup APIs on the microengines. This is discussed in more detail in Section 9.2, “Microengine Lookup Library” .

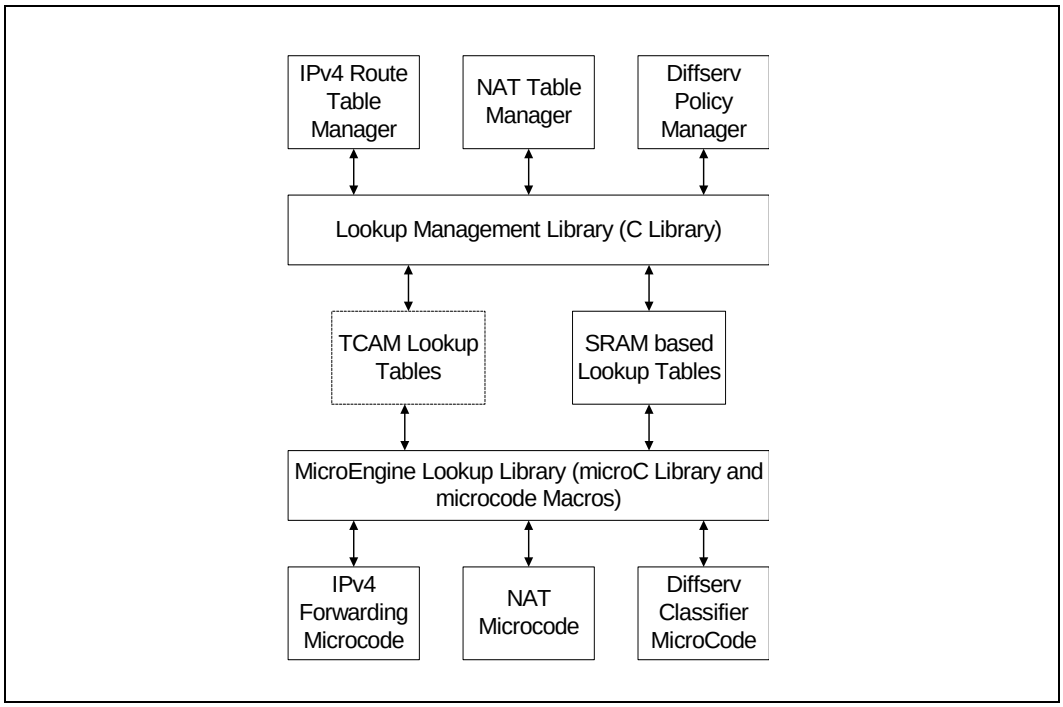
Entries in tables are specified using a key and a mask value. When searching through the table with search_key, an entry matches if you take the search key and logically AND it with the mask value and the resulting data matches the entry key. For this to work, the key, mask, and search_key must all be the same size.

Because different entries may have different masks and search_key may match multiple entries, each entry has an additional weight value that is used to break ties when multiple entries match a search key. The entry with the lowest weight wins. Matching multiple entries with the same weight returns undefined results.

It is assumed that any shared NPU control and status registers are correctly configured by an external entity prior to the calling of this API. (For example, the NPU SRAM controllers are expected to be correctly configured by the calling application, etc.)

The following sub-sections discuss more details about each of the libraries.

Figure 9-1. Library Components



9.1 Lookup Management Library

The management library runs on the core processor. This is used to initialize and create tables as well as adding and removing entries from the tables. These libraries are typically called from higher-level applications such as a route table manager, a NAT table manager, or program that is managing diffserv policies. Table 9-1 lists lookup handles, data structures, and the APIs included in the Lookup Management library. For complete descriptions and definitions of the API functions, see the *Intel® Internet Exchange Architecture Portability Framework Reference Manual*.

Table 9-1. Handles, Data Structures, and Lookup Management APIs

Name	Description
ix_lkup	Handle that is returned when the application first initializes and gets a handle to the lookup library.
ix_lkup_table	Handle that is returned when a new table is created by calling IX_LKUP_CREATE_TABLE () on a valid ix_lkup.
ix_lkup_table_type	Defines the different table types and the associated search methods for that table.
ix_lkup_tcam_params	Data structure that is passed when initializing the TCAM version of the library by calling ix_lkup_tcam_init().
ix_lkup_table_conf	Data structure used to pass the configuration parameters when a new table is created.
ix_lkup_cookie	Opaque cookie that is passed between some of the API calls, primarily calls that are used for enumerating the contents of a table.

Table 9-1. Handles, Data Structures, and Lookup Management APIs (Continued)

Name	Description
ix_lkup_sw_init()	Initializes the software lookup management library and gets a handle to the library for subsequent operations.
ix_lkup_tcam_init()	Initializes the TCAM lookup management library and returns a handle to the library for subsequent operations.
IX_LKUP_CREATE_TABLE( )	Creates a new instance of a search table.
IX_LKUP_DESTROY_TABLE()	Destroys a table that was created earlier with IX_LKUP_CREATE_TABLE.
IX_LKUP_FINI()	Destroys a previously obtained ix_lkup handle.
IX_LKUP_ADD_ENTRY()	Adds an entry to a table.
IX_LKUP_REMOVE_ENTRY()	Removes an entry from a table.
IX_LKUP_UPDATE_ENTRY()	Updates the data associated with an element already in the table.
IX_LKUP_SEARCH_TABLE()	Searches a specific table and returns the associated data if a match is found.
IX_LKUP_FIND_ENTRY()	Searches a table for an entry that matches the exact key and mask, weight combination.
IX_LKUP_READ_FIRST_ENTRY()	Retrieves the first item stored in the table.
IX_LKUP_READ_NEXT_ENTRY()	Retrieves successive items stored in the table.
IX_LKUP_RESET_TABLE()	Clears all the items in the table and resets it to its initial state.
IX_LKUP_SET_PROPERTY()	Allows the caller to set special attributes of the table.
IX_LKUP_GET_PROPERTY()	Allows the caller to get special attributes of the table.
IX_LKUP_GET_TABLE_INFO()	Returns table identifier and data information.

9.2 Microengine Lookup Library

The microengine lookup libraries are used by an application running on the microengines to search tables that are created and managed by the core applications. These libraries are provided as microengine C and microengine Assembler macros.

There are two distinct libraries for searching the tables; one for software based searches and one for hardware based searches. While it would have been possible to provide the same interfaces for both hardware and software, the interfaces may have been awkward to use and for applications to get the best possible performance, the decision was made to make the APIs as natural and as optimal as possible. Table 9-2 lists all the APIs included in the Microengine Lookup library. For complete descriptions and definitions of the API functions, see the *Intel® Internet Exchange Architecture Portability Framework Reference Manual*.

Table 9-2. Microengine Lookup APIs

Name	Description
ix_tcam_lkup_build_handle()	Builds the handle that must be passed to the search functions.
ix_tcam_lkup_start()	Starts a search using the in_key to launch the search request.
ix_tcam_lkup_complete()	Completes a search that was started and returns the results.
ix_tcam_lkup_get_data()	Returns the data associated with a successful search.

Table 9-2. Microengine Lookup APIs (Continued)

Name	Description
ix_sw_lkup_lpm_build_handle()	Builds the handle that must be passed to the search functions for longest prefix match searching.
ix_sw_lkup_lpm_search()	Searches a longest prefix match table and returns the results.
ix_sw_lkup_exact_build_handle()	Builds the handle that must be passed to the search functions for exact match searching.
ix_sw_lkup_exact_search()	Searches an exact match table and returns the results.
ix_sw_lkup_range_build_handle()	Builds the handle that must be passed to the search functions for range match searching.
ix_sw_lkup_range_search()	Searches a range match table and returns the results.
ix_s_lkup	Data structure that all implementations need to fill out and return when the library is initialized.
ix_s_lkup_table	Data structure that all implementations need to fill out and return when a table is created.

Core Component Infrastructure 10

10.1 Terminology and Key Components of the Core Component Infrastructure

The following chapter describes the Core Component Infrastructure interface. This interface provides framework support for the following functionality:

- Each core component runs on its own execution engine, each of which encapsulates a calling application thread of control
- Message and packet data paths are prioritized

10.1.1 Inputs

Each core component can have multiple inputs, each of which is associated with a different packet/message handler. Each input is associated with a hardware or software queue, and has a globally unique ID. The execution engine registers a packet/message handler for each input ID on behalf of a core component.

The set of IDs for packet handlers and message handlers have the same range. For example, a packet handler can have an ID of 64 and so can a message handler. The ID's are allocated at compile time and are assigned using the following convention:

Table 10-1. Core component ID allocation

ID	Description
0-63	Reserved for microblocks on Ingress IXP2400 or IXP2800 (including `IX_NULL`, `IX_DROP` and `IX_EXCEPTION`). Each microblock is assumed to have one unique input ID.
64-255	ID's for core components on the Ingress IXP2400 or IXP2800.
256-319	Reserved for microblocks on Egress IXP2400 or IXP2800 (including `IX_NULL`, `IX_DROP` and `IX_EXCEPTION` when bit 8 is stripped out). Each microblock is assumed to have one unique input ID.
320-511	ID's for core components on the Egress IXP2400 or IXP2800

Bits 9-16 of the input ID are reserved for blade numbers, which are not currently supported.

10.1.2 Outputs

Outputs are logical outputs that determine the flow of data through the system. The outputs typically represent the result of some packet classification in the core component. For example, an IP Filter Core Component may have two outputs: `IP_FILTER_PASS_OUTPUT` and `IP_FILTER_DENY_OUTPUT`.

Note: Unlike the IXA-SDK 2.0 implementation, there are no active elements in the IXA Portability Framework corresponding to targets. Core components could conceivably be configured without defining outputs; a packet handler could use the input ID of another core component directly in order to send it a packet. However, the use of output labels is a recommended design methodology to decouple system data-flow design from the design of the individual core components.

10.1.3 Binding core components

The notion of outputs and inputs allows the developer to decouple the data flow from the actual development of the components. Binding simply consists of mapping outputs to inputs. This is done at compile time; there are no binding functions.

10.1.4 Execution Engine

An execution engine is the thread of execution that runs one or more core components. This is a kernel thread for Linux and a task for VxWorks. In the future, we may support user processes for Linux. The developer of the system application controls how many execution engines there are and which core components run in each execution engine. The system requires the developer to write an initialization and termination function for each execution engine. The system provides an API function that spawns a thread/task/process to run the execution engine, invoking the user-defined initialization function.

10.1.5 Scheduling Policy

This allows the developer of the system application to control how the different queues in an execution engine are scheduled. Each execution engine runs one or more core components and each core component has one or more inputs (which map 1:1 with a queue). The scheduling policy is specific to an execution engine. If no policy is specified, then Round Robin is the default.

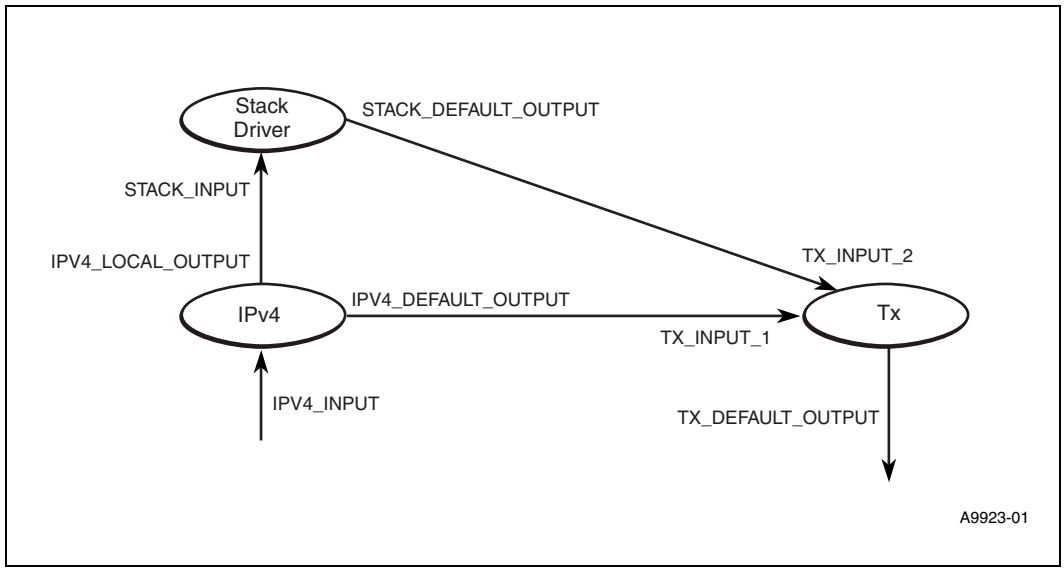
10.1.6 Core Component Configuration Example

Figure 10-1 shows the three core components listed in Table 10-2.

Table 10-2. Core Component Infrastructure Example

Core Component	Description
IPv4 Forwarder	This has two outputs `IPV4_DEFAULT_OUTPUT` connected to the TX component and `IPV4_LOCAL_OUTPUT` connected to the stack driver. It has one input which receives packets from the microblock.
Stack driver	This has one output `STACK_DEFAULT_OUTPUT` and one input which receives packets from the IPv4 forwarder block
TX	This has one output `TX_DEFAULT_OUTPUT` and 2 inputs.

Figure 10-1. Core Component Datapath Example



The bindings may be described as follows in a file bindings.h included at compile time by all the components.

```
/* Core Component Inputs: */
#define IPV4_INPUT          100
#define STACK_INPUT         101
#define TX_INPUT_1          102
#define TX_INPUT_2          103

/* Microblock Inputs: */
#define TX_MICROBLOCK_INPUT        10

/* Core Component Outputs and Bindings: */
#define IPV4_DEFAULT_OUTPUT     TX_INPUT_1
#define IPV4_LOCAL_OUTPUT       STACK_INPUT
#define STACK_DEFAULT_OUTPUT    TX_INPUT_2
#define TX_DEFAULT_OUTPUT       TX_MICROBLOCK_INPUT
```

Figure 10-2. Assigning Core Components to Execution Engines

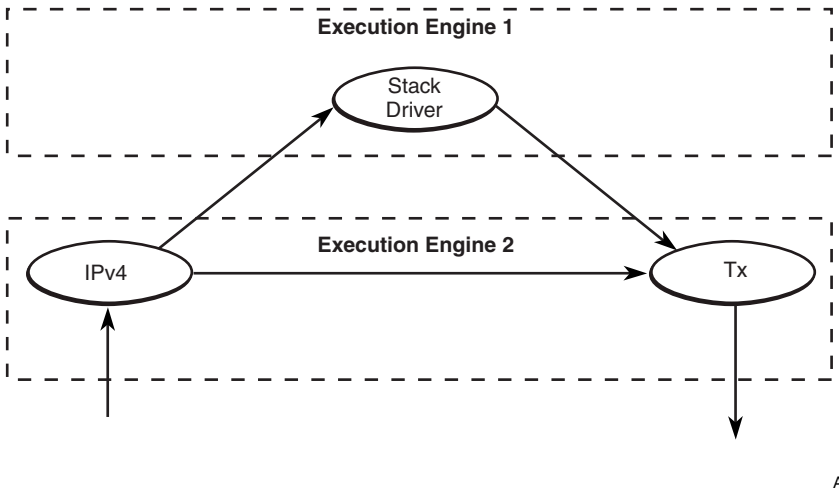


Figure 10-2 shows how the application developer has chosen to run the core components in two execution engines each corresponding to one thread of execution.

In addition, the developer can associate a scheduling policy with each execution engine. For example, the Execution Engine 2 manages three queues associated with the inputs `TX_INPUT_1`, `TX_INPUT_2` and `IPV4_INPUT`. The developer can associate a scheduling policy with these queues such that different inputs receive different priorities; for example, the IPv4 input could receive the highest priority.

10.2 Core Component Infrastructure Design Decomposition

The Core Component Infrastructure provides support for:

- Handling messages and packets in separate threads from the thread in which the message or packet was sent
- Selecting between message/packet paths in the case that user wants to process more than one message/packet path in the same thread
- A mechanism to allow users to drive the message/packet-processing functions using their own thread of control

The core-component infrastructure includes the following constructs:

- A Core Component used for processing messages and packets in core space and for configuring and patching code for microblocks
- An Execution Engine which encapsulates a thread of control to execute any number of message/packet processing functions attached to one or more core components
- A Scheduling Policy which is used to schedule the processing of messages and packets if more than one message/packet-processing function is controlled by an Execution Engine

10.2.1 Design Purpose

The Core Components Infrastructure provides the following advantages:

- Separation of system development (controlling how core components work together) from the development of individual core components.
- Ability for the system designer to initiate execution externally from the core component (a core component does not include an internal thread-of-control loop for message processing).
- Ability for the system designer to design and modify the execution-control behavior of a core component or group of core components working within a single thread or task.
- Ability for the core component designer to invoke separate message/packet handlers in response to data arriving from different message/packet sources or to use separate input IDs with the same message/packet handler so that scheduling can be fine-tuned.
- There does not have to be a one-to-one correspondence between microblocks and core components. An individual microblock may not communicate with any entity in core code, and a number of microblocks may be so closely related that they could be configured from just one core component.
- Minimal infrastructure overhead.
- Users that have ready-defined and tested data structures to support message and packet processing can still use those structures with core components without having to modify those structures to include core components.

10.2.2 Design Constraints

The Core Component Infrastructure was designed for running in a kernel with a monolithic address space; it was conceived with the understanding that it is not required to work in an environment where each thread of control operates in its own address space. Whereas the design does not preclude the possibility of porting the framework to an operating system environment with partitioned address spaces, this would require some framework constructs be created in shared memory space and for some redesign of the message/packet notification mechanism. The design, however, does allow for the possibility of communication between the kernel-based infrastructure and a control application running in a separate (user) address space.

10.2.3 Core Component Infrastructure Constructs

10.2.3.1 Core Component

A Core Component manages data structures used in packet/message processing. A core component can:

- Configure associated microblocks
- Receive exception packets from microblocks
- Send packets to other core components and microblocks

However, there is no requirement for a core component to be associated with any microblocks.

Internally, a core component is implemented as a structure supported by a library of Infrastructure API functions. Externally, a core component is a user-defined library exporting a set of user-defined functions. The internal structure is hidden from the user; it may only be accessed via a core-component handle using the infrastructure API functions.

Each core component will contain or reference the following:

- A pointer to an initialization function, which will allow the user to perform initialization on his/her own private structures and to configure and download microcode
- A pointer to a termination function
- A table of packet-input IDs (which is initially empty)
- A table of message-input IDs (which is also initially empty)

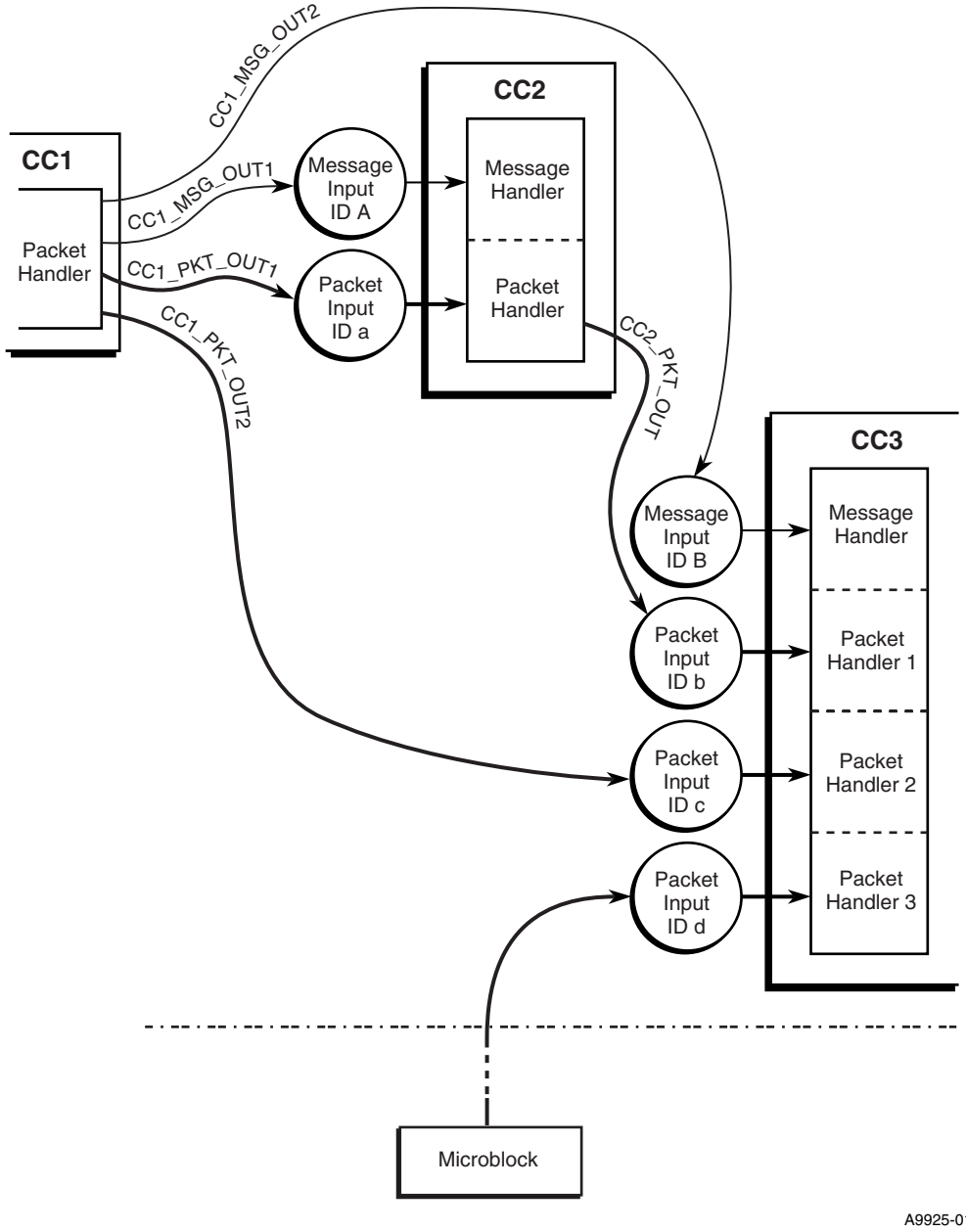
The infrastructure will include a number of APIs to support the creation and setup of core components:

Table 10-3. Core Component Infrastructure APIs

API	Description
`ix_cci_cc_create()`	Allows the user to set the core component's initialization and termination functions
`ix_cci_cc_add_packet_handler()`	Links a packet-input ID with a packet-handling function and data context
`ix_cci_cc_add_message_handler()`	Links a message-input ID with a message-handling function and data context
`ix_cci_cc_remove_packet_handler()`	Unlinks a packet-input ID with a packet-handling function and data context
`ix_cci_cc_remove_message_handler()`	unlinks a message-input ID with a message-handling function and data context
`ix_cci_cc_add_event_handler()`	Links a timed event with an event-handling function and data context
`ix_cci_cc_destroy()`	Unlinks the packet handlers and message handlers from the core component inputs and frees any memory resources dedicated to the core component

Figure 10-3 shows how core components are interconnected to form message and packet data paths.

Figure 10-3. Example of Packet and Message Data Paths between Core Components

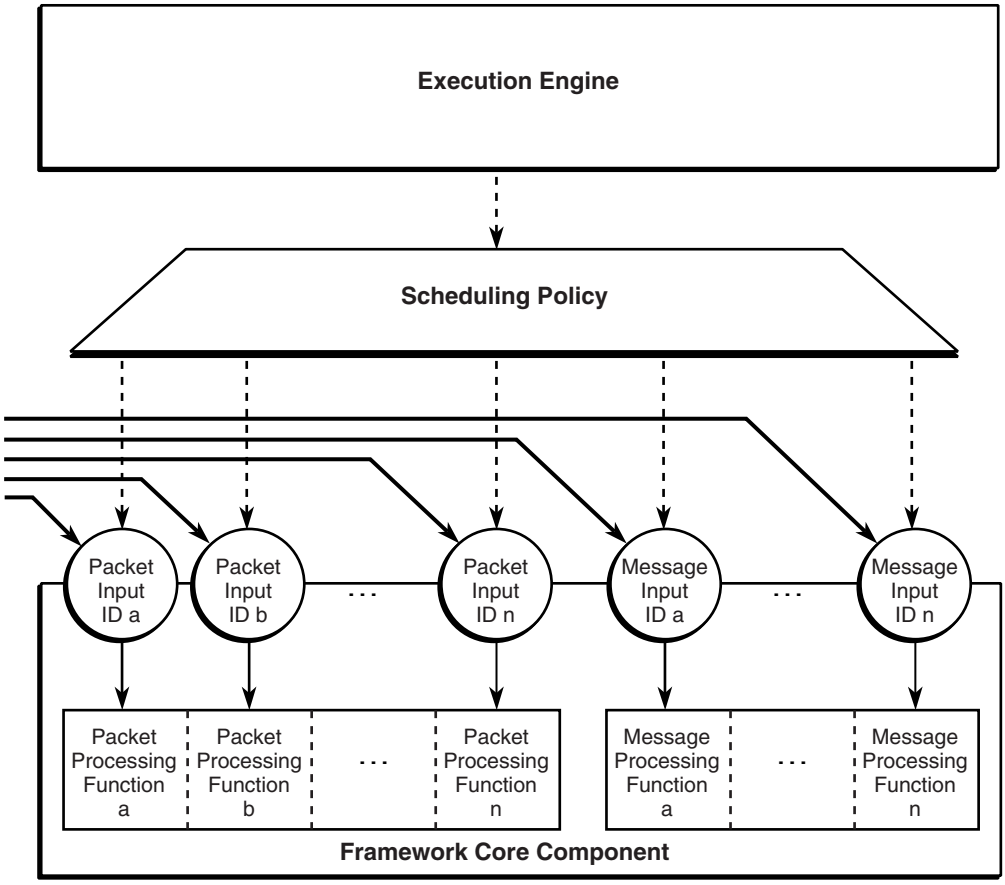


A core component is run by an execution engine. An execution engine includes a function pointer so that the thread of control for a core component may be provided by the user's application code. If the core component contains more than one message-handling or packet-handling function, to be run within one execution engine, a scheduling policy (SP) selects which processing function will be run in the case when data is present on more than one input. Figure 10-4 shows an end-user's view of a core component connected to an execution engine via a Scheduling Policy. Data input

paths are shown as solid lines; execution control paths are shown as dotted lines. For further details about execution engines, refer to Section 10.2.3.2, "Execution Engine." Scheduling policies are described in Section 10.2.3.3, "Scheduling Policy."

Note: An execution engine includes a default policy tree that schedules message and packet processing using a round-robin mechanism. The policy tree will give messages strict priority over packets.

Figure 10-4. Core Component Infrastructure Constructs



When the API `ix_cci_send_message()` or `ix_cci_send_packet()` function sends data to an input ID, the message/packet data is stored in the queue associated with that ID. Later, the handler registered in the associated queue structure is executed in the context of the execution engine thread. This may or may not be in a different thread of execution from that of the sender, depending on the execution engine in which the sender is running.

10.2.3.2 Execution Engine

A core component does not have any threads of control. Packet and message processing is done in the context of an active element called an execution engine.

The execution engine will have a shutdown function that will set an internal flag to break the *infinite* loop in the execution function.

For operating systems that use co-operative multitasking, such as VxWorks, the infrastructure also allows execution engines to control how many tokens they will process before switching out to the next task run by another execution engine.

10.2.3.3 Scheduling Policy

If a core component running in a single engine has multiple data-input paths, then if messages/ packets (tokens) are pending on more than one input, the core component or some other separate entity must decide which input must be processed next. The idea of a scheduling policy is to make this decision based on a criterion, such as:

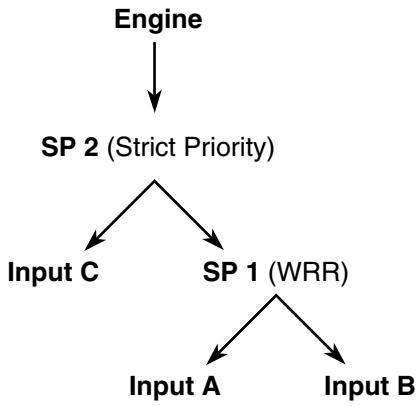
- Strict priority
- Round-robin
- Weighted Round-robin

The infrastructure will provide policies that select based on strict priority, round-robin, and weighted round-robin. By default, the execution engine automatically adds input IDs to the default policy or policy tree whenever message/packet handlers are added to a core component associated with that execution engine. A core component is associated with an execution engine when the core component is created. The default policy will treat all packets with the same round-robin priority and all messages with the same priority but messages will have strict priority over packets.

10.2.3.3.1 Hierarchical Scheduling

Scheduling Policies and core component inputs can be configured into a tree hierarchy to provide a mix of policies; policies are branches in the tree and core component Inputs are leaves, as shown in Figure 10-5.

Figure 10-5. Policy Tree Example



For example, SP 1 could select between Input A and Input B using a weighted round-robin algorithm. SP 2 could select between Input ID C and SP 1 in strict priority, so that Input C takes higher priority. Thus, the weighted round-robin paths would only be selected if there are no packets/messages pending in Input C. The default policy is actually a policy tree consisting of three policies: a round-robin message-selection policy, a round-robin packet-selection policy, and a root policy that selects between the message-selection policy and the packet-selection policy using strict priority.

The user creates a policy tree using `ix_cci_policy_add_branch()` and `ix_cci_policy_add_leaf()`. The policy tree is added to an execution engine using `ix_cci_exe_add_policy()`.

10.2.4 Packet/Message Flow

A Scheduling Policy (SP) can be seen as a core component input with several selection paths.

Because a scheduling policy may control a number of core component inputs, the core component input must pass an ID flag to identify itself when it receives data into an empty queue. The scheduling policy assigns the ID value when the core component input is added to the scheduling policy; it is not the globally unique input ID. The reason for this is because the ID indexes into the list of branches and leaves—children—added to the policy, and into an internal 32-bit pending flag in the scheduling policy that indicates the children with data pending. After updating its internal data, the scheduling policy sends its own ID to its parent—which may be another scheduling policy or the execution engine. If the parent is an scheduling policy, this process repeats until the execution engine at the root of the scheduling tree is reached. At this point, the semaphore in execution engine is unlocked to wake up the execution engine's thread if it is asleep. All the foregoing actions take place in the thread context of the message or packet sender.

The execution engine calls the message-handling functions associated with the inputs that have messages/packets pending. If scheduling policy scheduling flags indicate that more than one core component input has packets/messages queued for processing, the scheduling policy or scheduling policy tree is responsible for scheduling the order in which the packets/messages are processed.

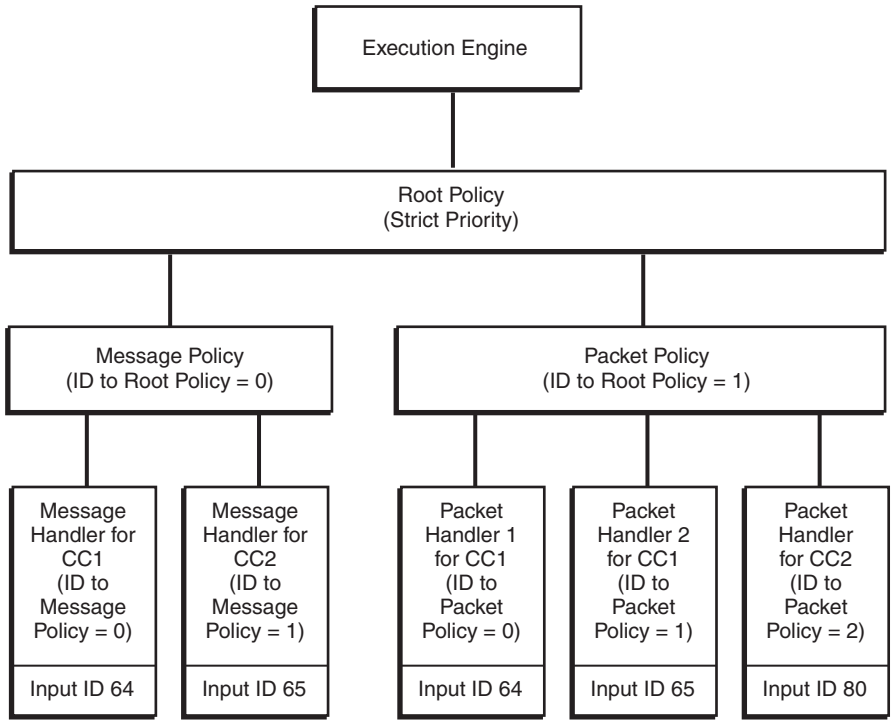
For example, consider the scheduling hierarchy shown in Figure 10-6, and assume a starting condition where all packet and message queues are empty. If a packet is sent to Input ID 65—which corresponds to packet handler 2 of core component 1, then the packet policy updates bit 1 of its pending flag, the root policy updates bit 1 of its pending flag, and the execution engine is woken up. The processing function in the execution engine runs the processing function in the root policy, which selects the packet policy because the message policy bit is clear. The packet policy examines its pending flag. Since only bit 1 is set, the policy runs packet handler 2 of core component 1.

While packet handler 2 is executing, a message arrives on message input ID 64, another packet arrives on packet input ID 65, and a packet arrives on input ID 80. The message causes bit 0 of the pending flag in the message policy and root policy to be set, the packet on ID 65 causes bit 1 of the pending flag in the packet policy and root policy to be set again, and the packet on ID 80 causes the bit 2 in the packet policy to be set—and bit 1 in the root policy.

When the handler for the first packet has terminated, the root policy examines its pending flag again. This time, both bits 0 and 1 are set, and since messages get fixed priority over packets, the message policy is selected. The message policy sees bit 0 of its pending flag is set, and runs the message handler for core component 1 accordingly. If no more messages and packets are received, bit 0 of the message policy and root policy are cleared. The root policy selects the packet policy. The packet policy uses its round-robin mechanism to select the packet handler for core component 2—input ID 80. If no more messages and packets are received while the core component 2 packet

handler is running, bit 2 of the pending flag in the packet policy clears, and the root and packet policy selects packet handler 2 for core component 1 to handle the remaining packet associated with input ID 65.

Figure 10-6. Scheduling Hierarchy Example



10.2.5 Mapping to the IXA SDK 2.0 ACE Framework

This section is provided to allow users familiar with the ACE Framework of IXA SDK 2.0 to map concepts from the ACE environment to the new Core Component Infrastructure.

Packet buffers will be implemented based on the buffer abstraction provided by the Resource Manager APIs. For more information, see Chapter 7, “Resource Manager.”

The Target is no longer a construct, but just an output ID, typically defined in a header file.

The memory will be handled entirely by the Resource Manager's Memory API. For more information, see Chapter 7, “Resource Manager.”

Error handling will be implemented in the OSSL. For more information, see Chapter 11, “Operating System Service Layer (OSSL) Support.”

Semaphores, mutexes, conditions, etc. from the ASL (Action Services Library) in the ACE Framework will be moved into the OSSL. These constructs are OS-dependent and so it is natural to implement them in the only OS-dependant library. This design will allow for easy infrastructure porting. If the OSSL is ported to a new OS, then the rest of the applications should be just recompiled.

Other ASL constructs will be treated as follows:

- Some support constructs like DICTIONARY, HASH and CHECKSUM will be included in the new API only if needed.
- NCL (Network Classification Language) support constructs like ELEMENTs and SETs will be dropped entirely.
- There will be no C++ support in the original API, C++ wrappers could be easily created when needed.
- The TRACE facility that existed in the ASL, but was not implemented, will be dropped entirely.
- The TASK construct will be dropped. IDL support has been replaced with a static message-passing system. The semantic of a former TASK can be achieved by any entity that makes calls to the provided API send functions.
- Most of the functionality of INDEX and ARENA is provided by the Resource Manager's memory-cell abstraction. Sequential read/write from/to memory cells will be provided. For SDS (Serialized Data Stream) support, type-dependent read/write functions will be provided.
- The TIME facility will be moved into the OSSL, due to the OS dependency.
- NETIF support will be dropped.
- The accessor/modifier functions for the former ASL packet buffer will not be implemented because, as a library, we have no knowledge about the structure of packet metadata (because this will now be application-dependent). It will be the responsibility of application developers to provide equivalent functionality based on their own needs.
- Classification/Action support will be implemented in the form of a handler function that will be invoked for each core-component input receiving a packet. It is the programmer's decision to decide whether classification and action implementation will be merged or not. The same mechanism will apply to message processing. Internally, there is no difference between packets and messages but we separate them to increase performance. The relative performance can be tuned by setting the priority of packet processing versus message processing using SPs. By default, round-robin scheduling will be used between messages and between packets, but messages will get strict priority over packets.
- The EVENT construct in the ASL will be implemented by the new API as a function associated with an execution engine.

10.3 External Data Structures

The constructs in the core-component infrastructure are exposed as handles only. Handles have been chosen over pointers for the following reasons:

- Reduces the number of arguments required by API functions by encoding the locations of important *parents* in the handle—for example, the handle for a core component includes information about the location of its controlling policy and its execution engine.
- Allows the framework to include stronger validity checking—a pointer can only be checked for `NULL`.
- Hides private infrastructure data from the user, reducing the chances of misuse leading to unexpected program behavior.

10.4 External APIs

Table 10-4 gives a summary of the APIs exposed by the core-component infrastructure. For complete descriptions and definitions of the API functions, see the *Intel® Internet Exchange Architecture Portability Framework Reference Manual*.

Table 10-4. Core Component Infrastructure API

Name	Description
`ix_cci_cc_add_event_handler()`	Adds an event handler using the handle of the component.
`ix_cci_cc_add_message_handler()`	Adds a message handler to a core component and associates it with an input ID.
`ix_cci_cc_add_packet_handler()`	Adds a packet handler to a core component and associates it with an input ID.
`ix_cci_cc_create()`	Creates a core component and returns a component handle.
`ix_cci_cc_destroy()`	Destroys a core component specified by a handle to the component.
`ix_cci_cc_remove_event_handler()`	Removes an event created using `ix_cci_cc_add_event_handler()`.
`ix_cci_cc_remove_message_handler()`	Deletes a message handler.
`ix_cci_cc_remove_packet_handler()`	Deletes a packet handler.
`ix_cci_exe_add_policy()`	Adds a policy or policy tree to an execution engine. A credit quantum may be specified for a weighted round robin policy or a priority for a strict priority policy.
`ix_cci_exe_get_info()`	Returns the execution-engine handle, engine number, and a context pointer associated with the execution engine in which the caller is running.
`ix_cci_exe_run()`	Runs the execution engine in a spawned task, thread, or process.
`ix_cci_exe_set_default()`	Sets an engine as the default engine whose information is returned by `ix_cci_exe_get_info()` when that function is called from a non-engine thread.
`ix_cci_exe_shutdown()`	Shuts down the execution engine identified by a handle, terminating its task, thread, or process.
`ix_cci_init()`	Initializes the framework for use by the core-component infrastructure.
`ix_cci_fini()`	Terminates the framework.
`ix_cci_policy_add_branch()`	Adds a branch—another policy or policy tree—to a scheduling policy, supporting construction of a hierarchical scheduling policy. A credit quantum may be specified for *a* weighted round robin policy or a priority for a strict priority policy.
`ix_cci_policy_add_leaf()`	Adds a leaf node or input ID to a scheduling policy or execution engine. A credit quantum may be specified for a weighted round robin policy or a priority for a strict priority policy.
`ix_cci_policy_create()`	Allocates a scheduling policy specifying the type—type is one of *round robin*, *weighted round robin*, or *priority.* Returns a handle to the policy.

Table 10-4. Core Component Infrastructure API (Continued)

Name	Description
`ix_cci_policy_destroy()`	Frees the scheduling policy. Deallocates all resources associated with a tree.
`ix_cci_register_fatal_error_handler()`	Allows a control application to register a fatal-error handler.
`ix_cci_send_message()`	Sends a message to a specific input of a core component.
`ix_cci_send_packet()`	Sends a packet to a specific input of a core component or to a microblock identified by microblock ID.

Operating System Service Layer (OSSL) Support 11

The IXA SDK Tools CD provides additional library support for the Operating System Services Layer (OSSL). Applications based on the IXA SDK use the OSSL library's operating system independent APIs and data types for system services such as threads, semaphores, and memory management, etc.

For details concerning the use of these libraries and the specific APIs they support, see the *Intel® Internet Exchange Architecture (IXA) Software Reference Manual* on the IXA SDK Tools CD.

Intel XScale® Core Support 12

The IXA SDK Tools CD provides additional library support for Intel XScale® core applications. This includes Intel XScale® core support for:

- Microengine Loader
- Hardware Abstraction Layer (HAL)
- Tools

12.1 Microengine Loader for the Intel XScale® Core

The Microcode Loader is used to load microcode images, created by the microcode linker `uc ld`, to the appropriate microengines. Intel® IXP2400 and IXP2800 Network Processor applications consist of two distinct parts: one consisting of microengine images and another of ARM-compiled code linked against the Microcode Loader Library, `uclo.a`. This section describes the Microcode Loader.

For details concerning the use of these libraries and the specific APIs they support, see the *Intel® Internet Exchange Architecture (IXA) Software Reference Manual* located on the IXA SDK Tools CD.

12.2 Hardware Abstraction Layer for the Intel XScale® Core

The Intel XScale® core and the Microengines share access to the same Intel® IXP2400 and IXP2800 Network Processor functional units—SRAM, SDRAM and the FBI (IX Bus). An Intel XScale® core application interfaces to these units in one of three ways:

- Through memory-mapped locations in the Intel XScale® core address space when running on the hardware
- Using the AMBAIO and XACT API calls to the Transactor when running as a simulation
- Using the APIs provided by the Hardware Abstraction Layer (HAL)

The HAL generates code to interface either to the Intel XScale® core-based hardware or the Transactor. A Intel XScale® core application that uses HAL can run in hardware mode or simulation mode without changes to the code that accesses the functional units. This increases portability of the application code when moving between the simulation environment of the Transactor and the Intel XScale® core-based hardware.

For details concerning the use of these libraries and the specific APIs they support, see the *Intel® Internet Exchange Architecture (IXA) Software Reference Manual* on the IXA SDK Tools CD.

12.3 Tools

Additional APIs provide support for debugging and remote debugging of microengine code from the Intel XScale® core.

For details concerning the use of these libraries and the specific APIs they support, see the *Intel® Internet Exchange Architecture (IXA) Software Reference Manual* on the IXA SDK Tools CD.

Control Plane PDK 13

13.1 Overview

Control Plane Platform Development Kit (CP PDK), implements the Network Processor Forum (NPF) APIs. The architecture has been defined to accommodate and take advantage of the Intel® Internet Exchange Architecture Software Development Kit (Intel® IXA SDK) software.

The Network Processing Forum (NPF) APIs are well designed for this purpose as they present a flexible and well-known programming interface for the implementation of control plane applications. The hardware properties and nature of the interface between the control and the forwarding planes are encapsulated in the API implementation. This includes transparency with respect to the existence of multiple forwarding planes as well as vendor-specific implementation details. Protocol stacks and network processors available from different vendors can be integrated easily with the NPF APIs. The APIs included in the CP PDK are based on the NPF APIs. For more information about NPF, refer to http://www.npforum.org/.

For complete information about these APIs, refer to the CP PDK documentation supplied with the Intel® IXA SDK.

Framework Memory and CPU Usage Summary A

The following metrics are relevant to the Intel® Internet Exchange Architecture Software Development Kit (Intel® IXA SDK) 3.5 release of the Framework.

A.1 Code and Image Sizes

The tables listed below provide details on the code and image sizes of various components of the Framework software.

Table A-1. Code Size for Framework Infrastructure

	Lines of Code	Lines of Comments	Total Raw Lines	Comment %
Resource Manager	24158	21233	55104	46.8
Core Component Infrastructure	4209	4175	9420	49.8
OS Services Library	4324	4236	9611	49.5

Table A-2. Compiled Code Size for VxWorks* Image

	IXP2400	IXP2800
Resource Manager	194 Kb	196 Kb
Core Component Infrastructure	40 Kb	40 Kb
OS Services Library	9 Kb	9 Kb

Table A-3. Compiled Code Size for Linux* Kernel Image

	IXP2400	IXP2800
Resource Manager	138.1 Kb	139.5 Kb
Core Component Infrastructure	26.9 Kb	27.4 Kb
OS Services Library	8.2 Kb	8.2 Kb

A.2 Memory Consumption

A.2.1 Resource Manager

The Resource Manager (RM) uses both operating system (OS) memory and non-OS memory.

For non-OS memory (the common Intel XScale® core/Microengine memory), the RM uses 1.3 Kb on each SRAM channel for QArray descriptors. Further, it uses SCRATCH memory for two internal scratch rings, the size of which is configurable by the programmer. By default, the scratch usage is about 1 Kb.

If the PCI communication facilities are to be used, then the RM allocates a hardware free list and a software free list of 1024 elements each for this purpose. The data part for the buffers is 2048 bytes, and the metadata part would be 32 bytes for hardware buffers and 16 bytes for software buffers. In total, this amounts to 2Mb + 16 Kb of DRAM and 32Kb of SRAM. In addition, the RM uses another software free list of 2048 element for configuration properties. This free list has a data area of 64 bytes and metadata area of 64 bytes, amounting to another 256 Kb of DRAM.

For OS memory, the RM will occupy the code size itself (see Section A.1) plus its static data segments (about 100 Kb) and OS dynamic memory (about 700 Kb). Altogether RM will use about 800Kb OS memory. Of course, extra OS memory will be used based on the requests of the particular application.

A.2.2 Core Component Infrastructure

The Core Component Infrastructure uses about 2Kb of OS memory and a very small amount (less than 1 Kb) of non-OS DRAM.

A.2.3 OS Services Library

The OS Services Library does not use any memory for itself.

A.2.4 Memory Usage Summary

Table A-4. Summary of Framework Memory Usage

	DRAM	SRAM	SCRATCH	OS
Resource Manager	~2.3 Mb	~40 Kb	~1 Kb	~800 Kb
Core Component Infrastructure	~1 Kb	0	0	~2 Kb
OS Services Library	0	0	0	0

A.3 CPU Usage

The load on the CPU is based solely on the Intel XScale® core application performance and the amount of data sent by the microengines to the core. However, below are some general guidelines for the framework software.

A.3.1 Resource Manager

The Resource Manager (RM) has five active internal threads, each waiting on semaphores to start processing. Thus, in an idle state, the RM will not consume any CPU cycles. This is due to the fact that all these threads (except the 64-bit counter thread) start processing only when they are awakened by interrupts.

The internal threads of the RM are as follows:

- RM packet dispatch thread
- RM message dispatch thread
- RM 64-bit counter thread
- RM PCI communication receive thread
- RM PCI communication transmit thread

If the microengines do not send and packets to the core, then the packet dispatch thread will not consume any cycles. Alternately, if they do send packets to the core, then the CPU usage depends on the number of packets and the amount of processing per packet. The same applies for the message dispatch thread.

If there are no 64-bit counters created, then the 64-bit counter thread will not consume any CPU cycles. If these counters are instantiated, then the CPU usage depends on the number of counters and their minimum update timeout value.

The PCI communication threads will not consume any CPU cycles when there is no communication between Ingress and Egress. Of course, if they do send messages to each other via PCI, then the CPU usage depends on the number of messages and the amount of processing per packet.

In summary, the Resource Manager does not use any CPU cycles without external input.

A.3.2 Core Component Infrastructure

The Core Component Infrastructure will use a small amount of CPU time, even in the case when there nothing to process.

In the CCI, the amount of processing depends on how many execution engines are instantiated. Even when there are no packets, messages, or events to process, the execution engines use some CPU cycles as it gets awakened from time to time.

A.3.3 OS Services Library

The OS Services Library (OSSL) does not use any CPU cycles if the application does not make calls to the exported functions.

Like the Resource Manager, the OSSL does not use any CPU cycles without external input.

Glossary B

A

AAL	ATM Adaptation Layer. The ATM standards layer that allows multiple applications to have data converted to and from an ATM cell. A protocol used that translates higher layer services into the size and format of an ATM cell.
AAL5	ATM Adaptation Layer 5. AAL functionality to support variable bit rate, delay-tolerant connection-oriented data traffic.
active computing element (ACE)	A logical entity that represents a specific packet-processing activity in the IXA SDK 2.0. IXP1200 applications use ACEs to process packets. The ACE Programming Framework in the IXA SDK 2.0 is now replaced by microblocks and core components in the IXA SDK 3.0
application programming interface (API)	A set of routines, classes, methods, structures, and/or functions used to write applications.
Asynchronous Transfer Mode (ATM)	A transfer mode in which information is organized into cells. It is asynchronous in the sense that the recurrence of cells containing information from an individual user is not necessarily periodic.

B

big endian	A compiler term specifying that, for multibyte values, the most significant byte is first. See also *little endian, network byte order.*
byte order	The way a system stores numeric data, with the most or least significant byte first. Most significant byte first, or *big endian* byte order, is also known as *network byte order.* See also *endianness*.

C

constant bit rate (CBR)	An ATM service class.
committed burst size (CBS)	An IP QoS traffic contract parameter/metric.
committed information rate (CIR)	An IP QoS traffic contract parameter/metric.
cell loss priority (CLP)	An ATM QoS traffic contract parameter/metric.
content addressable memory (CAM)	A hardware feature where a content match is performed to get an index to associated information.

context pipeline	A software pipeline in which different functions are performed on different microengines as time progresses and the packet context is passed between the functions or microengines. Each microengine constitutes a context pipe-stage and cascading two or more context pipe-stages constitutes a context pipeline. The context pipeline get its name from the fact that it is the context that moves through the pipeline.
control plane	The abstraction for a functional area of an application that controls and configures the data plane and handles exception packets, distinguished from the *data processing plane*. Control plane activities are typically performed by *code modules* within the *IXA application*. Compare to *management plane*, whose activities are usually outside the IXA application, in a *host* application.
core component	A packet-processing entity that configures its microblock, initializes and maintains common data structures that may be updated by other applications, and provides exception handlers and control message handlers to process packets/messages sent by the microblock.
Core Component Infrastructure	Infrastructure of a number of APIs to support the creation and setup of core components.
cyclic redundancy check (CRC)	A mathematically computed numerical value transmitted with packet data to ensure the integrity of packet data transmitted between endpoints.
critical section	A section of code in which only one microengine thread has exclusive modification privileges for a global resource (such as a memory location) at any one time. The IXP2400 uses inter-thread signaling to implement critical sections across microengines.
CSR	Acronym for *control status register.*

D

decap	Decapsulation - Removing one or more protocol header from a packet.
DiffServ	Differentiated Service. A means of classifying IP packets into “classes,” based on the DiffServ Code Point (DSCP) in the packet’s IP header.
dispatch loop	A dispatch loop combines microblocks running on a microengine thread and implements the data flow between them.
deficit round robin (DRR)	A QoS queue-scheduling algorithm.
DiffServ Code Point.(DSCP)	A 6-bit field in the IPv4 header.

E

EE	Acronym for execution engine.
encap	Encapsulation - Adding one ore more protocol headers to a packet.

endian, endianness	A compiler term for the *byte order* of multibyte values. See *big endian* and *little endian*.
Ethernet	A local area network (LAN) technology designed for interconnecting networking nodes over a shared medium, as specified in standard IEEE 802.3. Also typically used to refer to the Layer 2 networking protocol as specified in standard IEEE 802.2.

F

fast path	Data path in which the packet is completely processed on the MEv2 microengines without any intervention from the Intel XScale® core.
folding	A software technique used by threads running on the same ME, to optimize read/modify/writes in a critical section. The technique uses the CAM and strict thread ordering enforced via inter-thread signaling to fold the read/modify/write into a single read, multiple modifies, and one or more writes depending on the cache-eviction policy.
Functional Pipeline	A software pipeline in which the context remains with an microengine while different functions are performed on the packet as time progresses. The microengine execution time is divided into “n” pipe-stages and each pipe-stage performs a different functions. The Functional Pipeline get it's name from the fact that it is the function that moves through the pipeline.

G

Guaranteed Frame Rate (GFR)	An ATM service class.

H

head of line blocking	A situation where the transmit operation on a group of ports is blocked by a single port at the head of the transmit queue. This scenario typically occurs when the port at the head of the transmit queue is blocked because of flow control issues and the remaining ports on the queue have data pending but need to wait for this port to finish the transmit operation.
Header Error Check (HEC)	An 8-bit field within an ATM header that is generated by a sender, and checked by a receiver, to determine the validity of an ATM header.

I

Intel® Internet Exchange Architecture (IXA)	A new approach to designing networking and telecommunications equipment based on reprogrammable silicon and open interfaces. Manufacturers of networking and communications equipment can use components from the IX-based product portfolio for designing new, more intelligent network systems.

intrinsic	A C function-like interface that implements a chip-specific hardware feature, not otherwise supported by the C language. Direct use of intrinsics results in non-portable code.
IP	An acronym for *internet protocol*, a standard network protocol. See also *TCP/IP*.
IPv4	Internet Protocol Version 4.
IPv6	Internet Protocol Version 6.
IXP	Acronym for *Intel® Internet Exchange Processor*, and a current instance of this processor.
IXP2400, IXP 2800	Internet eXchange network processors. The IXP2400 has 8 microengines targeted at OC-48 POS line rates and the IXP2800 has 16 microengine targeted at OC-192 POS line rates.

L

L2	Layer 2.
L3	Layer 3.
LLCSNAP	Logical Link Control/Sub Network Access Protocol - Data link layer packet encapsulation headers that identify a protocol, as well as client and control information. Refers to IEEE standards 802.3 with 802.2.
longest prefix match (LPM)	Algorithm IP routers apply to an IP packet destination address to determine the packet's egress port, and hence forward the packet out the egress port.
little endian	A compiler term specifying that, for multibyte values, the least significant byte is first. See also *big endian, endian*.
longword	A 32-bit word; 4 bytes long.

M

medium access control (MAC)	A protocol layer responsible for providing access to a shared communications medium. Also stands for medium access controller - The device used to interface with the physical layer medium.
ME	See *microengine*.
MEv2	A microengine specific to the IXP2xxx network processor family.
microblock	A discrete unit of IXP2xxx code written in microcode or MicroC that is written to the guidelines specified in the IXA Software Framework. Microblocks conform to one of three different types: source, transform or sink. Typically, a microblock has an IIntel XScale® core component that is used to configure and manage the microblock.

microblock group	One or more microblocks that have been combined into a thread executable on a microengine. Typically all threads on the microengine will execute the same microblock group, but it is not required. A typical use instantiates the same microblock group on several microengines.
Microengine (ME)	One of many programmable, specialized processors (8 for IXP2400, 16 for IXP2800).
Mixed Pipeline	A software pipeline where some microengines run a single function (context pipe-stage) and others run multiple functions (Functional Pipeline).
MPKT	M Packet - An IXP2xxx media bus interface data-transfer unit that can be configured to be 64, 128, or 256 bytes in length.
MEv2	Microengine version 2, which is the microengine used for the IXP2400 and IXP2800 network processors.
microcode	Hardware-specific machine code. A *code module* written in microcode can run only on the processor it is written for.
mutual exclusion, mutex	Mutual exclusion is used to guard the critical sections accessed by threads.

N

nrtVBR	Non-Real Time Variable Bit Rate - An ATM service class.
network byte order	The system of storing numeric data with the most significant byte first. See also *big endian, endianness*.
network services application	General descriptive term for the kind of application built with the *IXA SDK*.

O

Operations Administration and Maintenance (OAM)	A group of network management functions that provide network fault indication, performance information, and data and diagnosis functions within an ATM network. Also the type of ATM cell payload used to carry such information.
OC-12, OC-48c	Optical Carrier (SONET) - Level (e.g., Level = 3, 12, 48, 192). Often used to specify data rates; the base level rate is 51.84 Mbps (OC-1); each level thereafter operates at a multiple of the base level rate (thus, OC-3 runs at 155.52 Mbps, OC-12 runs at 622.08 Mbps, etc.).
OS	Acronym for *operating system*.
Operating System Services Library (OSSL)	An OS abstraction API used within the IXA SDK to achieve portability.

optimized data plane libraries	A library of low-level macros and functions for microengine program development. The purpose of this library is to provide a layer of portability, so programmers can write code that will run on IXP1200, IXP2400, IXP2800, and future IXP chips. For details, see the *Intel® Internet Exchange Architecture Optimized Data Plane Libraries Reference Manual* on the IXA SDK Tools CD.

P

payload	The part of a packet that carries data, as opposed to those parts that carry information about the packet.

Q

quadword	A 64-bit word; 8 bytes long.
quality of service (QoS)	A networking term that specifies a guaranteed throughput level.

R

Resource Manager	A programming interface between Intel XScale® core applications and the microcode running on the microengines for the IXP2400 and IXP2800 network processors.
Round Robin (RR)	A scheduling algorithm in which entities/queues are services/scheduled in a consistent serial manner.
rtVBR	Real Time Variable Bit Rate. An ATM service class.
Rx	Receive.

S

SAR	Segmentation And Reassembly - The process of transforming frames-to-cells and cells-to-frames.
SDE	Acronym for *software development environment*.
SDK	Acronym for *software development kit*.
semaphore	Semaphores are the primary means for providing thread synchronization.
sink microblock	A function or macro that disposes of a packet, either by enqueuing it within the IXP or sending it to an external interface.
slow path	The execution path of the packets that require exceptional handling. This may be error packets or packets that need to be handled differently than the normal case. In this case, it will take longer to process because they will be handled by a general-purpose processor (Intel XScale® core in our case). See also *fast path*.
software pipeline	The MEv2 employs a software pipeline model in the fast path processing of packets. There are three different types of pipelines—*Context Pipeline*, *Functional Pipeline*, and *Mixed Pipeline*.

source microblock	A function or macro that obtains a packet, either by dequeuing it within the IXP or getting it from an external interface.
SP	Acronym for *scheduling policy.*
stdmac	Acronym for *standard macros*. Assembly macros that are microengine-specific, for instruction simplification.

T

TCP	An acronym for *transmission control protocol*, a standard network protocol in which transmission status can be confirmed. Establishes a point-to-point connection, in contrast to *UDP* which is connectionless. See also *TCP/IP.*
TCP/IP	A standard network protocol, using *TCP* over *IP.* See *TCP* and *IP.*
TM4.1	Traffic Management version 4.1 - An ATM specification for managing/ controlling traffic congestion within an ATM network by the actions of buffering, adjusting transmission rates, and policing VCs.
Type of Service (TOS	Refers to an 8-bit field in the IPv4 header.
Tx	Transmit.
thread	An independent task, which can be processed in parallel with other tasks.
transform microblock	A function or macro that parses, analyzes, classifies, or modifies a packet.

U

unspecified bit rate(UBR)	An ATM service class.
usage parameter control (UPC)	VC traffic contract characteristics, that permit ATM network nodes to monitor, control, and police the traffic within the ATM network.

V

variable bit rate (VBR)	An ATM service class.
virtual connection or virtual channel (VC)	A communications channel between ATM systems nodes that provides for the sequential transport of ATM cells.
virtual connection identifier (VCI)	A 16-bit numerical tag within an ATM cell header that identifies a virtual channel over which the cell is to travel.
virtual path identifier (VPI)	An 8-bit numerical tag within an ATM cell header that indicates the virtual path over which the cell should be routed.

VPN	Virtual Private Network.
virtual port (VPORT)	A field accompanying a MPKT that identifies the port (and possibly line card) to/from which the MPKT payload is sent/received.

W

Wide Area Network) (WAN)	A network that spans a large geographical area relative to a local area network (LAN). A WAN typically experiences greater traffic delays (due to distance between nodes and greater network congestion) and packet loss (due to switches dropping packets).
WRR	Acronym for *weighted round robin*.

X

XScale® core	The ARM architecture core processor in the IXP2400 and IXP2800 network processors.