LSI

Storage. Networking. Accelerated.™

Axxia® AXX2500 Family of Communication Processors

Virtual Pipeline™ Technology and Accelerator Engines Reference Manual

Advance, Version 0.1
September 2012

DB13-000407-00

Revision History

Version and Date	Description of Changes
Version 0.1, September 2012	Initial Release

Corporate Headquarters
Milpitas, CA
800-372-2447

Email
globalsupport@lsi.com

Website
www.lsi.com

Document Number: DB13-000407-00

Table of Contents

Chapter 1: Introduction to the Axxia Virtual Pipeline Technology

This chapter introduces the LSI® Axxia® Virtual Pipeline™ technology. Additional information about specific engines is available in their respective chapters.

Overview

The Axxia architecture lets you use both dedicated accelerator engines and general purpose processors in any sequence to process packets. The processing does not require the CPU for scheduling or packet processing. This approach enables system designers to create multiple, custom packet processing paths to match their application packet and traffic flow processing requirements. This removes the need for special CPU programming to control job queuing and scheduling for each packet processing path.

System designers can use the Axxia architecture to design packet processing paths with a sequence of accelerator engines that can receive a packet processing request, perform processing, and then send the packet to another engine for additional processing. The following figure shows an example Virtual Pipeline instance, which is a sequence of accelerator engines.

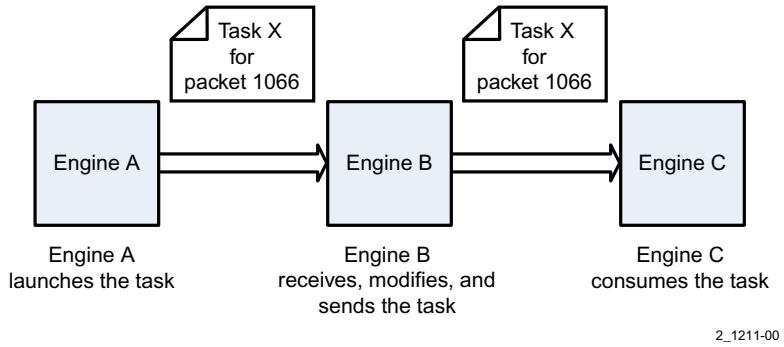

Figure 1 Example Accelerator Engine Sequence (Simple Virtual Pipeline Instance)

A set of engines that define a processing path through the device is called an *engine sequence*. One or more engine sequences can make up a *Simple Virtual Pipeline instance*. You can have multiple independent engine sequences (similar to the previous figure) or dependent engine sequences in a *Branching Virtual Pipeline instance* that accomplish the packet processing.

To create a system, the designer identifies the processing steps required for each traffic flow, and builds sequences of engines to provide the appropriate processing in the required order. The Virtual Pipelines define one or more custom processing engine sequences through the Axxia device.

Engines within an engine sequence, communicate by passing a message or *task* from one engine to the next sequential engine. A task is a data structure that passes information from engine to engine to process a packet. Accelerator engines can create, consume, or modify tasks.

Tasks passed through an engines sequence, contain the following general information.

- Contain a packet, or more often, have pointers to packet data in memory.
- Contain parameters that instruct engines how to process the packet.

Accelerator engines can perform prioritize tasks using these operations.

- Assign a priority (task created by start engines)
- Place tasks into select queues, based on priority
- Adjust task priority based on queue congestion (backpressure)

When a packet arrives for processing, the engine that receives the packet creates a task for that packet. Processing engines use the task to exchange packet processing information. The task contains processing parameters passed from engine to engine in the Virtual Pipeline instance. The task parameters affect packet processing for each engine. When an engine completes its processing for a task, it sends the task to the next engine in the Virtual Pipeline instance.

You can use Virtual Pipeline instances to provide the entire engine-to-engine processing path for the packet, and also for coprocessing with the CPUs. When used for coprocessing, Virtual Pipeline instances define simple engine paths to and from the CPUs, allowing the CPUs to offload compute-intensive packet processing functions to the accelerator engines. For example, the main packet processing can be done by the CPUs, using the accelerator engines to perform encryption, decryption, reassembly, or scheduling as steps in the CPU processing.

Accelerator Engines

The Axxia architecture lets you configure dedicated accelerator engines and general purpose processors in any sequence to process packets. The Axxia device includes the following accelerator engines.

- **Ethernet Input and Output Adapters (EIOA), Preclassifier and Ethernet Switch (PCX)**

  Each EIOA engine can receive packets from and transmits packets to external Ethernet interfaces. You can configure and monitor the Ethernet interfaces, plus the preclassification and switching (PCX) logic. The Axxia device can preclassify and switch traffic between Ethernet ports, which offloads this processing from the intermediate engines.

  There are two EIOA adapters in the AXX2500 device, EIOA0 and EIOA1.

  The EIOA engine serves as a start engine or an end engine for an engine sequence, creating a new task on packet arrival, and receiving a task for transmitting a processed packet.

- **Modular Packet Processor (MPP)**

  The MPP engine classifies packets using highly multithreaded pattern processing engines which provide the required processing and access items such as routing tables and access control lists. You can program the MPP engine using the Functional Programming Language (FPL).

  Based on the classification, the MPP engine (using the FPL) can perform the following additional packet processing.

  — Maintaining multiple hash tables for dynamic flow classification.
  — Maintaining statistics and policing traffic flows.
  — Editing packets, including segmenting packets, adding headers or footers, and duplicating, adding, modifying, or deleting packet data.
  — Performing CRC and checksum operations.

  The MPP engine serves as a start engine or end engine for engine sequences.

- **Packet Assembly Block (PAB)**

  The PAB engine performs both traditional reassembly functions and generalized packet data buffering for simple segment-level editing and transmission.

  The PAB engine is command driven, with commands sent to the PAB as tasks from another accelerator engine such as the MPP engine or the CPU to establish and control reassemblies.

  You can use the PAB engine to reassemble packet segments in a sequence that is contiguous, has gaps, or has overlaps, with segments sent later overwriting all or part of previous segments. Gaps can be filled with pad data. For example, you can change both the packet start and endpoints to delete header or trailer information. The PAB engine can accept byte-level and bit-level offsets.

  The PAB engine is an intermediate engine in engine sequences.

- **Packet Integrity Check (PIC)**

  The PIC engine calculates a cyclic redundancy check (CRC) or checksum value for a packet. The calculated CRC or checksum value can verify an incoming packet, or calculate a new value for a modified packet. Optionally, the PIC engine can write the calculated CRC or checksum value into the packet.

  The PIC engine is an intermediate engine in engine sequences.

- **Stream Editor (SED)**

  The SED engine is a programmable block editor for packets. For many applications, the SED is an efficient way to modify, update, or add header information. It permits you to edit a packet — header, payload, and trailer — in 128-byte blocks.

  The SED engine is an intermediate engine in engine sequences.

- **Modular Traffic Manager (MTM)**

  The MTM engine performs packet buffer management and scheduling. You can program buffer management policies.

  You can configure scheduling structure hierarchy and shaping and scheduling services, with support for weighted, shaped, and strict priority algorithms, with the option to add custom programmed scheduling and custom programmed buffer management. The MTM engine supports up to six levels of scheduling.

  The MTM engine is an intermediate engine in engine sequences.

- **Security Protocol Processing (SPP)**

  The SPP engine accelerates the decryption and encryption of packets for many standard security protocols. It performs packet integrity authentication, anti-replay protection, and programmable security protocol processing that includes packet modifications and error checking.

  The SPP engine is an intermediate engine in engine sequences.

- **Deep Packet Inspection (DPI)**

  The DPI engine performs unanchored searches across one or more packets using standard regular expressions. Based on regular expression search technology, the DPI engine supports simultaneous evaluation of up to one million expressions.

  The DPI engine is an intermediate engine in engine sequences.

- **Timer Manager (TMGR)**

  The TMGR engine provides large-scale (millions) timer services to the CPUs and accelerator engines. The TMGR engine supports timers for the following general purposes.

  — Scheduling timer that expires after a set time.
  — Time-out timer that prompts for an action.
  — Process expiration timer that indicates a process is dead or has timed-out.

  The TMGR engine is an intermediate engine in engine sequences.

- **Memory Management Block (MMB)**

  The MMB engine responds to request from accelerator engines to allocate and de-allocate memory blocks. The MMB engine manages data blocks in memory requested by the other engines. The MMB allocates the data blocks, tracks the number of memory block accesses, and frees the memory when the block contents are no longer needed.

  Because accelerator engines use the MMB engine implicitly, you *do not use* this engine as part of an engine sequence.

- **Network Compute Adapter (NCA)**

  The Network Compute Adapter (NCA) transfers tasks between CPUs (internal or external) and the accelerator engines. The NCA enables the host CPU to configure the all aspects of Virtual Pipeline instances and engine sequences. The NCA can be a start engine or an end engine in an engine sequence.

Engine Sequences and Virtual Pipeline Instances

A set of engines that define a processing path through the device is called an *engine sequence*.

An engine sequence can include three types of engines. Engines that can serve as the first or last engine in an engine sequence are called *start engines* or *end engines*, depending on their location in the sequence. *Intermediate engines* receive a task, optionally modify it, and send it to the next engine in the sequence.

Sequences optionally include one or more intermediate engines that receive, optionally modify and send the task to next engine. Sequences complete with an end engine that receives and consumes the task.

Engines used in an engine sequence have the following characteristics.

- Start engines create and launch new tasks for packet processing.
- Intermediate engines do not generate new tasks and typically do not consume a received task.
- End engines consume the task.
- Multiple instances of the same intermediate engine type can exist.
- Some engines can be used as the start engine *and* the end engine.

The purpose of an engine sequence is to process tasks (and their associated packets). You can create an engine sequence with two or more engines. Engine sequence processing begins with a task launch by the first engine in the sequence, and concludes with the last engine in the sequence consuming that task. Each task is typically associated with a packet, although there can be tasks without packets.

One or more engine sequences can make up a *Virtual Pipeline instance*. There are two types of Virtual Pipeline instances.

- Simple Virtual Pipeline instances that contain only one engine sequence.
- Branching Virtual Pipeline instances having two or more branches (engine sequences), that share the same start engine. The start engine selects an engine sequence and sends a task down the selected engine sequence.

Branching Virtual Pipeline Instances and Flows

In some cases, packets from the same source require a different processing path. For example, packets for a traffic flow might contain both unencrypted and encrypted packets. The engine sequences required for both types are identical, except for the addition of the Security Protocol Processor for the encrypted packet path.

Although you can design separate Simple Virtual Pipeline instances for the two cases, this approach duplicates the common software processing to handle common packet operations.

The Axxia architecture provides a branching pipeline option that enables you to launch both cases using the same function from the same start engine.

A Branching Virtual Pipeline instance defines two or more engine sequences, or branches, that share the same starting engine, can use the same set of start engine output parameters, and are grouped together. The start engine for a branching pipeline must be either the MPP, Expander, or CPU.

A user-defined set of packets that require the same type of processing is called a *flow*.To define a specific set of processing parameters for each flow in a pipeline, you can define a flow table. You can use application software to create the flow table. In the table, you use a programmable flow ID as an index to a set of parameter values that are organized as a flow table entry. The start engine for the pipeline loads the flow table values into the task.

NOTE When the flow table is updated, any tasks already started, continue to use the original information that existed at the time the tasks started.

The Branching Virtual Pipeline instance launches in the start engine the same way that a single pipeline launches. The start engine determines the appropriate branch for processing the packet based on information that is stored in a separate data structure called a flow table, which the engine references using a flow ID.

A user-defined set of packets that require the same type of processing is called a *flow*. Flows are defined based on some field or data in the packet, such as the source IP address. Each entry in a flow table contains parameters and other information needed to process the packets in that flow. Packets with one set of flow IDs are routed down one branch of the Virtual Pipeline instance, while packets with other flow IDs are routed differently.

The following figure shows a branching pipeline with two engine sequences.

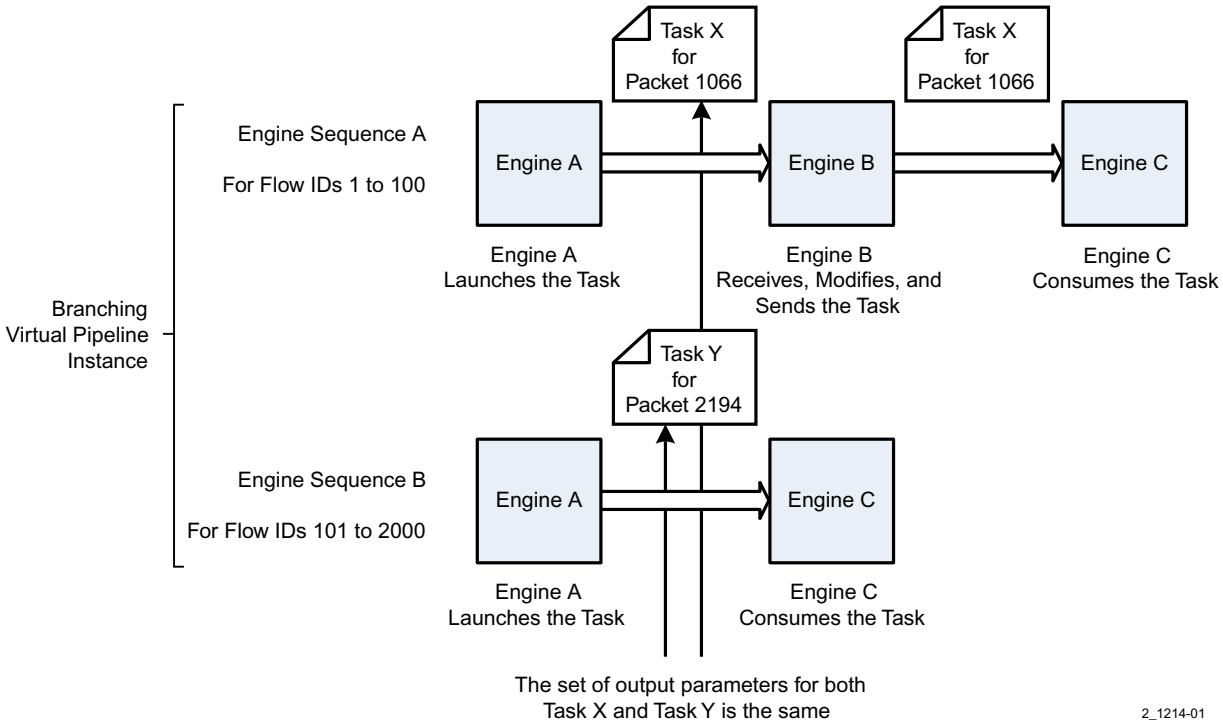

Figure 2 Example Branching Virtual Pipeline Instance

Branching Virtual Pipeline instances are used primarily when the downstream processing in each branch requires a similar set of parameters from the start engine such that the differences between the branches are manageable simply as different entries in the flow table.

APIs for Managing Virtual Pipeline Instances

The run-time environment (RTE) is a set of functions and commands that can initialize, define, and control an Axxia device. The RTE APIs for managing Virtual Pipeline instances at runtime include the following functions.

- ncp_vp_handle_get – Get the Virtual Pipeline instance handle given the type of launching engine (CPU, MPP, EIOA and Expander) and the name of the Virtual Pipeline instance
- ncp_vp_id_get – Get the Virtual Pipeline id from its instance handle
- ncp_vp_num_flows_get – Get the number of flows defined for a Virtual Pipeline instance
- ncp_vp_launch_engine_check – Determine if a Virtual Pipeline instance is launched by the specified launch engine
- ncp_vp_flow_data_func_set – Set encode and decode functions for a given engine sequence identified by vpHdl and engineSeqId. The encode function is used by ncp_vp_flow_data_write() API to encode the flow data structure into hardware format

- ncp_vp_task_param_decode_func_set – Set task param decode function (used by ncp_task_recv() API of task I/O module to decode the task parameters from the received task before sending them up to the application) for a given engine sequence
- ncp_vp_task_param_encode_func_set – Set task param encode function (used by ncp_task_send() and ncp_task_send_segments() APIs) of task I/O module to encode the task parameters) for a given Virtual Pipeline instance
- ncp_vp_engine_seq_id_get – Get the engine sequence id given a Virtual Pipeline instance handle and the Engine Sequence name
- ncp_vp_flow_data_write – Write a flow data entry for the given vpHdl, engineSeqId, and flowId
- ncp_vp_flow_data_read – Read a flow data entry for the given vpHdl, engineSeqId, and flowId

Task Communication

To pass information or data between engines, an Axxia device uses data structures called *tasks*. Tasks provide engines with the following types of information.

- Packet data
- Information on how to process the packet
- The next engine to receive the task

Messaging in Pipelines

Engine sequence processing begins with the launching of a task by the first engine in the sequence, and ends with the last engine in the sequence consuming that task. Each task is typically associated with a packet, although you can have tasks without packets.

The following figure illustrates a simple engine sequence of three engines.

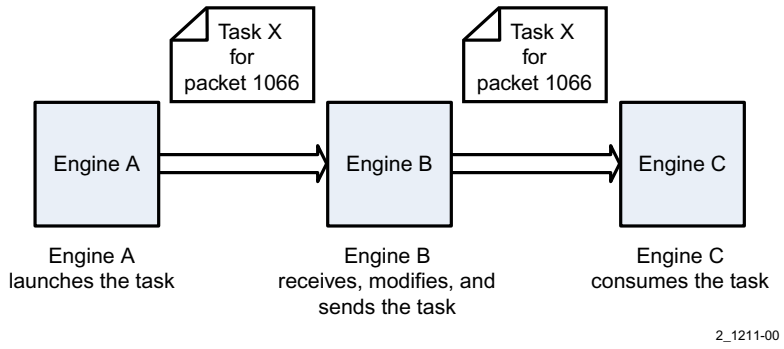

Figure 3 Engine Sequence Task Flow

The engine sequence task flow in the previous figure defines the following process.

1. Engine A, a start engine, creates and launches the new task, task X, to process packet number 1066.
2. Engine B, an intermediate engine, receives the task, reads its processing parameters, and optionally modifies the task and its packet. When Engine B completes processing, it sends the task to Engine C.
3. Engine C, an end engine, receives the task and consumes it.

When launching a task, some start engines can make decisions about task processing that include the following options.

- Launch a new task that continues processing the packet down another pipeline.
- Discard the packet and task.
- Perform any of the following programmed functions.
  - Send out a different packet
  - Send out multiple packets
  - Generate an error.

For example, to send the packet forward in another pipeline, the start engine creates and launches a new task for additional processing. The following figure shows the process.

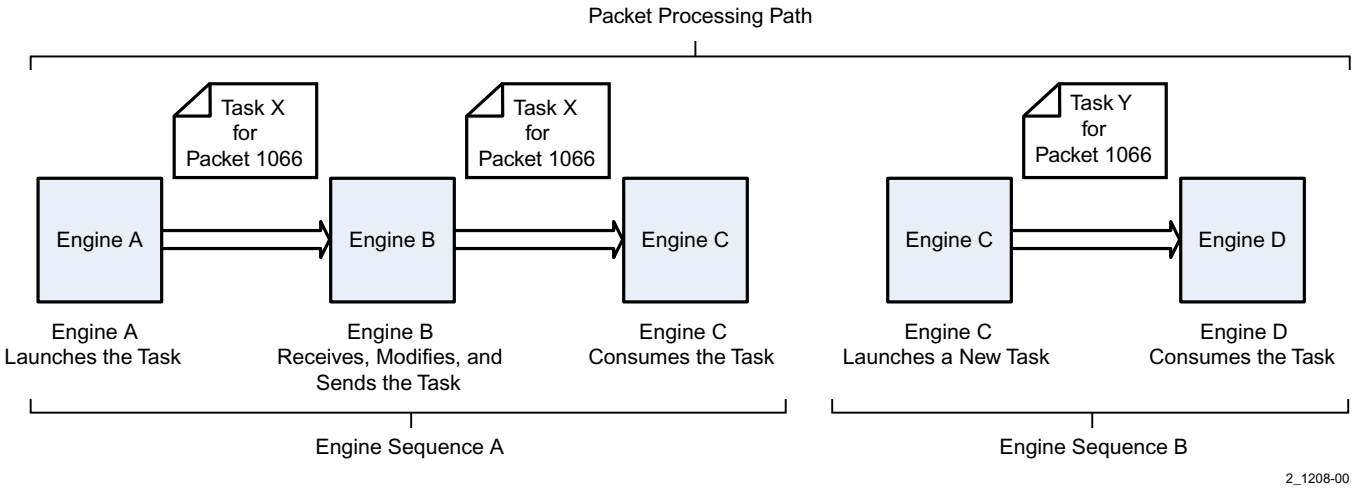

Figure 4 Two Engine Sequences in a Processing Path

Passing Parameters Using Tasks

As part of the task, each engine receives a set of input parameters and transmits a set of output parameters.

The following sources provide the parameters the tasks can pass.

- The start engine can generate the input parameters for any engines in the pipeline, and include them in the task.
- Intermediate engines can calculate input parameters as part of their processing, and add them to the task for a downstream engine in the engine sequence.

Task Management

The Axxia architecture provides the following features for managing tasks.

- Task Queueing – Each engine maintains queues for input tasks, called task receive queues, that have buffer management capabilities.
- Sending Backpressure – Engines can send backpressure based on task buffer usage or other specific conditions.
- Mapping Task Priority – Task priority is a value from 0-7 that is mapped to a receive queue. This mapping determines the queues to which the tasks are delivered.

- Maintaining Task Order – Tasks can optionally be created with a task order ID, a number used to ensure that an engine can process packets from a given flow that have the same task order ID in arrival order without restricting the relative ordering between packets of other flows which have different task order IDs. Engines that take advantage of task order ID will have taskOrdering as an input parameter shown in the ASE when building a pipeline that uses them.
- Load Balancing – For accelerator engines consisting of multiple parallel engines, the tasks can be used for load balancing by distributing the tasks among the engines.

Task Receive Queues

Each engine maintains an address space in system memory for one or more task receive queues to store incoming tasks. The number of queues varies, depending on the following engine type.

MPP – The MPP engine has four task receive queues.

NCA – The NCA supports up to 12 task receive queues, six per core.

EIOA – The EIOA engine has two task receive queues for each port.

MTM – The MTM engine has four task receive queues, two task receive queues for unicast traffic and two for multicast traffic.

The task priority is a value from 0-7 that is mapped to a receive queue. This mapping determines the queues to which the tasks are delivered. The following sources can define a task's priority.

- The pipeline start engine. For example, the MPP defines the task priority as a parameter when it sends a new task.
- Flow data. The task priority is set by a value defined for a flow in a flow data table.
- The pipeline configuration.

You specify (in the ASE under Engines-<engine name>-TaskReceive) the mapping of the task priority to task receive queues for each engine. For example, with four task receive queues, you can assign one queue as the highest priority, and assign it tasks with priority 0 and 1. The next highest priority queue receives tasks with priority 2 and 3, and so on.

NOTE Priority 0 is special priority, that is, these tasks can not be dropped. It should be reserved for critical operations and not for regular packet processing.

You can configure the arbitration of the task receive queue servicing using the following scheduling algorithms.

- Weighted Round Robin – Define weights for each task receive queue to define the servicing ratio between queues.
- Strict Priority – The highest priority queue with traffic is always serviced.
- Rotating – Servicing that attempts to strictly maintain servicing queues in a sequence, even if queues have previously been skipped because they were empty. For example, with four queues, if the current queue to be serviced is two, but queues two and three do not have any tasks to service, the fourth queue is serviced. On the next round servicing begins with the third queue, strictly maintaining the servicing sequence as if all queues always have traffic.
- Round Robin – Round robin servicing begins servicing queues with the next queue in sequence after the previously serviced queue. For example, with four queues, if the current queue to be serviced is two, but queues two and three do not have any tasks to service, the fourth queue is serviced, then on the next round the first queue is serviced, always following the last queue serviced.

Task Receive Queue Management

Task receive queues have thresholds levels that trigger actions. You can configure these thresholds and manage the buffers, based on queue occupancy (as number of tasks or as 256B memory blocks) and system buffer availability.

Task buffer management behavior is defined by setting the following attributes for the buffer size of each queue, per task priority.

- Backpressure – Sets the buffer fill level that triggers a backpressure signal.

  Applications can send backpressure messages to monitor the current occupancy of the task queues and signal the EIOA, MTM, MPP or CPU so they can react earlier. With the MPP and CPU the result of the backpressure signal is programmable; the EIOA can send PAUSE frames and in the MTM, backpressure signals can be mapped to queues and schedulers, based on the configuration or scripts.
- If the system configuration does not use the backpressure messages, packets are dropped based on the following discard schemes of the queue.
  - Discard Random – Sets the buffer fill size that triggers the Random Early Discard (RED) buffer management policy. The behavior of the algorithm is based on the RED slope value, and the relationship to the Discard All value.
  - RED Slope – Defines the slope of a vector, from 0 to 1; with a larger slope value defining a more aggressive buffer protection policy.This slope is used to determine the discard probability of incoming tasks when the buffer size is in the range from Discard Random to Discard All. The vector is graphed as a buffer size (X axis value) that triggers the discard probability (Y axis value) based on the RED slope value. The formula is Discard Probability Percentage = Percentage of Buffer (Discard All - Discard RED) x (RED Slope).
  - Discard All – Sets the buffer fill level that ends the RED algorithm and discards all new tasks.

NOTE Engines do not discard tasks with priority 0 when they violate receive queue thresholds. Priority 0 tasks are only discarded based on the system buffer availability threshold for this task receive queue, configured for the Memory Management Block.

When a new task is received, it is assigned to a queue depending on the task priority. Each queue belongs to a Threshold group. Each group has a set of thresholds and a RED slope defined for each task priority. Those values are compared to the buffer state to determine whether the new task is kept or discarded.

You define a set of these values for each task priority using Threshold Groups. Most engines have a single group; the EIOA has two, and the NCA has eight. For example, you can select one of the two defined priority thresholds groups for each port you configure for the EIOA, changing the buffer management policy per port.

The following figure shows two similar configurations. The first example uses a 1.0 RED slope value and the second example uses a 0.25 RED slope value.

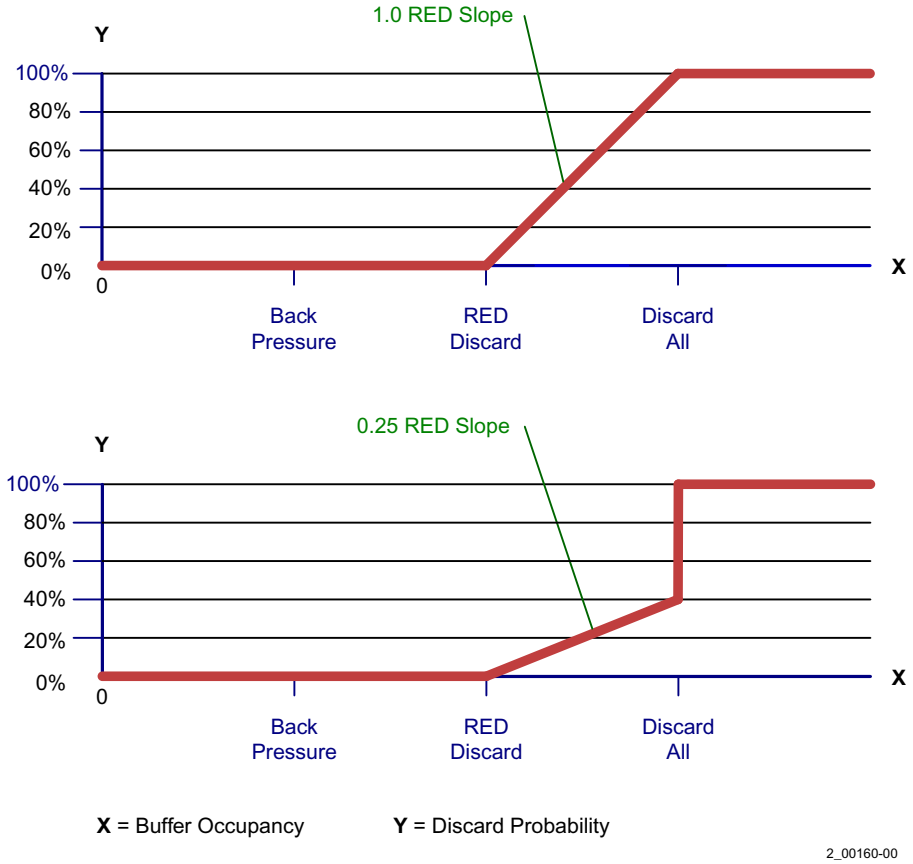

Figure 5 Sample RED Slope Discard Probability Examples

The higher 1.0 RED slope value has the more aggressive buffer protection behavior. Similarly, the maximum discard probability for a given slope is equal to the slope value multiplied by one hundred.

Use the ASE to configure task priority assignments to task queues, task queue scheduling arbitration, and the backpressure, discard random, and discard all thresholds for each task priority.

Maintaining Task Order

Typically, most engines keep packets in order as they process them by maintaining them in unique flows. For some packet flows, it is important to process packets in order in accelerator engines that employ multiple parallel engines. For example, the Deep Packet Inspection (DPI) engine has multiple internal threads for searching. If you are using the DPI engine for cross-packet searches, you must ensure that, as a flow is processed by the DPI engine, the threads do not process different packets of the flow out of order.

To provide this capability, the task includes a field for task ordering called the Task Order ID determined by the application software. This field contains a compressed version of the flow ID. This 12-bit value ensures that, on engines with multiple logical task receive queues, if tasks are arriving concurrently with the same Task Order ID and the same task priority, they are assigned to the same subengine, to maintain their order.

For example, consider a high priority task with a Task Order ID of 6 arrives and is assigned to subengine 0. High priority tasks with the same Task Order ID are dynamically assigned to subengine 0. This is irrespective of the configured subengine assignment algorithm, such as load balancing. If a task with an Order ID of 6 arrives, and there are no other tasks in the flow with the same ID, the configured queue algorithm is used to assign the task to a queue. In this case, the task is assigned to subengine 1.

Additionally, unlike other tasks, those tasks with Task Order IDs, or ordered tasks, are treated as if they have not been removed from the task receive queue until the processing of that task is complete. This ensures that while the engine is processing an ordered task, a task that arrives with the same priority and Task Order ID is not assigned to a different queue.

Load Balancing using Tasks

Tasks can be used for load balancing for engines with built-in load balancing capabilities. These accelerator engines have multiple parallel- processing engines that are independently assigned tasks using the task receive queues.

The Axxia architecture has the following engines with built-in load balancing processing.

- DPI
- SPP
- Configure the NCA to provide multiple processors for pipeline processing

While the MPP has multiple parallel engines, tasks are balanced using task receive queues. Four task queues are used to prioritize traffic entering the MPP engine. Load balancing uses high and low priority logical queues and tracks the number of tasks being processed by that resource.

You can configure task load balancing to work in one of two modes.

- **Processing engine** – Engine uses high and low priority queues as a single resource. New tasks are given to the processing engine with the lowest load in both high and low priority queues. This mode tends to provide a *more even distribution* to the overall task load, but may not provide enough resources to high priority tasks.
- **Priority** – High priority tasks are given to the high priority task queue with the lowest load, and low priority tasks are given to the low priority queue with the lowest load, without regard to processing engine. This mode provides *more resources to high priority tasks*, but may not provide an even distribution to the overall task load.

The NCA does not use the task priority field for mapping tasks to queues. Priority with respect to the NCA queues is determined by the rate at which each queue is serviced by the software.

Flow Parameters

To define a specific set of processing parameters for each processing flow in a pipeline you can define a flow table. You can use application software to create the flow (associated RTE APIs are briefly mentioned later in this chapter). In the table, you can use a programmable flow ID as an index to a set of parameter values that are organized as a flow table entry. The start engine for the pipeline loads the flow table values into the task. Thus, if the flow table is updated, any tasks that have already been launched will continue to use the original information that existed at the time they were launched.

The constant pipeline parameter values you define at configuration are stored in the engine's local template memory (which is not otherwise visible to application software), and the template is indexed by the task.

Use the ASE to manipulate the set of engines in an associated engine sequence. The ASE can determine what engines send the output parameters and receive the input parameters on an octet boundary with an octet-level granularity.

Backpressure

This section introduces the concept of backpressure.

Backpressure Signals

To control congestion and to optimize the use of resources, every engine can send backpressure signals. These are based on task receive queue buffer thresholds or specific conditions, such as port backpressure or other programmable conditions.

Backpressure signals can generate messages to turn backpressure on or off. To avoid excessive signaling when a buffer size hovers around the backpressure threshold and backpressure is asserted, engines use an ASE-configurable attribute (hysteresis). This delays sending backpressure messages to turn off backpressure until the buffer occupancy reaches a low enough value.

Although all engines can send backpressure signals, only four engines can receive the signals.

- NCA - Makes the backpressure signal status available to software
- MPP - Makes the backpressure signal status available to software
- EIOA - Backpressure messages can trigger signalling Ethernet backpressure using PAUSE frames
- MTM - Backpressures queues and schedulers

The CPU and MPP engines permit you to map the backpressure state of different queues using up to 32 flags each. The Axxia device makes the status of these flags available to an FPL program running in the MPP, or to a program running on the CPU. This availability is through variables defined in the ASE generated header files.

Engines can make decisions and take action depending on the backpressure status of a queue or set of queues. For example.

- The ingress EIOA can read port-based backpressure signals to control the reception of new packets.
- The MTM can read backpressure signals to control packet scheduling queues and schedulers. A backpressured queue or scheduler cannot transmit traffic.

You can configure the backpressure behavior in the following ways.

- Assign a single backpressure source to a single destination.
- Assign a single backpressure source to multiple destinations.
- Assign multiple backpressure signal sources to a single destination.
- Assign multiple backpressure signal sources to a multiple destinations.

Namespaces

A namespace is a region of memory that contains one or more engine tables. Engine tables are data structures that some accelerator engines access and maintain.

The Axxia device provides a uniform way of creating and accessing engine tables. Namespaces provide an optimal way to organize and assign the pool of system memory for engine tables, without requiring you to perform detailed memory mapping.

Logically, the software partitions the memory into namespaces and engine tables that are indexed by a 24-bit value. The maximum number of engine table entries in all namespaces that you can allocate is 16M, due to the 24-bit index.

The engine tables hold specific data structures required by individual accelerator engines. Optionally, a namespace can contain more than one engine table.

You configure the maximum number of entries for namespaces and the entry sizes for the engine tables through the ASE. Entries can be added or deleted dynamically.

Engine Tables

The following list describes the different types of engine tables that you can configure within namespaces.

- Scheduling queues, schedulers, and scheduling levels (MTM)
- Shared scheduling parameters (MTM)
- Stream Editor parameters (SED)
- Deep packet inspection contexts (DPI)
- Packet Assembly Block reassemblies (PAB)
- Stateful classification, policing and statistics tables (MPP state engine)
- Security Association information (SPP)
- Cancellable timer information (TMGR)

You can define these structures in your hardware configuration file through the ASE. You can populate them dynamically as needed during processing.

Namespace Configuration

Use the ASE — under Namespaces > Namespace (*name*) — to configure each namespace. To configure a namespace, define the number of entries. This value is the maximum number of entries in the namespace, and by extension the maximum number of entries for all engine tables within the namespace. For each engine table, you define an entry size (width).

For example, to perform IP Reassembly with a requirement to support 1000 simultaneous reassemblies, configure a namespace with 1000 entries and a PABReassemblies engine table within that namespace. The ID of the first entry in the namespace is called the base ID. The ASE can automatically assign the base ID values for all namespaces. The figure below illustrates the base ID and number of entries for a namespace and the entry size for an engine table.

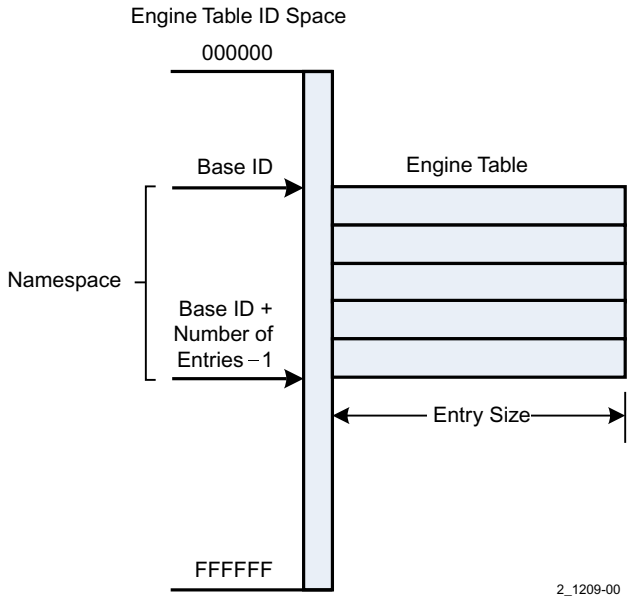

Figure 6 Base ID for Engine Namespace Table

The Axxia architecture uses namespaces to support the following features.

- Engine tables can have different entry sizes. This improves memory usage because you do not have to define the maximum entry size for the engine.
- By configuring multiple engine tables in the same namespace, the namespace reserves a range of IDs for use by multiple engines. In this case, the same ID values can be used to index the corresponding entries in multiple engine tables. This reduces the number of entry ID lookups required and the number of parameters passed between the engines. For example, the MPP and the PAB could share the same range of IDs for a set of packet flows.
- The architecture enables entries from multiple engine tables within a namespace to alternate within memory. This memory layout is called *striping*. Striping allows entries from the different engine tables to be placed in the same cache line. This layout can increase both locality of the memory accesses and performance of some applications. The system cache has a line size of 256 bytes.
- Atomic write operations for engine table entries allow the software to update the engine table entries at runtime, without risking a race condition. For example, the atomic write capability enables updating a scheduling structure while the MTM is using it to schedule traffic.

Engine Table Characteristics

Each engine table has specific features to support its associated accelerator engine. See the engine-specific chapters in this document for additional details specific to each engine.

- **MPP State Engine**

  Use an engine table for the MPP State Engine to set up entries that serve as storage space for C-NP scripts that run in the State Engine compute engine. This storage space is typically used to track state such as counters, statistics, reassembly status, or policing results. Entry size can be up to 256 bytes. An individual entry can store multiple items adjacent to each other within the entry. The FPL code uses a 24-bit index and an 8-bit offset to access the engine table.

- **PAB Engine**

  The Packet Assembly Block engine table reserves packet reassembly memory space. Each reassembly requires an entry of either 128 bytes or 256 bytes. If you use timers for reassembly, you must use the 256-byte entry size. If timers are not used for reassembly, use the 128-byte entry size.

- **SED Engine**

  Although the Stream Editor engine table is not required for SED processing, using it provides additional editing flexibility and features. The SED engine table can provide additional parameter bytes, alternative parameter values, and up to 240 bytes of packet prepended data.

  You can define one of the following engine tables for different types of SED parameters.

  — Direct Parameters – Support 16 or 32 bytes of programmable parameters per entry. These parameters are made available to the SED C-NP script.

  — Indirect Parameters – Support up to 256 bytes of parameters in a fixed structure. This includes any data to be prepended to the packet. This can also include SED commands that indicate, for example, which script to execute or how to perform fragmentation. In addition, up to 32 bytes of indirect parameters can be made available to the SED script.

- **TMGR Engine**

  The engine table for the TMGR is required *only* if you are using the TMGR directly in a pipeline engine sequence to support *cancellable* timers. Only the CPU and MPP engines can be used as start or end engines for the TMGR in a pipeline engine sequence.

  When timers are used indirectly in the MPP, for example, in a hash table entry, or by the PAB or MTM, an engine table for the TMGR is not necessary.

- **MTM Engine**

  The MTM employs the following types of namespace and engine table entities.

  — MTM Scheduler Level – A special namespace used to define a level of scheduling hierarchy. Depending on the level, you can add schedulers to it. The special namespace stores the schedulers as fixed format entries.

  — MTM Scheduler – Schedulers are added as entries in the scheduler level namespace. Their configuration information is stored in a fixed 64-byte format.

  — MTM Queue – Queues do not belong to the MTM scheduler level namespace. They must be added as an engine table in a different namespace. Queue configuration information is stored in a fixed 64-byte format within the engine table.

  — MTM Shared Parameters – Shared parameters do not belong to the MTM scheduler level namespace. This optional engine table defines parameters used by the Buffer Traffic Manager and Traffic Shaper compute engines of the MTM. The entry size is fixed at 32 bytes.

- **DPI Engine**

  The DPI engine can use large or small memory buffers, called *contexts*, for its pattern matching functions. Small contexts are defined as equal to or less than 256 bytes. The DPI engine automatically requests large context memory when needed from the Memory Management Block (MMB). For small context storage space, the DPI Contexts engine table must be defined with an entry size of either 32, 64, 128, or 256 bytes.

- **SPP Engine**

  The Security Protocol Processor engine requires storage space to store information and status for each secure connection. This space is called a security *context* (or a security association). You can configure the entry size up to 256 bytes, which depends on the protocol used, such as IPsec.

Namespaces with More than One Engine Table

The Axxia architecture defines namespaces that specify a range of IDs for a single engine or for multiple engines to access. This capability enables the effective creation of parallel engine tables with identical ID ranges so common traffic flows can use the same ID values in different engines.

For example, the MPP State Engine and PAB can use parallel engine tables in the same namespace. When reassembling packets, the ID that the State Engine uses to refer to the packet reassembly can also be used by the PAB. Typically, accelerator engines share data using tasks. If two or more engine tables reside in the same namespace, you can configure the engine tables so that multiple engines can share data by reading and writing engine table entries.

Engine Table Entry Size and Memory Usage

One of the hardware requirements for the entries in engine tables is memory alignment. Each entry must start on a boundary whose value is a power of 2.

For a namespace with a single engine table, this requirement implies that an entry always occupies a memory chunk that is a power of 2. For example, consider the following figure. If the `MPPStateEngine` engine table has an entry size of 64 and the size is changed so that the entry size is 48, the resulting engine table takes up the same amount of memory, with 16 bytes wasted per entry. To reduce the memory footprint of this engine table, you must further reduce the entry size to 32 bytes. The most memory-efficient entry sizes are power of 2 sizes.

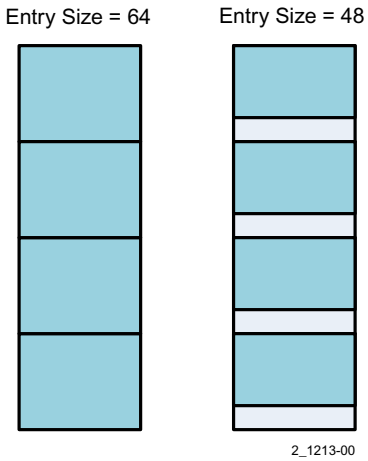

Figure 7 Entry Size Comparison

Even though reducing entry size from 64 bytes to 48 bytes does not save any memory, it does save memory bandwidth. This bandwidth saving occurs because less data needs to be transferred between the accelerator engine and the memory subsystem. The savings can result in marginal performance improvements.

If a namespace has multiple engine tables that are not striped, the same memory efficiency issues apply to each engine table in the namespace. The power of 2 entry sizes are the most memory efficient.

Striped engine tables provide the only case where entries without power of 2 entry sizes can save memory. For example, when you stripe a SED engine table with an entry size of 16 bytes with an engine table for the MPP State Engine with an entry size of 48 bytes, the result fits into a 64-byte cache line. Striping the engine tables together is more memory efficient than leaving them non-striped.

Managing Namespaces and Engine Tables from FPL Software or Using the RTE

For most system designs, the namespaces and engine tables are created through the ASE, reserving memory for table data. The engine tables are then populated and managed at run time, during operation.

After an engine table is defined and the software application starts adding and removing entries, a mechanism is needed to keep track of the entry IDs. For example, when the software sets up a new application flow, it must add an engine table entry. To add the entry it must find a free entry, mark it as in-use and use it to set up the flow. The LSI hardware and software provide two ways to manage entries for a namespace: from an FPL program or through RTE APIs.

Using the RTE APIs for Engine Table Management

The following list shows the RTE APIs for engine table management.

- ncp_rsrc_create – Create a resource enumeration object
- ncp_rsrc_destroy – Destroy a resource enumeration object
- ncp_rsrc_find – Find the named object
- ncp_rsrc_allocate – Allocate contiguous ID(s)
- ncp_rsrc_allocate_fixed – Allocate contiguous ID(s) starting at the given ID
- ncp_rsrc_free – Release previously allocated ID(s)
- ncp_rsrc_isset – Check whether the ID has already been reserved
- ncp_rsrc_query – Query the attributes for a given ID
- ncp_rsrc_dump – Dump a listing of the resources of a specified type or types to standard output (stdout)

NOTE These APIs are generic, that is, they can manage a set of entry IDs for any purpose, including but not limited to engine table ID management.

Engine Table Management using an FPL Program

The fplManaged attribute of a namespace element in the ASE designates whether the engine tables in a given namespace are managed by an FPL program or through RTE APIs. The following restrictions apply.

- An FPL program can manage a maximum number of 16 engine tables.
- The FPL managed engine tables cannot have entries defined statically in the ASE. They should not be accessed through RTE APIs.
- The number of entries in an FPL-managed engine table must be a multiple of 20 and cannot exceed 2^{20} (approximately 1M).

Sharing Data Between Engine Tables

The Axxia architecture uses tasks as the most efficient way for engines in an engine sequence to communicate with each other. This includes sending data from one engine to another. Task communication associates data with specific packets and automatically maintains data and control information integrity.

You can also configure engine tables within a single namespace to share data with each other. This capability is only available for specific accelerator engines.

Sharing data allows entry modifications done by one engine to be visible to another engine. However, sharing engine table entries has a risk. If an entry contains control information, such as memory block addresses obtained from the MMB, the possibility of error exists. If you corrupt the control information, you can lock up the device or corrupt packet data.

To avoid potential problems, only the following engine tables support data sharing.

- MPP State Engine table
- MTM Shared Parameters
- SED Direct Parameters

  SED *Indirect* parameter entries contain control information, so they *do not* support sharing.

Because a system design normally has multiple packets in flight, data sharing can work differently from what is expected in a single-threaded application, as shown in the following example.

- Assume the pipeline is MPP->SED-> (other engines). An engine table for the MPP State Engine shares data with a SED Direct Parameters engine table.
- The MPP maintains byte and packet counts, while the SED reads and uses the counts.
- Seven packets are sent into this pipeline engine sequence, P1-P7, 100 bytes each.

The following figure shows an example of the shared engine table entry latency.

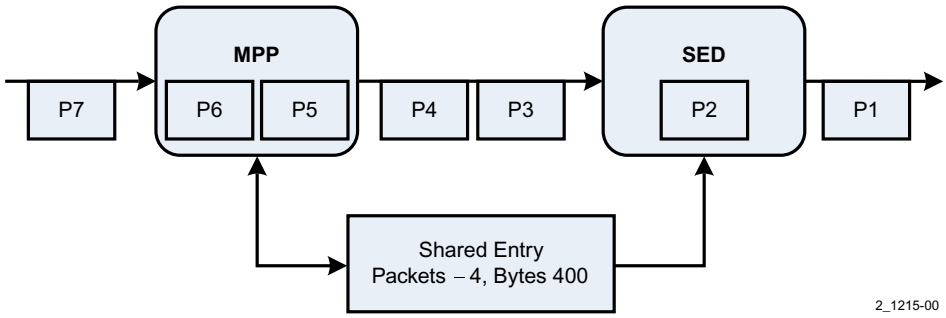

Figure 8 Shared Data Table Entry Latency

Consider what happens when the SED accesses packet P2. If the system worked with one packet at a time, the values would be packets=2, bytes=200. However, by the time P2 gets to the SED, the subsequent packets, P3 and P4, have been completed by the MPP and are waiting in the SED's task receive queue. This means the SED reads the values from the shared entry as packets=4, bytes=400. In general, when engine tables share data, it is difficult to predict the packet and byte values the downstream engine reads. In this example, for the P2 values read by SED the only prediction that can be made is that packets ≥ 2 and bytes ≥ 200.

Using the readThrough and writeThrough Modes in Sharing

Because there are readThrough and writeThrough flags for every engine table, setting them for shared tables does not affect the performance of other engine tables. These attributes are configured per engine table and their values are transferred to the hardware when the base IDs are assigned.

The readThrough and writeThrough attributes place additional constraints on base ID assignment. For example, the area of address space used for engine tables is a factor. The ASE cannot define a namespace that contains an engine table for the MPP State Engine with writeThrough set to true in the same area of internal memory as a namespace that contains an engine table for the MPP State Engine with writeThrough set to false.

NOTE In general, do not enable the readThrough or writeThrough attributes unless you share data between engine tables, because these attributes affect performance.

For the previous example with an MPP->SED pipeline, you must set writeThrough to true for the MPPStateEngine engine table. No readThrough setting is required for the SED Direct Parameters table because the SED does not use the L1 cache (caches namespace table contents between system cache and each engine).

If you have a pipeline engine sequence that includes (other engines) ->MTM->MPP and you need to share the data between the MTM Shared Parameters and the MPP State Engine engine tables, set writeThrough to true for the MTM Shared Parameters and readThrough to true for the MPP State Engine.

Using the Level 1 Cache

For a SED Direct Parameters engine table, there is no Level 1 cache. The SED reads the complete engine table entry for every packet.

For MTM Shared Parameters, the MTM engine uses Level 1 cache. When readThrough is set to true, the merge is done on 16-byte granularity.

The MPP State Engine has the following Level 1 caches.

- Stat (ALU) cache. It has 16-byte entries and is responsible for built in functions such as seIncrement32 and seWrite32, that are invoked from the FPL code.
- State Engine script cache. It has 64-byte entries and is used for caching the contents of param_block1 and param_block2.

The readThrough and writeThrough flags set for the MPP State Engine engine table apply to both L1 caches simultaneously. When readThrough is set to true, the merge is done on a byte granularity for the first 16 bytes of the cache entry size and on a 16-byte granularity for the remaining 48 bytes of the State Engine script cache.

MPP State Engine Data Sharing

Because the MPP uses multiple L1 caches, there are two cases of data sharing that you can use with the MPP.

Case 1: Sharing data between statistics calls and a State Engine script

- Both the readThrough and writeThrough attributes must be set to true for that MPP State Engine engine table.
- The FPL program must call barrier functions to make sure the corresponding writes complete.

The following example shows writing the data with an SE script and reading it with a statistics call.

```
seScript0R(SCRIPT_PARAM_ADDRESS)
seScriptBarrier(SCRIPT_PARAM_ADDRESS)
paramValue = seRead32(SCRIPT_PARAM_ADDRESS)
```

Assume the script takes 4 bytes of data in the parameter block and returns data as the 4 least significant bytes of the result. To get the expected value, call the `seStatBarrier` function to ensure the preceding `seWrite32` call completes before the script is executed.

For more information about ensuring that preceding SE calls complete before the script executes, refer to the *Axxia Communication Processor Functional Programming Language (FPL) Programming Guide*.

Case 2: Sharing data between the MPP and some other accelerator engine

In this case, readThrough or writeThrough should be set based on whether or not the MPP is reading or writing the data. It is not necessary to call the seScriptBarrier/seStatBarrier functions in the FPL program because any outstanding MPP write actions complete before the packet gets to the next engine.

Engine Table Memory Addressing

Within namespaces, you create and access engine tables without concern for the actual location in memory where the engine tables are stored.

Logical Addressing

Each namespace defines a range of logical addresses for entries in one or more engine tables. The entry IDs are logical addresses. They do not directly correspond to memory addresses. There is *no* direct relationship between namespaces or engine tables and memory. Only addresses actually used by engine table entries reserve physical memory. The result is there are no memory gaps or wasted memory for unused entry IDs in an engine table.

The base ID for a namespace is also a logical address. It is used to access engine table data that is significant in its relationship with other namespaces and the available entry IDs. When the ASE assigns a base ID for a namespace, that ID is part of the address used by either an FPL program or in an RTE API call to create and access entries in the engine tables of that namespace. For example, the logical address used to access an engine table entry is the base ID of the namespace plus the entry number. (This logical address is sometimes called the *absolute ID*).

ASE Static Entry Addressing

Engine table entries defined through the ASE are called *static entries*. The ASE gives each static entry an ID that is a relative to the engine table. To access these engine table entries through software requires that the relative ID be added to the base ID of the namespace.

If you define a namespace with two or more engine tables that share data, you must ensure you do not define values for static entries with the same entry ID in more than one of the shared tables. Entries with the same index (entry ID) point to the same data, and writing more than one overwrites any others. Defining a value for a single shared table makes the data available to all of the shared engine tables in the namespace because they share the table entries.

Display Namespace Base Addresses

You can use the ASE to display a namespace or engine table base ID and the corresponding physical address in system memory. Right-click the engine table in the ASE and select Print Base Addresses to print the following information.

- The absolute ID of the engine table entry
- Base address of the engine table entry in system memory. Reading the address in either hardware or the simulation environment provides the entry content.
- For engine tables, the address of the first entry displays in the log.

The interactive debugging tool within the ASE also provides a command to display the physical memory address that corresponds to an engine table entry.

Assigning Base IDs

The ASE software assigns the base IDs for namespaces (which apply to all engine tables in the namespace). LSI recommends that you use the ASE to define all base IDs. The complexity of manual base ID assignment increases as namespaces or engine tables are added or changed.

Optionally, you can manually define the base IDs, typically for testing purposes. If you manually define base IDs, use the following rules:

- Specify the base ID values for each namespace so that namespaces used by the different application components do not conflict.
- Because the hardware addresses engine table entries starting initially with the first 4 bits of the 24-bit entry ID, engine tables with different attribute definitions, such as readThrough or writeThrough settings, cannot be created in the same address range that shares any logical address with the same first four bits of address space.
- Similarly, two namespaces that have at least one type of engine table in common cannot be assigned logical addresses that share the same first four bits of address space *if* the engine tables are defined with *different* entry sizes.

You can manually define base IDs for some namespaces, while allowing the ASE to configure base IDs for other namespaces.

Engine Table Code Generation

The ASE generates code based on your namespace and engine table configuration to help you access engine tables from FPL programs or C programs that call RTE APIs.

When you create namespaces and engine tables in the ASE for your project, the ASE generates code (header files) during compilation to enable you to reference the engine tables, even when the memory location of the table has moved after a different compilation. The ASE generates FPL and C code header information and includes the header files in your ASE project.

FPL Code Generation: Base IDs

The code generation for namespaces provides the *baseID*, *numEntries*, and *entrySize* definitions in the generated FPL header file. The *entrySize* definition is only used for engine tables for the MPP State Engine.

```
#define NCP_NS_BASEID_name              baseId
#define NCP_NS_NUM_ENTRIES_name         value
#define NCP_NS_ENTRY_SIZE_name          value
```

For example:

```
#define NCP_NS_BASEID_IPSecOut          0x100000
#define NCP_NS_NUM_ENTRIES_IPSecOut     0x000100
#define NCP_NS_ENTRY_SIZE_statistics    256
```

In this example, if the IPSecOut namespace contains an engine table with entries IPSecContext(0), IPSecContext(1), and IPSecContext(2), an FPL program can use the following code to access these entries.

```
myTree : 10.1.1.1  fReturn(NCP_NS_BASEID_IPSecOut + 0);
myTree : 10.1.1.2  fReturn(NCP_NS_BASEID_IPSecOut + 1);
myTree : 10.1.1.3  fReturn(NCP_NS_BASEID_IPSecOut + 2);
myTree : BITS:32   invalidIpAddress();
```

FPL Code Generation: SET UP NAMESPACE

If a namespace is managed by an FPL program, the ASE also generates code in addition to the base ID definition.

```
#define NCP_NS_name fplId
SETUP NAMESPACE(NCP_NS_name, baseId, numEntries);
```

For example:

```
#define NCP_NS_atmReassembly                0x1
SETUP   NAMESPACE(NCP_NS_atmReassembly, 0x000000, 0x010000);
```

Here, the *fplId* values are the numbers between 0 and 15 automatically assigned by the ASE to FPL-managed namespaces. There can be at most 16 FPL-managed namespaces. The *fplId* is the hardware resource number from which entry IDs are allocated and the *name* value is the software-managed resource name.

The SETUP NAMESPACE directive enables you to use the hash engine of the MPP to manage the entry IDs. For example, allocate a new entry ID by invoking:

```
fHashAllocateId(NCP_NS_atmReassembly)
```

C Code Generation

The ASE generates the following definitions for C code that calls RTE APIs.

```
#define NCP_NS_name   "name"
```

For example:

```
#define NCP_NS_saStateEngine            "saStateEngine"
```

You can use these constants instead of hard-coding the names of individual namespaces. If a namespace name changes in the hardware configuration file of your ASE project, the name of any hard-coded #define constant will not change and show up as a compile error. If you do not use these generated constants, changes to namespace names are only detected at run time.

Troubleshooting

This section provides troubleshooting information.

Engine Monitoring

The Axxia device provides the following mechanisms for monitoring engine processing and tracking errors.

- **Task Error** – The task contains an internal flag that can be set by the hardware under certain conditions to indicate an error with that task. This is an internal flag that is typically set by hardware. For example, a task error bit might be set when the engine runs out of memory and cannot process the task. Different engines process task errors differently.
- **Engine Interrupts** – Each engine supports interrupts that are tied to specific internal conditions. You can enable the interrupts with the RTE APIs. When enabled, these interrupts can notify the host processor that an interrupt condition is met. Engines have two interrupt types, service interrupts and failure interrupts.

- **Task Monitoring** – Each engine can be configured to monitor packet traffic by counting tasks processed, tasks dropped, or other task characteristics you can select from the available engine counters. These SMON counters can be used to help debug pipeline processing by identifying engines with any significant bottlenecks or errors.
- **Conditional Logging** – The Axxia device enables task tracing for engines logging tasks that match specific conditions you define.

Task Error

The task contains a Task Error bit that indicates when the packet data for a task is corrupted. The packet data can be corrupted and the task error bit set due to the following events.

- When the task is being generated, there is an error writing the packet to memory.
- When the EIOA packet buffer overflows when receiving packet data for a task.
- When an engine determines the task packet data is corrupt.

Depending on the engine, the behavior of the engine in generating or reacting to the task error bit can be different. The following table shows the behavior for each engine.

Table 1 Engine Actions Upon Receiving a Task with a Task Error Bit

Engine	Generates Task Errors	Action when Receiving a Task with a Task Error Bit
MPP	Yes	Starts the Pattern Processing Engine context with the trap handler, and uses the task error as the error condition.
MTM	No, does not write packet data	Passes the input task error out with the output task.
NCA	Yes	Sends the input task error indication to the CPU in the input task header it forms for the CPU.
EIOA	Yes	Writes an incorrect CRC on the Ethernet packet
SPP	Yes	Bypasses security processing, and forwards the task with the task error set.
DPI	Yes	Bypasses pattern inspection and forwards the task with the task error set.
SED	Yes	Does no packet editing and forwards the task with the task error set.
PIC	Yes	Does not perform processing on the packet and forwards the task with the task error set.
PAB	Yes	Does not read task packet data and marks the reassembly context with an exception with task error as the cause. Any task generated from the reassembly context has the task error set.
TMGR	No	Ignores the task error.

Engine Interrupts

Each engine has a set of host processor interrupts that you can enable or mask using the RTE.

The engines support two types of interrupts.

- Service Interrupts - These interrupts signal to the host processor that the engine requires service from the CPU. For example, when the NCA completes a DMA operation, it triggers an interrupt that notifies the CPU that the operation is complete.
- Failure Interrupts - These interrupts notify the host processor of a specific failure with engine processing. For example, a failure interrupt could be triggered by a FIFO overflow or a packet processing error.

Enable the interrupts to monitor the set of conditions for each engine to which you want the host processor to respond.

Task Monitoring

There are two ways to monitor tasks.

- Tracing tasks using task logs. This technique can incur additional processing time.
- Tracking engine counters for processed tasks. This technique uses dedicated hardware in the engines and does not affect timing.

Each accelerator engine has monitoring capabilities provided by a statistics and monitoring (SMON) block.

Each accelerator engine maintains a common set of statistics on tasks and additional statistics about engine-specific functions. You can select counters to monitor using the SMON block. You can configure two SMON counters for each engine. Depending on the engine, you can configure the SMON counters to monitor one or more set of task types, such as tasks received or transmitted, and the processing contexts.

You can use the SMON block to debug pipeline processing, such as an engine that is dropping a large number of tasks.

Conditional Logging

The task trace facility enables you to use the accelerator engines to log the tasks they send. Use this feature to debug the processing path by recording how many times tasks are processed by the engines or, optionally, engine output task information.

Tasks can be logged using one of the following modes.

- Task-based logging – Tasks are launched from a start engine with the debugging flag set, causing each engine that sends the task to log when the task is sent.
- Engine-based logging – The engine is configured to log all tasks that the engine sends.

You configure each engine to maintain its own separate task log. Engines write task logs to dedicated locations in system memory until the memory fills up or writes continuously in a circular fashion.

There are additional mechanisms that provides filtered logging. For example, you can set up the filter to trace only tasks with certain parameter bytes or PDU bytes.

A module within each accelerator engine enables portions of an output task to be copied to a trace log for post-processing and analysis. The task structure uses the following fields for tracing.

- Trace Enable – Enables start engines to trace a task.
- Debug Field Valid – Selects one of two possible formats for the debug field.
- Debug Field – An 8-byte field that supports two possible formats.
  - Trace-oriented format that holds software-generated information including a sequence number and 24 bits defined by software, and two flags indicating a dropped task and a task that was generated by a timer.
  - Performance-oriented format that tracks the engine-to-engine timestamps.

The Axxia task tracing facility provides the following features.

- Trace enabling by pipeline start engines
- Trace filtering
  - Pattern matching tracking – You can define a 24-bit pattern and count the tasks with headers that match the pattern.
  - Conditional start and stop – When a configured number of pattern matches have been counted, the start or stop trigger is asserted.
- Statically allocated memory region for trace logs

Relevant RTE APIs for task tracing include the following functions.

- ncp_task_trace_enable – Enables task tracing for the specified engine
- ncp_task_trace_disable – Disables task tracing for the specified engine
- ncp_task_trace_trigger_enable – Enables triggering for task tracing in the specified engine
- ncp_task_trace_trigger_disable – Disables triggering for task tracing in the specified engine
- ncp_task_trace_trigger_set – Specifies trigger conditions that must be satisfied for tracing to occur
- ncp_task_trace_buffer_get – Retrieves specified engine's trace buffer
- ncp_task_trace_buffer_clear – Clears specified engine's trace buffer

Chapter 2: Modular Packet Processor (MPP) Engine

The MPP is an enhanced version of LSI APP classifier technology. It is programmed using the Functional Programming Language (FPL), and has support engines that are called using the FPL functions.

Overview

The MPP design is an enhanced version of LSI Advanced Packet Processor (APP) classifier technology. You program the MPP engine using the Functional Programming Language (FPL), using the FPL functions to call other accelerator engines. The Axxia MPP provides more than 20 Gb/s of packet processing bandwidth, using IMIX traffic and light touch classification. IMIX traffic is defined as: 56% 64-byte packets, 20% 576-byte packets, and 24% 1518-byte packets.

The MPP includes the following features.

- Up to four pattern-processing engines (PPEs) with 32 processing threads (which are called contexts) each. Each pattern processing engine contains a flow engine and a tree engine.
- On-chip flow engine instruction memory holding 16K flow instructions per PPE that can share the memory between two PPEs to support an effective size of 32K flow instructions.
- Access to 256 KB of on-chip memory for storing classification database information (32 K tree engine instructions) that are expected to be used frequently. Each tree engine inside a PPE also has a dedicated micro-root on-chip memory that holds 1K tree instructions.
- A single DDR3 interface for access to optional external classification database (for example, tree instruction) memory.
- A Hash engine for hardware-assisted exact match-based lookups, namespace ID management, and timer management.
- A State engine for maintaining state and statistics, and for executing script-based state and policing scripts.
- A Function Bus Interface (FBI) to MPP subengines such as the Hash engine and State engine. The FBI logic also contains 16 global registers and the order queue logic.
- A Prequeue Modifier (PQM) for editing, adding, deleting, or duplicating packet data.
- A Semaphore subengine for locking shared resources for atomic operations.
- An MPP Packet Integrity Check (MPP-PIC) subengine, for calculating CRCs or checksums.

MPP Classification Capabilities

The MPP classification engine is a fully programmable classifier that supports simple, complex and hierarchical classification. Its features include:

- Flexible support for nested protocols.
- Hash function based flow identification generation for load balancing.
- Stateful classification with dynamically allocated memory for per flow state.
- Support for variable header positions and classification lengths.

Some examples of classification capabilities include the ability to recognize:

- RLC frames, to generate flow ID based on RLC header.
- TCP flows, to perform TCP termination.
- MAC addresses and VLAN tags, to forward packets.

Features

This section introduces the MPP Engine features and the various subengine components.

Pattern Processing Subengines

The pattern processing subengines (PPEs) perform packet classification using user-defined patterns. All tree patterns are stored in the classification memory. The classification memory includes both on-chip (1 MB) and off-chip classification memories. Use on-chip memory as much as possible to achieve better engine performance for lookup functions, because on-chip memory has less access latency.

Each of the eight PPEs has internal logic that executes flow control and pattern matching instructions and enables the following classification applications.

- Routing tables
- Access Control Lists (ACLs)
- Flow Identification

Hash Subengine

The Hash subengine manages hash tree entries in a hardware-maintained hash table and provides hardware-assisted Namespace and Timer resource allocation. The FPL compiler optimizes the hash memory usage and validates the hash memory capacity, as necessary.

Hash subengine features include the following.

- Performs basic hash table operations.
- Is configurable.
- Cannot be modified from the CPU.
- Can be checked and stopped manually.
- Time-out events can be manually checked and stopped.
- Time-out events are invoked by hardware.
- Configurable static and overflow memory.

State Subengine

The State subengine maintains and retrieves information whose lifetime can span multiple packets. Typically, the State engine helps provide the policing functions, statistics functions, and OAM functions. For more complicated applications, the State engine can provide per session protocol state information or other general purpose state maintenance functions.

The State subengine supports the following features.

- Performs computations and stores state information for the following common uses.
  - Policing.
  - State maintenance.
  - Parameter sharing with other accelerator engines.
- Reports overflow and underflow events for statistical functions and reports special events.
- Provides the following processing functions.
  - Programmable script function that runs a C-NP script.

  - Built-in FPL functions.
    - Arithmetic functions.
    - Boolean logic functions and bit search functions.
    - Memory read and write functions and barrier functions.
    - Sequence generation, checking, and timestamp functions.
- Shares parameters between the built-in and programmable functions.

Prequeue Modifier

The Prequeue Modifier (PQM) edits the packet in the MPP engine, typically before other accelerator engines modify or schedule a packet. The PQM does packet data insertions, deletions, modifications, and segmentation. Invoke the PQM with FPL functions.

Packet modification within the PQM can improve application performance to offload another accelerator engine, if the MPP engine has enough flow cycles available.

The PQM has the following features.

- Evaluates the PQM-specific FPL function calls that it receives from the Function Bus Interface (FBI) associated with a PDU.
- Determines whether to split the PDU into multiple pieces or delete it.
- Determines whether to modify the PDU or insert specific data (any part of the data stream).
- Determines whether to transmit the PDU out of the MPP.
- Consists of the context number, an 8-bit function call number, and up to 105 bits of function specific parameters.
- Provides the termination function to reset all state that is associated with the processing context.
- Provides the function call to set up the launch parameter information for output task processing.
- A work list is built after the PQM calls an FPL function.

The PQM has the following usage restrictions.

- PQM functions must not be called when the context owns an unordered semaphore.
- The PQM must deal with a different data stream than the data stream the MPP uses to perform classification. If you use the PQM to modify packet data, you cannot perform classification on the just-modified packet data.

Semaphore Subengine

When the operation order of the packet with MPP contexts must be considered, use the Semaphore subengine. It defines a sequence of FPL functions to be called atomically and in packet arrival order. It also provides semaphore locks to ensure access to a shared resource operation between context threads is atomic.

The Semaphore subengine is invoked synchronously with the MPP engine and supports the following features.

- Four groups of semaphores, each having semaphore request and release functionality.
  - A semaphore request function requests exclusive access to one out of 1024 available locks. When the requested lock is available, the lock is marked as locked and the context is returned to the scheduler.
  - A semaphore release function releases the lock on the semaphore that was locked by the context on which this function was issued. When the semaphore release function is not issued to release the context owning the lock, the termination of context results in the release of the locks.
- A 10-bit unique flow ID used as an identifier among the different flows. You can use the hash key and index generation capability to compress multiple fields into the 10-bit flow ID.
- Ordered or unordered semaphore calls that are determined by FPL code.
  - A single ordered call to a given semaphore group can only be performed during the context lifetime.
  - A context can acquire an ordered semaphore, release it, and then reacquire it as an unordered semaphore.

MPP Packet Integrity Check Subengine

The MPP Packet Integrity Check (MPP-PIC) subengine operates in a manner similar to the Packet Integrity Check (PIC) accelerator engine. It performs a subset of PIC functions within the MPP engine, such as checking IP checksums and header CRCs. Optimization of the MPP-PIC subengine provides maximum performance to eliminate the overhead incurred by using the external PIC engine for small packet calculation.

The MPP-PIC subengine offers standard CRC and generic CRC calculations. Although you can use software-programmed CRC polynomials, the usage comes with a performance cost, because software-programmed CRCs require up to 16 times longer to complete.

The MPP-PIC subengine is accessible by the PPEs by using an FPL function call. The parameters in the FPL function control the operation performed by the MPP-PIC engine, which performs the requested operation and returns the result to the PPE. The MPP-PIC subengine does not modify the packet data.

The MPP-PIC subengine supports the following features.

- Up to four CRCs to supply the polynomial.
- Layer 4 protocol checksum operations that permit you to do IP header checksum, including IPv6 extension headers and IPv4 options headers.
- Generic checksum you can use to calculate a checksum on any portion of the packet.
- The MPP-PIC subengine is multithreaded and does not maintain operation order.

The MPP and PIC accelerator engines use the task ring and exist as separate engines in pipelines. You can access the MPP-PIC subengine only through FPL functions.

MPP and Engine Sequence Interaction

The MPP is a programmable engine for processing traffic and can be a start engine or an end engine.

When the MPP is an *end* engine, the ASE software generates a root function in the FPL code, in the FPL header file, that is called to start MPP processing for that engine sequence. When the MPP is a start engine, the root function uses the FPL transmit function that starts a task down another engine sequence.

A header file created by the ASE contains information on root functions, engine sequences, and launch functions. It also contains backpressure information and namespace definitions.

The MPP can also be a backpressure aggregation point. You can monitor the backpressure status of up to 32 different backpressure signals in the MPP. You can use this information to avoid congestion later in the packet path by discarding packets earlier in the processing. This avoids discarding packets after significant processing has taken place, and helps prevent high-priority traffic from hitting congestion bottlenecks.

MPP Block Diagram

MPP Pattern Processing Engines

The MPP contains four pattern processing engines (PPEs) that are programmable with the Functional Programming Language (FPL). Each PPE supports up to 32 contexts, which are treated as process threads. With the four PPEs, the MPP engine supports up to 128 processing threads. Each thread processes using a run-to-completion model, supported by a scheduler that assigns new tasks to threads as they become free

The following figure shows the PPEs, MPP subengines, and the interconnect to the Virtual Pipeline instance.

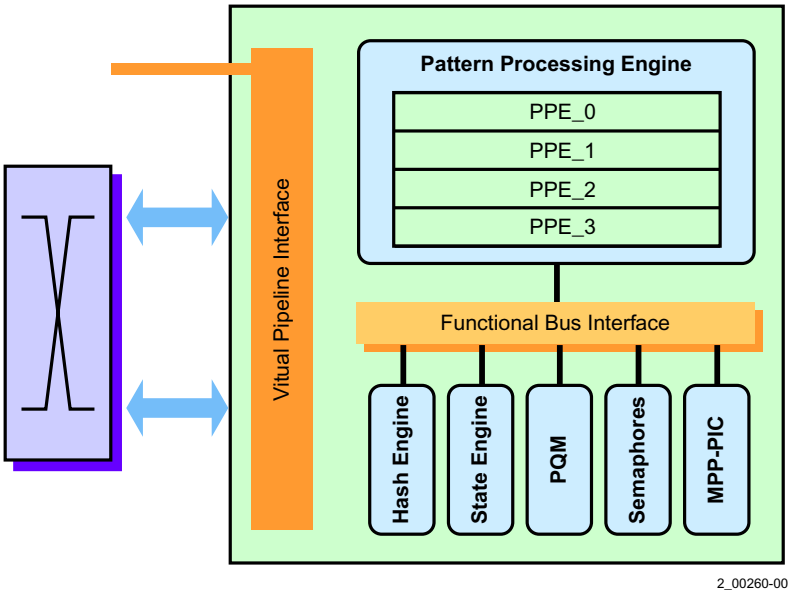

Figure 9 MPP Block Design

Each PPE contains the following elements.

- One flow engine and one tree engine.
- A flow engine that runs flow instructions for program control flow.
- A tree engine that runs tree instructions to perform longest prefix matching-based pattern matching. The MPP uses the matching for lookups and decision-making. The tree engine can have a dedicated 1000 tree instructions stored in memory.
- 16 K words of cache for flow instructions. This is low access latency internal memory for the first 16 K words of flow instructions. The remaining flow instructions are stored in external memory, either on or off the Axxia device.
- 1 K words of cache for tree memory. This is low access latency internal memory for the first 1 K words of tree instructions. The remaining tree instructions are stored in external memory, either on or off the Axxia device.
- An interface to the Functional Bus Interface (FBI) that provides access to built-in functions.
- Connection to a classification memory fabric that connects to eight internal and one external classification memory controllers. Each internal classification memory controller connects to a 16K words of classification memory (8 x16K words equals 128K words equals 1 MB), and one external classification memory controller connects to the external DDR3 memory.

MPP Hash Engine

The MPP Hash Engine is a support engine that is invoked by FPL function calls. The Hash Engine is useful for wire-speed learning and stateful processing. The Hash Engine has the following features.

- Supports exact match lookup, insert, and delete operations
- Number of hash table entries limited only by SDRAM capacity.
- Virtually unlimited number of logical hash tables supported.
- Hash entries can also be associated with timers and used for dynamic and atomic namespace ID allocation. You can easily set time-out values, for example, for packet reassembly.

MPP State Engine

The State Engine allows the MPP to perform computations and to store state in Namespace tables for policing, state maintenance, and parameter sharing. It also reports overflow and underflow events for statistical functions and reports special events by setting a flag with the compute engine scripts.

It provides the following types of processing functions.

- Built-in function – Processed by dedicated hardware.
- Programmable function – Runs a C-NP script on the MPP compute engine.

You invoke the State Engine functions using FPL function calls. The State Engine enables the coherent sharing of parameters between the built-in and programmable functions.

MPP Prequeue Modifier (PQM)

The Prequeue Modifier modifies the packet in the MPP, typically before the packet is modified or scheduled by other accelerator engines. The PQM performs packet data insertions, deletions, modifications, and segmentation. You invoke the PQM primarily with the `fModifyPdu` and `fTransmitData` FPL functions.

The MPP performs classification on unmodified packet data. If you use the PQM to modify packet data, you cannot then perform classification on the just-modified packet data. To perform MPP classification on the modified packet data, you would need to send the packet through the MPP again.

PQM data modification functions build a worklist which is applied to the packet when it is transmitted.

For additional information about the Prequeue Modifier, see the *Packet Modification* section, in the Modular Packet Processor Engine chapter.

MPP Semaphore Subengine

For most packet applications, you do not have to consider order of operation of packet processing contexts in the MPP. If that is required, the Semaphore subengine defines a sequence of FPL functions to be called atomically and in packet arrival order, using flow locking. The Semaphore Engine ensures exclusive access to a shared resource allowing for atomic operations between context threads.

The Semaphore subengine defines up to four groups of semaphores supported with a 10-bit flow ID, used to reduce the chance of a false dependency between flows. You can use the hash key and index generation capability to compress multiple fields into the 10-bit flow ID.

For additional information about the Semiphore Subengine, see the *Atomic Access Using Semaphores* section, in the Modular Packet Processor Engine chapter.

MPP Packet Integrity Check Subengine

The MPP Packet Integrity Check (MPP-PIC) subengine, which operates similarly to the Packet Integrity Check (PIC) accelerator engine, performs a subset of PIC functions, such as checking IP checksums and header CRCs, without leaving the MPP. It offers several standard CRCs, as well as a generic CRC capability that allows you to use software-programmed CRC polynomials, although with a performance cost. Software-programmed CRCs can take as much as 16 times the duration it takes hardware CRCs to complete.

The MPP-PIC is accessible by the PPEs using an FPL function call. The parameters in the FPL function call control the operation performed by the MPP-PIC. The MPP PIC performs the requested operation and returns the result to the PPE. The MPP PIC does not modify the packet data.

Table 2 MPP PIC Supported CRC protocols.

CRC Protocol	Bit Polynomial
ROHC-3	$x^3 + x^1 + 1$
CPS-5	$x^5 + x^2 + 1$
UTRAN-7	$x^7 + x^6 + x^2 + 1$
ROHC-7	$x^7 + x^6 + x^3 + x^2 + x^1 + 1$
ATM-8_0x00	$x^8 + x^2 + x^1 + 1$
ATM-8_0x55	$x^8 + x^2 + x^1 + 1$
ROHC-8	$x^8 + x^2 + x^1 + 1$
OAM-10	$x^{10} + x^9 + x^5 + x^4 + x^1 + 1$
UTRAN-11	$x^{11} + x^9 + x^8 + x^2 + x^1 + 1$
UTRAN-16	$x^{16} + x^{15} + x^2 + 1$

In addition, you can specify up to four CRCs in which you supply the polynomial.

The MPP-PIC is multithreaded and does not maintain operation order.

The MPP and PIC accelerator engines exist as separate engines in engine sequences, but the MPP-PIC is a subengine to the MPP that you can only access through FPL functions.

Input and Output Parameters

MPP End Engine Input Task Parameters

When the MPP is an end engine, you can configure it to receive up to 32 bytes of input parameters. Using the ASE, you can split the 32 bytes of input parameters, on a byte level, to match the size of the output task parameters from engines earlier in the pipeline. The input task parameters you create this way are generated in the header file, and the variables are populated with the correct values at run time through the template merge process.

The 32 bytes of parameters are loaded into the eight $iParam variables ($iParam0 through $iParam7) in 4-byte increments. The engine sequence ID can be included in iParam7 by the MPP.

The MPP also receives the following system-generated parameters that describe task attributes and state.

- 32 bits from the Task Header that includes:
  - Input task queue ID (8 bits)
  - Task priority (3 bits) – A value from 0-7 that defines the task priority
  - taskError flag (1 bit) – If set, invoke the error handler.
  - timerValid flag (1 bit) – If set, override the root function defined for the engine sequence and call the root function with ID 255, the timer boot code function.
  - launchID (18 bits) — The launching ID for the task. The lower 8 bits define the launching engine. The task engine IDs are:

    ncp_engine_min = 0

    ncp_engine_eioa0 = 1

    ncp_engine_timer =2

    ncp_engine_pab = 3

    ncp_engine_mpp = 4

    ncp_engine_spp = 5

ncp_engine_sed = 6
ncp_engine_cpu = 7
ncp_engine_dpi = 8
ncp_engine_pic = 9
ncp_engine_mtm = 10
ncp_engine_expander = 11
ncp_engine_eioa1 = 12
ncp_engine_max = 15

NOTE The most significant bit of the Task Header is reserved.

- queueFillLevel (32 bits) — The input task queue fill level in bytes.
- rootAddress (20 bits) — The root function address for the Virtual Pipeline instance, which is useful for debugging.
- context (7 bits) — The current context ID. The upper two bits are the PPE ID; the lower five bits are the flow or thread ID for the PPE. Each PPE supports 32 threads.
- flowControl (32 bits) — A bit mapped register value that provides you information on the backpressure state for up to 32 different engine task receive queues and other objects, such as MTM queues or schedulers, MTM scripts, and output ports. You define the mapping in the ASE in the MPP engine configuration under the Backpressure Destinations element. The destination variable names you assign to backpressure signal sources are added as variables to the generated FPL header file.

MPP Start Engine Output Task Parameters

As a start engine, the MPP can send up to 32 bytes of output task parameters.

With the ASE, you define those 32 bytes as the parameters the downstream engines receive from the MPP. You also use the ASE to generate the header file with the prototypes for the functions that pass those parameters to the task structure.

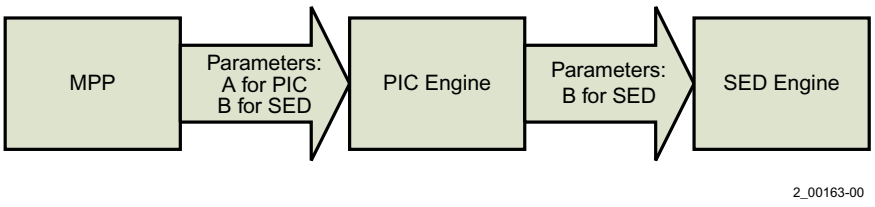

Figure 10 MPP Output Task Parameter Passing

Depending on the way you define the parameters in the ASE, you can have the task parameters passed out of the MPP using a single function, `fTaskParms()`, or you can use multiple functions.

By using the Stage capabilities of the ASE, you can group the parameters for multiple `vpParms()` functions that are outlined with the generated header file. You can send parameters as they are calculated, instead of having to store them in FPL register space until they are sent.

Packet Processing

The MPP can classify based on any combination of fields anywhere in the packet, along with any associated task parameters, policing and state maintenance script results, and any previously stored state information. The set of fields that the FPL program examines can be based on the contents of previous fields along with any state information from previous processing.

The compiled FPL program runs on the MPP pattern processing engines. When a packet arrives on an MPP task queue, the MPP assigns the incoming task to an available processing context and automatically invokes the FPL program at the entry point for the associated Virtual Pipeline instance to classify and process the packet. Each processing context runs the FPL program concurrently – each pattern processing engine runs in parallel. The FPL program uses function calls to invoke subengines as needed to support the classification and state maintenance processing. The MPP multiple contexts and its simultaneous multithreading execution model allow it to effectively tolerate variable latencies due to memory accesses and subengine execution.

Based on the results of classification, the FPL program instructs the MPP to launch packets and commands as needed for downstream engines. For more information on FPL for the Axxia devices, refer to the *Axxia Communication Processor Functional Programming Language (FPL) Programming Guide*.

C-NP is the other programming language for the MPP. You use C-NP to program the compute engine inside the MPP state engine. You can write programs using C-NP and the programs are executed in the MPP state engine compute engine. You can call state engine functions as subroutines from your FPL programs to perform policing, statistics, and state maintenance functions. For more information on how C-NP is used in the Axxia devices, refer to the *Axxia Communication C-NP Language Reference Manual* and the *Axxia Communication Processor Compute Engine Programming Guide*.

Classification

A key aspect of classification processing is pattern matching and table look-ups. The MPP supports two mechanisms to support this functionality: trees (which use the tree engine) and hash tables (which use the hash engine).

Tree-based pattern matching supports longest prefix match, exact match, range match, multi-field classification, and ordered search. Trees can only be updated from the host processor so the tree mechanism is best suited for tables which need no more than tens of thousands of updates per second.

Hash-based pattern matching supports exact matching and dynamic match key update through FPL, and it allows timer and namespace ID management to atomically be attached to hash engine calls. FPL uses a hash tree to efficiently manage hash key generation over a potentially disconnected set of packet fields and other data. Hash-based matching is ideal for high pattern learn rates (millions of learns per second are supported) and is also ideal for exact matching of wide bit patterns. It is also very efficient when a pattern needs to be associated with a timer and/or a namespace ID.

In contrast, the tree engine supports more versatile matching capabilities but its performance is inversely proportional to the total width of the pattern being matched. However, the hash engine is not a limitless resource and it does consume a nontrivial amount of system memory bandwidth.

For more information about using tree-based or hash-based matching, as well as flow function usage in FPL programs, refer to the *Axxia Communication Processor Functional Programming Language (FPL) Programming Guide.*

Dynamic Resource Management

Hash-based pattern matching permits timer and namespace ID management to atomically be attached to hash engine calls. The hash engine can perform the following hash table operations.

- Search a hash table for a key match and return the associated result value.
- Insert a new entry.
- Update an existing entry if it exists, else insert a new entry.
- Delete an entry.
- Manage FPL timers and dynamic name allocation.

When managing timers the associated hash entry holds a timer. When managing namespace IDs the associated hash entry holds a namespace ID. The following types of hash entries are based on what type of information the entry associates with the hash key.

- A maximum of 51 bits of data (future functionality will support hashable trees, a function ID so that a hashable tree can be called a function just like a normal tree function can).
- Allocated namespace ID
- Timestamp and 30 bits of data
- Timestamp and allocated namespace ID.

For more information about the hash engine functions and dynamic resource management, refer to the *Hash Engine Functions* section in the *Axxia Communication Processor Functional Programming Language (FPL) Programming Guide.*

Names, Timers, and Hash Table Structure

The hash engine maintains a set of 16 stacks to support namespace ID allocation. A hash call that requests a new namespace ID (which the hash engine associates with the current hash key) must specify a namespace stack ID from which the namespace ID is to be allocated. Since each namespace stack can hold up to one million IDs, this permits you to dynamically allocate and deallocate up to 16 million namespace IDs. Namespace IDs which are on the same stack have the same entry size, but multiple ASE-defined namespaces with the same entry size can share the same stack so long as the aggregate total number of entries in the stack is no more than 2^{20}. The ASE automatically manages the allocation of application namespaces to internal hardware tables. For more information on namespace allocation, refer to the *AXX2500 Family of Communication Processors Axxia Software Environment (ASE) Reference Manual*.

When an FPL hash engine call starts a timer, the hash engine creates a timer. This timer expires at the FPL-requested time. If FPL subsequently issues a hash engine call to update or cancel the associated timer, the hardware simply updates the associated timestamp stored in the hash entry.

When the timer expires, built-in FPL software first checks the stored timestamp against the current time. If the timestamp indicates that the timer has not expired (because it was reset one or more times previously), the built-in FPL software asks the timer manager (TMGR) engine to set a new timer to expire at the new requested expiration time.

If the timer has not been reset and expires, the built-in software calls the hash timer expiration function that the FPL application registered. While this behavior is transparent to FPL application software, it is visible in the FPL profiler in the simulator, which records the calls to built-in FPL software when the timer manager expiration tasks arrive at the MPP. These function calls are also visible when carefully measuring performance on the hardware when many concurrent timers are run with short time-out intervals.

Physically, the MPP supports up to 1024 hash tables. For optimal performance relative to memory space consumed each direct mapped table can be provisioned to hold 1.3 to 2.0 times the average expected number of table entries. If the application requires more than 1024 hash tables, it can include a logical hash table ID as one of the fields it uses to generate the hash key, thereby supporting millions of logical hash tables.

For more information on the hash engine, see the *Hash Engine* section.

For more information about the hash engine functions and timers, refer to the *Hash Engine Functions* section in the *Axxia Communication Processor Functional Programming Language (FPL) Programming Guide*.

Stateful Processing

The FPL application software uses the MPP state engine to maintain and retrieve information whose lifetime can span multiple packets. In traditional network transport applications, the state engine can provide such functions as policing, statistics, and OAM. In more sophisticated applications, the state engine can provide per-session protocol state information or any other general purpose state maintenance functions. The state engine helps the MPP perform computations, plus store and retrieve state information in Namespace tables. It also supports reporting statistics and computation overflow and underflow events for statistical functions, plus reporting special events by setting a predicate bit in a compute engine script.

The state engine provides the following types of processing functions.

- Built-in – Processed by dedicated hardware. For example, seIncrement64(MYCOUNTER) increments the 64-bit counter at the namespace ID and offset specified by MYCOUNTER. MYCOUNTER has a 24-bit namespace index and an 8-bit byte offset.
- Programmable – Runs the selected C-NP script on VLIW compute engine.

For more information on how applications invoke State Engine functions using FPL function calls, use built-in functions, and invoke C-NP script functions, refer to the *Axxia Communication Processor Functional Programming Language (FPL) Programming Guide*.

For more information on writing state engine C-NP scripts, refer to the *Axxia Communication Processor Compute Engine Programming Guide*.

For more information on writing state engine C-NP scripts, refer to the *Axxia Communication Processor C-NP Language Reference Manual*.

The state engine functions refer to memory identified by a 24-bit namespace address and an 8-bit byte offset. System software provisions a memory layout table based on how you configure the MPPStateEngine engine table in the ASE to specify how to map each entry to system memory. The ASE generates #define statements in the include file for the configuration to define the 24-bit base ID for each entry used by the state engine. For example, if you define a GlobalCounters variable then the ASE generates a `#define` statement for the symbol NCP_NS_BASEID_GlobalCounters to specify the 24-bit base address of this counter.

Operationally, assuming that the FPL application calls a state engine function with a namespace address plus offset X (where X is 32 bits), the state engine hardware performs the following actions.

- Uses the upper 24 bits of X to look-up the entry in the memory
- Adds the offset in the lower 8 bits to the resulting memory address
- Aligns the address to W (where W is 8, 16, 32, or 64 bits based on the size of data that the function uses) by ignoring the low order memory address bits as needed. For example, an seIncrement64 function aligns the data referenced to 64 bits by ignoring the least significant 3 bits of the resulting address.

The individual function descriptions in the *Axxia Communication Processor Functional Programming Language (FPL) Programming Guide* specify the size of data that each built-in function uses when it is not obvious. If a built-in function returns n bits, then the upper bits not returned are undefined and should not be relied on. FPL functions can reference zero, one, or two addresses plus offsets. The number and size of the memory regions that each function uses is specified when the function is written and configured. System software records this information in a script attribute table that the state engine hardware consults to check memory bounds when processing a script invocation. State engine function calls which reference one or more namespace address plus offsets are executed in a sequence defined by the memory address plus offset parameters referenced.

The state engine hardware looks at the function number (in the `AcpFbiFuncs.h` include file) to determine whether the function being invoked returns a result and whether it is a built-in function or a script function. Bit 7 of the function number determines whether the MPP should wait for the function result or continue. Bit 6 of the function number determines whether it is a built-in or a script function. Bits 0-5 determine which function within the group is being called.

State Engine Built-In Functions

The MPP provides the following types of built-in state engine functions.

- Arithmetic
- Boolean Logic and Bit Search
- Memory Read/Write and Barrier
- Sequence Generation, Checking, and Timestamp

Some of these operations can be done on 1-byte, 2-byte, 4-byte or 8-byte boundaries.

For more information on state engine built-in functions, refer to the *Axxia Communication Processor Functional Programming Language (FPL) Programming Guide*.

State Engine Compute Engine

C-NP functions can reference zero, one, or two namespace addresses (24 bits) plus an 8-bit byte offset. A namespace can reference 16B, 32B or 64B of parameters.

The number and size of the memory regions that each script uses is specified when the script is written and configured. System software records this information in a script attribute table that the state engine hardware consults to check memory bounds when processing a script invocation.

For more information on coherency restrictions on the namespace address plus offset parameters provided when invoking a script, refer to the *Axxia Communication Processor Functional Programming Language (FPL) Programming Guide*.

For more information on how to write state engine C-NP scripts to achieve customized behaviors, refer to the *Axxia Communication Processor Compute Engine Programming Guide* and the *Axxia Communication Processor C-NP Language Reference Manual*.

Exceptional Event Communication Using a Producer/Consumer Queue

To handle overflows, underflows, and similar exception events the C-NP script detects, the state engine uses a circular FIFO in system memory, which system software configures at initialization time. Only the host, not the MPP, can read this FIFO. Built-in arithmetical state engine functions also record overflow and underflow events in this same FIFO.

This 16-byte structure identifies the overflow source (script or built-in function)) and identifies which counters have overflowed. An ALU operation indicates an arithmetic overflow by writing the following data to the FIFO.

Table 3 ALU Operation Arithmetic Overflow Exception Bits

Byte Location	Description
Bytes[0]	8-bit function ID to decode the size of the register(s) that overflowed
Bytes[1]	Bit-0 First overflow/underflow event valid Bit-1 Overflow or underflow from the first operation 0: Underflow 1: Overflow Bit-2:3 Reserved Bit-4 Second overflow/underflow event valid Bit-5: Overflow/underflow from the second operation 0: Underflow 1: Overflow Bit-6:7 Reserved
Bytes[2:7]	Reserved
Bytes[8:11]	FPL namespace address plus offset associated with the function
Bytes[12:15]	Reserved

A state engine script reports an overflow event by setting a flag. When it sets this predicate and the script finishes, hardware generates the event, writes it out to system memory and updates the FIFO producer pointer. This must be done even when the same script is being invoked back-to-back so that the overflow information is not lost.

The format of the following 16 bytes is:

- Byte 0 – Function ID of the operation that caused the overflow (supplied by hardware)
- Bytes 1-7 – Seven bytes of data from the script register file from bytes 49:55
- Bytes 8-11 – Index address space and offset for the first parameter block (supplied by hardware)
- Bytes 12-15 – Index address space and offset for the second parameter block (supplied by hardware; 0 if not used)

For more information on how to direct the state engine hardware to write data to this FIFO to signal an exception to the software, refer to the *Axxia Communication Processor Compute Engine Programming Guide*.

For more information on detecting overflow and underflow events in C-NP code, refer to the *Axxia Communication Processor C-NP Language Reference Manual*.

Enforcing Ordered Execution

Context Ordering Semantics. The MPP has a number of contexts (threads of execution) and each context processes a task and runs in parallel with other contexts. Contexts often perform processing steps in an order that is different than the order in which the contexts started. For example, a context that started processing after another context can nonetheless finish its processing first. There is no guarantee in preserving order in the execution of contexts, though the MPP logic guarantees that any tasks launched from a context will be launched from the MPP in the order that the corresponding context started processing in the MPP.

To maintain arrival order in a context, you must define ordered FPL functions — external functions that are guaranteed to run for each task in the order the tasks were received on the corresponding input task queue.

NOTE Each ordering queue can only ensure that the first function is ordered for a given context in an ordering queue. Any subsequent function calls on a previously ordered context, during the life of the context is unordered.

For example, sequence number generation (for example, with seSequenceGen) would normally need to be done using an ordered function call if sequence numbers are to be generated in task arrival order within a flow. You can invoke both ordered and unordered versions of the same function.

NOTE Ordering external functions forces sequencing of the data path, which guarantees the order of external function calls. However, this can result in reduced throughput. Ordering should only be used when it is really necessary for correct application functionality.

The pattern processor execution logic for external functions provides hardware support for ordering external function calls issued on the FBI. It tracks incoming tasks and context assignments and ensures that the specified external functions are invoked in the correct order. External functions that must be ordered are grouped into equivalence classes. All functions within a class are equivalent when being numbered. When one of the functions in this class is executed, the function invocation is held up until previously allocated contexts also invoke one of these functions or they terminate.

The MPP FBI logic uses the following ordering queue sets for ordering specific sets of functions on the specific ordering queue.

- State engine function ordering queue – Four ordering queues in this set. Use the SETUP SEORDERQUEUE directive to assign up to 4 state engine functions to each ordering queue in this group.
- Semaphore function ordering queue. There are four semaphore ordering queues, one for each semaphore request function (so there is no need to explicitly assign semaphore functions to particular order queues).
- General function ordering queue for ordering hash engine, PQM and MPP-PIC functions. This set is made up of two ordering queues. Use the SETUP GENORDERQUEUE to assign up to four general MPP functions to each order queue in this group.

For more information on using the SEORDERQUEUE and GENORDERQUEUE directives, refer to the *Axxia Communication Processor Functional Programming Language (FPL) Programming Guide*.

Functions defined as ordered and issued on contexts that are not at the head of the ordering queue are queued and wait their turn, until one of the following conditions is met.

- All contexts started previously have been terminated, or requested to not order any functions.
- The context on the given queue has already called an ordered function
- The FPL application has requested to not order any functions on this context

The ordering logic keeps track of the order in which contexts were started. The functions processed by a specific ordering queue are sent out in the order the contexts were originally started in. Function calls that do not need to be ordered bypass the ordering logic. An FPL context that will not be making an ordered function call from that point on can call fSeUnblockOrderQueues, fNoSem, and fGenUnblockOrderQueues functions (for state engine, semaphore, and general MPP functions respectively) to release the contexts in the relevant order queues waiting for it.

Atomic Access Using Semaphores. For most FPL applications, you do not have to explicitly account for the concurrent execution of multiple FPL contexts. However, to explicitly guarantee atomicity across a series of operations the MPP semaphore subengine helps applications to acquire and release semaphores to lock and unlock access to an associated resource ID.

Semaphore functions are automatically assigned to their own order queues and are normally ordered, though unordered versions of them are also defined. You can define up to four groups of semaphores. FPL passes a 10-bit ID to each semaphore call to identify a particular flow or resource for which the FPL context is requesting exclusive access. If needed you can use the hash key and index generation capability to compress multiple fields into this 10-bit flow ID.

Internally, the MPP manages dependency analysis using a 512-entry table which the application partitions between the semaphore groups in use. If more than one application generated resource ID maps to the same entry in the 512-entry table it causes a false dependency and a context will unnecessarily wait to acquire a semaphore when there was no actual resource conflict. However, since the chance of this happening is relatively small, the overall impact on performance should be fairly minor.

To efficiently use the semaphore functions, observe the following restrictions.

- A context owning a semaphore in semaphore group N (where N is 0, 1, 2, or 3) can only get semaphores in groups greater than N. That is, if a context owns a semaphore in group 1, it can still acquire a semaphore in groups 2 and 3, but not 0 or 1. This restriction eliminates a possible circular wait that could cause a deadlock.
- Semaphore calls can be ordered or unordered, as determined by FPL. Only one ordered call to a given semaphore group can be done during the lifetime of a context. However, a context can acquire an ordered semaphore, release it, and then reacquire it as an unordered semaphore.
- For better performance, issue semaphore release functions on semaphore-owning contexts and on semaphore groups where the context owns the semaphore.
- While a context owns an unordered semaphore, it must not issue any MPP PQM function calls.
- While a context owns an unordered semaphore, it must not issue any ordered function calls.

The last two restrictions avoid a deadlock condition when semaphore functions are mixed with other ordered functions.

Packet Modification

The Prequeue Modifier (PQM) allows you to modify the packet in the MPP, typically before the packet has been modified or scheduled by other engines. PQM data modification functions build a worklist which is applied to the packet when it is transmitted. The PQM allows you to perform packet data insertions, deletions, modifications, and segmentation. If you use the PQM to modify data, the modified data is unavailable to the classification functions until after the MPP starts the associated task and the modified packet is sent back into the MPP.

MPP Interaction with Engine Sequences

The MPP can be both a start engine and an end engine for any engine sequence. When a packet associated with a task arrives on an MPP task queue, the MPP automatically invokes the FPL program to classify and process the packet. At any point in the processing, the MPP can launch commands or a potentially modified portion of the current packet into a Virtual Pipeline instance. The MPP hardware handles all ordering, coherency, and serialization to guarantee that all outgoing tasks are sent out in the order that the corresponding incoming task came in. This ordering is guaranteed even if the processing of a single incoming task generates multiple outgoing tasks. Ordering is guaranteed within an input task queue (regardless of the incoming or outgoing Virtual Pipeline instance) but is not guaranteed between input task queues.

An FPL program has an entry point for each engine sequence that is terminated by the MPP. With the ASE, for each engine sequence that terminates at the MPP, you configure the FPL root function that is called for each incoming packet to that engine sequence. The ASE generates an FPL header file (.fplh extension) which the FPL program includes to define these root functions.

Each arriving packet carried in a task can have some number of associated task parameters. These parameters (up to 32 bytes worth with any additional parameters being discarded) are presented to the FPL program as the built-in variables $iParam0 through $iParam7, where each $iParam variable contains 32 bits of parameter information. For more information on these variables as well as other variables derived from the incoming task (for example, $pduLength, $taskHeader), refer to the *Axxia Communication Processor Functional Programming Language (FPL) Programming Guide*.

The FPL program launches zero or more tasks as a Virtual Pipeline instance origination point and runs until it calls a terminating function such as fTransmit. Typically, the MPP launches a single task for every incoming packet. This task generally consists of the packet (potentially modified using the FPL PQM operations) and associated parameters.

These task parameters provide additional information associated with the results of classification needed by downstream engines. The parameters can originate either directly from the FPL program (from registers or constants) or indirectly from the FPL program specifying a flow ID which the hardware uses to look up information in the MPP flow table. However, some applications might decide to split up a single packet into multiple subpackets and generate an output task for each subpacket. Similarly, some FPL applications might only call the state engine to record state information about the packet that it processed (perhaps something as simple as incrementing a counter) and discard the packet without generating a new task.

You can determine which tasks to generate in response to an incoming packet, to decide what pipeline each generated task should be launched into, and provide any associated parameters that are needed downstream to process that task.

The ASE provides assistance in launching tasks into engine sequences. The MPP can replay packet processing and thus launch multiple tasks down the same or different engine sequences with the same packet data. For each engine sequence into which the MPP originates tasks, you use the ASE to configure the sequence name and the set of output parameters expected to be supplied. The ASE uses this information to provide an entry for each originating engine sequence in the header file that it generates. Each of these entries defines an alias for a function named fVpParms_<pipeline_name>, where <pipeline_name> is the name you give to the engine sequence in the ASE. You can call this function to identify the outgoing engine sequence and fill in the parameters needed when originating tasks into that engine sequence. Once this has been done, you can call fTransmit (or a related function) to launch the task into the engine sequence.

The PQM logic receives any fModifyPDU data function calls followed by one of the output task formation function calls. The context has to issue at least one fTaskParams function call (which specifies the flow ID) before issuing any output task formation function call, otherwise the PQM logic discards the packet as part of error processing. The fTaskParm calls can also be done in stages with each stage providing a portion of the 32 bytes of output task parameters that the MPP can generate.

Because of the way the PQM generates output tasks, for best performance an FPL context should avoid generating more than four output tasks for any incoming task. If many copies or pieces of an incoming packet are required, it is usually more efficient to have the MTM multicast expander generate these copies and have the SED or MPP PQM remove that portion of the original packet that is not needed for the new packet.

FPL Flow and Tree Instruction Fetch

FPL software uses the PPE flow and tree engines. In general, the flow engine code provides control flow and the tree engine code performs lookups. The MPP can fetch flow and tree instructions either from on-chip memory (which serves as a static cache) or from the external pair of DDR3 x9 classification memories if one or both of these optional memories are populated.

Fetching instructions from the on-chip memory results in the highest performance. This is especially true for flow instructions. It is recommended that you allocate all frequently used flow instructions to on-chip memory and allocate infrequently executed code to external classification memory.

The MPP supports 16K on-chip flow instructions per PPE, where two 16K memories can be cross connected together and shared between a pair of PPEs, to increase memory capacity. For large applications it is best to configure the memory for highest capacity, sharing between engines, since the performance cost of flow instruction fetch conflicts between the two engines will likely be small compared to the performance cost of fetching flow instructions from external classification memory.

While, from a functionality standpoint, although it has some impact on performance, it is helpful to be able to fetch flow instructions from external tree memory when required, as this supports much larger applications and also facilitates in-service upgrade processing where the active set of flow instruction software is being changed.

The MPP supports 1 MB of on-chip classification memory. You can partition this memory into a maximum of four replicated copies, each dedicated to one PPE, to reduce potential conflicts among tree engines fetching tree instructions. For most applications it is best to leave the memory as a single 1MB memory shared by all tree engines. If more tree instructions are needed than can fit in the on-chip memory, one or both of the x9 DDR3 classification memories can be populated. Populating both of these memories provides the highest performance.

It is possible to trade off performance and capacity in these memories by varying the degree to which data is replicated in DDR3 SDRAM banks (where total replication provides the highest performance but the lowest capacity and vice versa) and the ASE provides three such configuration choices. The algorithm generator software determines how to map patterns learned at the application level into tree instructions stored in the on and off chip tree instruction memories in an attempt to maximize performance and minimize on-chip classification memory usage.

Task Debugging

Use the `fTransmitDebug` and `fTransmitDataDebug` functions to turn on task debugging. Task debugging is not specific to the MPP.

Exception Handling

The MPP supports the standard Axxia mechanisms for reporting exceptions. The MPP engine checks for interface and internal errors and registers these error indications and additional error information in *sticky* status registers that can be read and cleared through the configuration interface. System software can configure exception reporting hardware to determine which error indications generate interrupts.

MPP subengines implement ECC-based single bit correction and single bit and multibit detection for all its large or persistent data structures and for external classification memory. Smaller data structures are protected via parity error detection. Any error detected leads to a sticky interrupt status bit being set in the associated module and an interrupt optionally being generated, based on the configuration of the interrupt reporting hardware.

Many MPP exceptions, such as an incoming task with the task error bit set, will also result in the currently executing context being sent to the FPL trap handler. The hardware passes an indication of the particular issue that caused the trap as an argument to the trap handling function. In general, exceptions that are detected while the issuing context is running will cause traps.

For more information on the trap indications and what each trap means, refer to the *Axxia Communication Processor Functional Programming Language (FPL) Programming Guide*.

Specifics as to how selected MPP engines handle exceptions and generate interrupts are discussed in the following subsections.

State Engine

The state engine checks that the incoming function indexes are valid entries in the MPPStateEngine engine table. An interrupt is set indicating this condition. For blocking functions (the context is returned when the function completes) invalid indexes are flagged and the werr flag is set when data is returned to FBI. The FBI converts the werr flag into a trap code back to the PPE.

The state engines uses a Producer-Consumer Queue (PCQ) model reporting overflow and underflows from built-in functions and exceptional events signaled by the C-NP script setting predicate bit 9. The hardware supports two modes for servicing the exceptional event FIFO event queue: interrupt mode and polling mode.

In interrupt mode, the state engine reports events to the host processor via a service interrupt. Software will service the contents of the event queue through appropriate API or NCA function calls. The service interrupt can optionally be generated based on a configurable number of events written to the event queue or on a configurable time-out value. Using this option, when an entry is added to the event queue and the configured interval of time has elapsed, an interrupt is generated.

In polling mode, the host processor will periodically check the state engine write and read pointer information and process the contents of the event queue as needed. The host processor will update the read pointer value once it has processed any overflow information.

MPP Packet Integrity Check Engine

In addition to detecting illegal events or illegal function calls, the MPP-PIC engine also detects illegal packet formats while doing calculations. This detection is based on enforcing packet consistency checks and field value checks as defined by the applicable networking standards. The result of the operations are reported to the FPL program as normal function returns, not as a trap to the error handler. In addition, the engine logs any exceptions it detects in sticky status register configuration bits which can optionally be configured to trigger an interrupt.

Hash Engine

Similar to the PAB and MTM engines that issue timer requests, the MPP Hash Engine checks the number of available 256 byte memory blocks against a configurable threshold when calling the fHashAllocateResource, fHashAllocateUpdateResource, and fHashCheckTimer functions. This check is needed to prevent issuing a timer start request if there is insufficient memory available. If there are insufficient 256 byte memory blocks available the associated hash function will generate an exception and the application software should delete the associated Hash Table entry (which is in effect corrupted at this point) and take any application-level corrective action is required to account for the failure to create the requested timer.

There are two different types of Hash Engine exception handling. One type is trap-conditions that can occur as a result of uncorrectable embedded-memory errors or hash-table structure errors. This type results in an FPL trap. Another type of exception-handling is dedicated response codes for function-call processing that result in exceptions.

MPP System Monitoring Capabilities

The MPP supports the standard Axxia mechanisms for system monitoring (SMON) functionality. The MPP SMON functions permit you to monitor a variety of internal conditions, and may be programmed to interrupt if a programmed threshold is exceeded. Each SMON instance contains two sets of logic and you can program each independently regarding the condition to monitor, the threshold to compare against, and whether an interrupt should be generated when the threshold is crossed.

Internally, the SMON control register controls the overall configuration of SMON, including the selection of the condition to monitor. Use the parameter register to qualify the condition being monitored — for example, an MPP context number for a per-context type of condition. The counter register shows the current count for the condition being monitored, and you should zero this count with software before starting. The maximum capture register shows the maximum value observed for the condition being monitored. The current value register shows the current value for the condition being monitored. The timer register shows the number of clocks elapsed since starting, and should typically be zeroed by software before starting. By default the timer counts single core clocks, but may be scaled to lower frequencies to allow sampling or averaging over longer periods of time.

For more information on operational details about the functions that use the MPP SMON capabilities through the Axxia RTE software, refer to the *Axxia Communication Processor RTE Reference Manual*.

MPP Task Debugging

The MPP supports the standard Axxia mechanisms for task debugging. Each of the output task forming FPL functions (fTransmit, fTransmitData, and fTransmitDataRestart) has a corresponding version (fTransmitDebug, fTransmitDataDebug, and fTransmitDataRestartDebug) that generates an output task with task debugging enabled. The FPL application can pass up to 24 bits of information to the debugging version of these functions.

Chapter 3: Memory Management Block (MMB) Engine

This chapter describes the Memory Management Block (MMB) engine, its features, and explains how to use it.

Overview

The Memory Management Block (MMB) manages memory data blocks based on requests from client engines. The MMB allocates the blocks, tracks the number of memory block accesses, and de-allocates the memory block contents that are not needed.

The data blocks are stored in sections of 256 byte, 2 KB, 16 KB, and 64 KB data blocks. The device stores each memory section in a dedicated memory pool, where you configure the minimum pool size. The MMB engine communicates with all accelerator engines whenever a task requires a memory block to store either task or PDU information.

The MMB engine has configurable internal storage for data block addresses. The internal memory is used by the MMB to store the addresses of available or *free* data blocks in a stack. The MMB engine contains a local cache and local buffer of free addresses to optimize the allocation and deallocation of data block addresses. The internal storage reduces the latency of storing and fetching the addresses in system memory.

The MMB engine manages a reference count for each data block. The reference count indicates the number of times the data block is accessed prior to being deallocated. The reference count is stored in the first 16 bytes of a data block. When the reference count is reduced to zero the data block is deallocated.

You configure the number of blocks of each size that are available to the accelerator engines.

The MMB engine issues a NACK when the number of free data blocks falls below a programmed threshold. This is useful for 2 KB data blocks, which are used for tasks and data. You can set an independent threshold for each. You can also configure the MMB to have additional data blocks reserved for tasks.

The MMB engine can issue an interrupt when the number of free data blocks fall below a threshold defined by the ASE. The MMB interrupts are accessed through the configuration ring. The MMB interrupt is cleared through software.

The MMB does not send backpressure based on the availability of blocks; the requesting engines are responsible for sending the backpressure signals.

Features

The MMB manages data blocks in system memory by performing the following tasks.

- Allocates blocks of main system memory to client engines.
- Keeps reference counts for all allocated blocks.
- De-allocates in-use memory blocks as directed by client engines.
- Provides memory usage statistics for client engines.

NOTE The MMB does not send backpressure based on memory block availability. Client engines must determine when to send backpressure signals because of low memory resources.

The MMB has a Memory Statistics interface which is a low performance interface that conveys memory usage statistics to memory clients that need it. For example, the MTM engine can use memory usage decisions to make packet discard decisions.

Chapter 4: Packet Assembly Block (PAB) Engine

This chapter describes the Packet Assembly Block (PAB) engine that performs flexible packet reassembly and editing.

Overview

The Packet Assembly Block (PAB) is an accelerator engine used to flexibly reassemble and transmit multiple segmented packets. The supported commands determine whether all or part of a reassembled packet needs to be transmitted, which lets you perform segment-level editing.

The PAB accelerator engine enables reassembly of multiple segments into the same reassembly buffer based on different requirements. The segment put into the reassembly buffer can be programmed with the required packet size and the starting position located in the shared buffer. The reassembly buffer holds up to 64 KB.

The PAB engine can simultaneously reassemble up to 2^{24} packets. It uses the cache and memory to hold reassembly context data during the assembly process. It also uses the cache to store a dynamic, reassembly context packet state in a table called the Connection Database (CDB).

Features

The PAB engine has the following features.

- Operates as a generalized holding buffer with assembly, transmit, retransmit, and delete capabilities for incoming data streams.
  - Supports inserting or extracting data from anywhere in the same single assembly buffer.
  - Supports gaps in the same single buffer.
  - Specifies to the bit level to support protocols such as the Medium Access Control-hs (MAC-hs).
- Supports the following traditional reassembly functions.
  - IP defragmentation
  - AAL5
  - AAL2
  - SSSAR
- Supports the following generalized holding buffer and sliding window protocols for the transmit and retransmit buffer functionality.
  - TCP origination and termination
  - MAC-d protocol
- Buffers and schedules packet data based on events, rather than rates or weights, especially when sending varying amounts of data.
- Supports millions of reassembly contexts and simultaneous reassemblies.
- Reassembles packet segments.
- Transmits either partial or complete segments.
- Edits segments to different locations in the reassembly buffer.
- Supports timer expiring and maximum reassembly size exceeded in order to end a reassembly context.
- Checks sequence numbers for internal integrity checking.
- Checks the status of a reassembly context.
- Monitors reassembly context resource usage.

PAB Functions

The PAB engine is programmable using commands sent using a task. Each task includes a single command, defined by a set of common parameters. These parameters can have different meanings for different commands.

Each command is used to achieve one or more of these functions:

- Identifying the range of segment data that should be included in the reassembled packet.
- Defining the placement of the segment in the reassembled packet.
- Defining the range of data of the reassembled packet to be transmitted.
- Transmitting the reassembled packet.
- Discarding the data from the reassembly context.
- Checking and returning the status of a reassembly context.
- Editing at bit-level.
- Time out Reassembly.
- Checking the sequence number.

Identifying the Segment Data for Reassembly

Each time you add a segment to a reassembly context, you can enqueue all of the packet data, or you can define a range of the packet data that you want to be added. To do this, you define parameters for the Enqueue command that edit the segment for reassembly by defining an offset at the beginning, used to define data to be removed from the start of the packet data, and a length, which can be used to remove data at the end of the packet.

The offset parameters, definable separately for the initial segment of the reassembly context, define a starting point for the segment within the packet data, measured from the start of the packet.

The length parameter defines the size of the segment data, starting from the beginning of the packet, or, if a starting offset is used, from the starting offset. A value of zero for the length parameter enqueues all of the packet data from the starting offset.

Because of the flexibility of the Axxia architecture, you can choose to edit the packet data using a different accelerator engine than the PAB. For example, you could use the MPP `fModifyPdu` function to edit the packet before sending it to the PAB. Similarly, you could send the packet to the SED for modification before sending it to the PAB.

Defining the Segment Placement in the Reassembly Context

When you add a segment to a reassembly context, you define the starting point for the segment in the reassembly context, called the write offset. The write offset is measured from the start of the reassembly context.

Segments can be added in any order. You can, for example, add a segment that creates a gap by adding the third segment to the reassembly context before the second segment arrives. Or, you can add a segment that overlaps another segment and overwrites the existing data in the reassembly context, as an additional way to edit packet data.

A special case arises when you plan to transmit data more than once from a reassembly context. Once you have transmitted data from a reassembly context, you cannot enqueue a segment that overwrites that reassembly context's transmitted data range or any data that precedes that data. This restriction is to prevent the same packet data from being referenced by two or more different tasks, which could cause packet data to be unintentionally overwritten.

To keep track of the start of writeable reassembly context memory space, the connection database uses a variable called the Minimum Enqueue Offset. This variable points to the first available byte that can be written to in the reassembly. Any enqueue offset you use must be equal to or greater than the Minimum Enqueue Offset. The Minimum Enqueue Offset is initialized to zero when a new reassembly context is created, and updated when packet data is transmitted.

Defining the Reassembled Packet Data to be Transmitted

When your reassembly is complete, you can transmit it as a packet to the next engine in the engine sequence. When you transmit the packet, similar to enqueuing segments, you can define a start offset and a length to select the data in the reassembly that you want to transmit.

For convenience and optimal performance, the PAB has a command that enqueues the last segment of a reassembly and transmits the reassembled packet. In this case, the start offset and length refer to the segment, not the reassembly, so the last segment data is added to the end of the reassembly. When you need to transmit less than the entire reassembly, use a separate enqueue and transmit commands.

The PAB does not keep track of the sizes for individual fragments in the reassembly, only the largest reassembly packet size. If the transmitted part of the reassembly contains gaps where no data has been written, the content of the gaps is undefined.

Transmitting the Reassembled Packet

The PAB supports two types of transmit commands:

- The Transmit command sends the reassembled packet and prevents any additional editing of the original packet data that has been transmitted, sending the new packet and its task to the next engine in the engine sequence.
- The Transmit Copy command creates a new packet by copying the reassembled packet to another memory location, creates a new task that references the new copied packet, and transmits the new task to the next engine in the engine sequence. This command does not affect the original reassembly context data, that is still available for editing and transmission. This is useful for applications that need to send multiple versions of a packet that vary slightly in content.

Using combination commands, such as an Enqueue and Transmit, reduces the number of tasks you have to send to the PAB.

Discarding the Reassembly Context

Each active reassembly context takes up memory space in the Namespace table created to hold the PAB connection database. The Namespace needs to release an entry's reassembly context data (deallocate memory) for one of the following reasons.

- The reassembly is complete and the Axxia device transmitted the reassembled packet.
- An error in the reassembly happened and the reassembled packet should be discarded.
- The reassembly context either timed out or exceeded the maximum size defined, called a sizeout.

You can delete a reassembly context or a subset of reassembly data using the discard command. The discard command can delete all or part of the reassembly data. It starts from the beginning of the reassembly and deletes the number of bytes defined in the length parameter; a value of zero deletes the entire reassembly.

For a time out or sizeout, you can select a flag for either that automatically discards the current segments in the reassembly context when the time out or sizeout condition is met. If another segment is then sent to the reassembly context, it is treated as if it is the initial segment sent to that existing reassembly context. If the flag is not set in either case, the time-out or sizeout event causes the PAB to send the packet to the next engine in the engine sequence.

Timers can be set to start or restart with each new segment enqueued, or only on the initial segment. You can use any enqueuing command to override and stop the timer for the reassembly context. Timers are reset whenever a reassembly context is terminated.

For cases where the state of the reassembly is unknown, you can use the Cleanup command to reinitialize the reassembly.

Checking Reassembly Context Status

The Check Status commands sends a task that does not affect the reassembly context data, and force the PAB to generate an output task with the state information for the reassembly context that you requested. This feature is most useful for debugging during development.

Using the PAB

The PAB creates, adds to, transmits, and deletes packet reassemblies through commands sent in tasks to the PAB from another engine, typically the MPP or the CPU. You can perform the following tasks with the PAB commands.

- Create a reassembly context.
- Add packet segments to the reassembly context.
- Check the status of a reassembly context.
- Transmit all or part of the reassembled packet from the reassembly context.
- Delete the reassembly context.

The PAB supports 14 base commands, and a set of options for those commands in the form of parameters. A task sent to the PAB includes a command and parameter options that define the action for the PAB to take for a given reassembly and, if included, packet segment.

The input tasks sent to the PAB result in adding or deleting data to the reassembly, clearing the reassembly, transmitting all or part of the reassembly data in an output task, or generating an output task with status information.

Creating a Reassembly Context

Before you can create a reassembly context in the PAB, you must define a Namespace for the PAB and add a PAB Reassembly table. Each entry in the table tracks a reassembly context, so the number of entries you define should be equal to the maximum number of simultaneous reassemblies that you plan to support.

The entry size can be one of two sizes: 128 bytes or 256 bytes. If you use timers, for timing out reassemblies, you need to choose 256-byte entries, otherwise use 128-byte entries.

The engine that sends commands to the PAB must keep track of the reassembly table entries: adding and deleting entries, and keeping track of the traffic flows that use the reassemblies. In the MPP, the Hash Engine offers a set of functions for managing tables that you can use to efficiently manage the PAB table. By maintaining a hash table in the MPP that mirrors the PAB entries, you can track status and add and delete entries as they are created or deleted in the PAB table.

You create a PAB reassembly context by using one of the following Enqueue commands to add a segment.

- Enqueue – Adds a segment to a reassembly context.
- Enqueue with Transmit – Adds a segment to a reassembly context, then transmits all of the reassembled packet to the next engine in the engine sequence.
- Enqueue with Transmit Copy – Adds a segment to a reassembly context, duplicates the packet data in memory, then transmits all of the duplicated reassembled packet to the next engine in the engine sequence.

NOTE Transmit Copy differs from a Transmit, because it creates a new copy of the packet data in memory that the output task points to, instead of pointing to the original packet data as a Transmit command does. The packet data sent with a Transmit Copy command can be modified without affecting the original packet data. Similarly, modifying the original data does not affect the copy created by the Transmit Copy command

- Enqueue, Transmit, and Discard – Adds a segment to a reassembly context, transmits all of the reassembled packet to the next engine in the engine sequence, then deletes the reassembly context.

To start a new reassembly context, the enqueue command must be assigned a reassembly index, a pointer into the reassembly table, that represents an entry not currently being used for another reassembly. The segment does not have to be the first segment of the packet being reassembled; a new reassembly can be created by enqueuing any segment of the packet.

Adding Segments to Reassembly Contexts

The PAB enqueue commands can also add segments to existing reassembly contexts. You can control the placement of each segment in the reassembly context space, overlapping segments or leaving gaps, so that in addition to strictly reassembling segments, you can perform byte-level and bit-level segment editing.

You can also define two different byte offsets: an offset to be deleted from the start of the initial segment, and an offset to be deleted from all other segments.

Transmitting All or Part of a Reassembled Packet

While the Enqueue Transmit combination commands send all of the reassembled packet data, the PAB also offers Transmit commands that allow you to select any part of the reassembly context data to send out with the output task.

The PAB engine Transmit commands include:

- Transmit – Transmits all or part of the reassembled packet, with pointers in the output task that point to the original packet data. The Transmit command restricts the transmitted data, and any other data preceding the transmitted data in the context, from being edited further by the PAB.
- Transmit Copy – Duplicates the packet data in memory to create a new packet, then transmits all or part of the reassembled packet, pointing to the new packet duplicated data in the output task.
- Transmit Discard – Transmits all or part of the reassembled packet, then deletes the reassembly context.

Another option to optimize command use for transmitting is to manually reset the sizeout value to less than the reassembled packet size when you enqueue a segment, triggering a sizeout and forcing a transmit.

Clearing a Reassembly Context

When you have completed a reassembly context, you need to clear the reassembly table entry, to free memory and to make the reassembly index available for a new reassembly.

Using the Enqueue and Transmit commands that include Discard — such as Enqueue, Transmit and Discard or Transmit and Discard — deletes the reassembly context. Or, you can delete a reassembly context using the Discard command, set to discard all.

In cases where the status of a reassembly context is unclear, you can re-initialize a reassembly context using the Cleanup command.

In addition, the PAB Discard command allows you to delete a number of bytes you specify from the start of the reassembly context, up to the entire reassembly. Use the Sticky Discard command to delete all the current data and any additional data that is received by the reassembly context until the reassembly context is cleared by a Transmit with Discard, Discard, or Cleanup command.

Monitoring Reassembly Context Status

The PAB engine has the following commands that can check the status of the reassembly context database.

- Get Status Reassembly State – Returns the status for the given reassembly context, using the standard output task status parameters plus the current sequence number, a timer ID, and a timestamp.
- Get Status Reassembly Priority Memory – Returns the memory usage for the reassembly context's priority, and the total memory usage for all priorities, both in 265-byte units.
- Get Status Priority Memory – Returns the memory usage for reassemblies with the indicated priority, and the total memory usage for all priorities, both in 256-byte units. The command requires that the task use the reassembly context ID field to indicate a priority.

Each of these commands generates an output task from the PAB, without affecting the state of the PAB reassembly contexts.

Bypassing the PAB

When the PAB is part of an engine sequence, you may need to send both packets that require reassembly and packets that do not require reassembly through the pipeline. The Passthrough command permits you to send a packet task to the PAB without initiating a reassembly context. Instead, the PAB simply passes the packet with the input parameter packet pointers through to the next engine in the pipeline. This can be useful for maintaining packet order.

Using Bit-level Operations

The PAB supports bit-level editing of packet segment data through the use of the BitFields parameter.

The PAB engine supports the following bit-oriented operations.

- Inserting a bit stream into an existing assembly at an arbitrary starting bit position relative to the beginning of the assembly.
- Transmitting a bit stream from an assembly using an arbitrary starting bit position.
- Advancing an assembly head pointer and its length status to delete data from the front of the assembly

Using Timers with the PAB

You can use timers with the PAB engine to time reassemblies for time-out error cases or to trigger the transmission of packets for time-sensitive applications. Timers are associated with reassembly indexes.

The PAB uses the Timer Manager services to maintain timers. It has a current time counter that synchronizes with the global clock and maintains the order of timer requests and events.

You can start a timer for a reassembly context by providing a value for the TimerValue parameter in the input task and selecting a TimerControl action in the Advanced Flags field: no action, stop, start unconditionally or start on first reference to an inactive reassembly context.

If multiple timers are started for a reassembly context, only the most recent value is stored in the reassembly CDB.

NOTE If you use timers for any other function other than cleanup, it may be difficult for the engine sending commands to the PAB, such as the MPP or CPU, to know the latest status of the reassembly context.

Using Sequence Numbers

You can use sequence numbers to ensure that the tasks sent to the PAB are not dropped by the system by having the PAB verify the sequence number of each segment that it receives for the reassembly. You do this by enabling sequence number checking and supplying a sequence number for each task you send to the PAB.

If the SeqCheckEnable parameter is set to one, the PABengine checks that the SeqNumber parameter supplied by this command is equal to the previous sequence number stored in the CDB plus 1 mod 256, provided that the CDB location indicates that the previous command for the reassembly had sequence checking enabled.

Note that the sequence number continues when a new reassembly context is created with this ID; this allows the feature to cover the initial Enqueue to a reassembly context. If the check fails, the PAB action marks the reassembly context with an error, frees up any memory allocated to this reassembly context, drops all future enqueues to this reassembly context, and optionally sends a task if ExceptionNotify is 1. When output tasks are sent normally, such as by a Transmit, the Exception Notify bit does not matter.

The PAB engine writes the SeqNumber and SeqCheckEnable parameter values to the CDB for every command except for the GetStatus and PassThrough commands. When a particular reassembly index is reused for a new reassembly context, by default, the previous sequence numbering is used for that reassembly index.

You can modify the sequence checking behavior by setting another parameter, the BlockSequenceCheck parameter. By setting this bit, you can turn off sequence error checking for this command, and, if used with the SeqCheckEnable bit set, reset a reassembly context sequence check to start a new count for the reassembly context in the CDB.

NOTE GetStatus and PassThrough commands do not update the reassembly CDB sequence checking state or allow sequence checking. time-outs do not affect the sequence check state of the reassembly context.

PAB Configuration Overview

Configuring the PAB engine in the Axxia Software Environment (ASE) requires you to define the following elements.

- Priority and global buffer management thresholds.
- Task Receive queue buffer management policies. The PAB engine supports both high-priority and low-priority task queues.
- A Namespace table for the connection database (CDB).

Additional initialization options using the RTE include enabling input task debugging with the mode register. This specifies whether or not debug data is supported for input tasks.

PAB Task Receive Queue Configuration

PAB has two task receive queues, the high and low priority task receive queues. Tasks are transmitted to the PAB with a 3-bit task-priority value. The PAB maintains a table that you can configure in the ASE that maps the received task priority to a PAB task receive queue.

You can set the arbitration between the high and low priority task receive queues as strict priority, simple round robin, or weighted round robin.

PAB Task Input Parameters

The following table shows the PAB task input parameters.

Table 4 PAB Parameter Encoding

Field	Size	Description	
Operator	1B	Defines the PAB command, with 3 sticky bits.	
ReassemblyIndex	3B	Reassembly ID Default is 0xFFFFFF.	
Length	2B	Length information in unit of bytes. With the Enqueue command, the length value of 0 means enqueue all data from the input packet from the Start/First Offset to the end. Likewise, with the Transmit command, the length of 0 means transmit all data from the WriteOffset to the end in the reassembly. Default is 0x0000.	
StartOffset	1B	Number of bytes to remove from the segment to be reassembled. For the initial segment to arrive in the reassembly context, use the FirstOffset instead of the StartOffset. Default is 0.	
FirstOffset	1B	Number of bytes to remove from the initial segment to arrive in the reassembly context. Default is 0.	
SizeoutThreshold	2B	Maximum size of reassembly in bytes. Value of 0xFFFF indicates that there is no limit in reassembly size. Default is 0xFFFF.	
WriteOffset	2B	Use the following information for both Enqueue and Transmit commands ■ When the Enqueue command is used, the WriteOffset indicates the byte location in which this segment is to be placed for reassembly. The value of 0xFFFF means append at the end of current reassembly for the Enqueue command. ■ When the Transmit command is used, the WriteOffset indicates the beginning byte location to be transmitted from the reassembly. For the Transmit command, value of 0xFFFF means transmit from the beginning of the reassembly. ■ For combined Enqueue and Transmit commands, the WriteOffset is used for enqueueing, indicating the byte location in which the segment is to placed, and the entire reassembly is transmitted. Default is 0xFFFF.	
TimerValue	3B	Time delta for the timer. A value of 0 indicates no timer. Default is 0x000000	
AdvancedFlags	1B	**Bit**	**Field**
		7	ExceptionNotify
		6	BMIgnore
		5	DiscardOnTimeout
		4	DiscardOnSizeout
		3:2	TimerControl
			00 – No timer is used
			01 – Stop
			10 – Start unconditionally
			11 – Start timer if the task references an inactive reassembly
		1	SeqCheckEnable
		0	BlockSequenceCheck
		Default is 0	

Field	Size	Description	
BitFields	1B	**Bit**	**Field**
		7:5	LengthBitOffset
		4:2	DestBitOffset
		1:0	Reserved
		Default is 0.	
SeqNumber	1B	Segment Sequence number. Default value of 0	

PAB Task Output Parameters

The PAB output task sends 12 bytes of output status parameter information and the reassembled packet data, if included, to the next engine in the pipeline.

PAB Task Output Parameters without the GetStatus command

The PAB task output parameters contain the following elements.

- Status byte – Contains the reassembly's sticky bits (from the CDB, and updated by the current task), and status bits indicating if the task experienced an exception or referenced a reassembly which had previously experienced an exception.
- Three-byte reassembly index value.
- Four bytes of CDB current state – Used for the reassembly (as updated by executing the current task). Note that if the CDB Active bit is 0, the remaining fields are undefined. If the CDB is in the exception state, that is, the first status byte Exception Code is a non-zero value, the priority field is undefined.

Table 5 PAB Output Parameter Format (without the GetStatus command)

Field	Size	Description
Status	1B	Status byte. See definitions in the *Status Byte Encoding* table.
Reassembly ID	3B	Reassembly index specified by the input task PAB parameters.
CDB State	4B	CDB State as defined in the *CDB State Encoding* table.

Table 6 Status Byte Encoding

Bits	Field
7:5	Sticky bits
4	Prior Exception. 0 – Exception status (if any) is due to an exception encountered by this task 1 – Exception status (if any) is due to a prior exception which was recorded in the CDB for this reassembly.
3:0	Exception code. Codes are explained in the *PAB Expected Exception Priority and Encoding* table.

Table 7 PAB Expected Exception Priority and Encoding

Encoding	Exception Definition	Meaning
0	No exception	The reassembly and the reassembly packet is a good packet.
1	Head Drop	Low priority input task queue reached its discard threshold.
2	Task Error	PAB received a task with the error bit is set. The PAB drops received tasks with the task error bit set.
3	Buffer Management	Input task receive queue reached its buffer management threshold.
4	Sticky Discard	PAB received a sticky discard command for this reassembly ID.
5	Sequence Check Error	PAB detected a sequence error for this reassembly ID.
6	Reassembly Corrupted	This is a special condition when an enqueue occurs with the write offset less than the minimumEnqueueOffset. Special precautions are required when a partial reassembly is transmitted with data left in the reassembly. For example, if there are 128 bytes in a reassembly and a Transmit command is called with the length information of 100 bytes, the minimumEnqueueOffset value for the next enqueue command is 100. Any attempt to write to a byte location less than 100 is prevented and sets this bit. The following conditions also set the reassembly corrupted bit. ■ Enqueue or Transmit with {Length, LengthBitOffset} parameters AND ■ {WriteOffset, DestBitOffset} parameters greater than or equal to ((64K-1) * 8) If any of the parameters have specially interpreted default values, the interpreted value is used. This means if Length and LengthBitOffset are 0, the PAB uses the current reassembly length as the {Length, LengthBitOffset } value.
8	Enqueue and Memory Allocation Error	The system is out of memory.
9	Timeout	A particular reassembly timed out.
10	Sizeout	A particular reassembly exceeded its maximum size.
11	Timer Start Failure	A PAB request sent to the timer engine failed. **NOTE** Timer Start Failure has a higher priority than exceptions encoded 1 through 10
12-15	Reserved	–

When the PAB reports an exception, only one exception is reported. If more than one exception occurs, PAB reports the highest priority exception.The Timer Start Failure, code 11, is the highest priority exception reported, followed by exceptions 1-10 in that order.

Table 8 CDB State Encoding

Bit(s)	Field
31	Active
30:27	Spare
26:24	Priority
23	Spare
22	Sequence Check Enable
21	Discard On Timeout
20	Timer Stopped
19	Spare
18:16	Bit Length
15:0	Length (bytes). This represents the highest enqueued byte.

PAB Task Output Parameters with the GetStatus command

If a task is transmitted because of an explicit GetStatus command, the PAB outputs a task with no packet data, but with PAB output parameters in specific formats, depending on which of the three commands you use.

The following tables show the individual formats of the GetStatus output for each command.

Table 9 GetStatus Format – Select 0 (Default, GetStatus_reasmState)

Field	Size	Description
Status	1B	Status byte. See definitions in the *Status Byte Encoding* table.
Reassembly Index	3B	Reassembly index specified by the task
CDB State	4B	Same format as in the *CDB State Encoding* table.
Sequence Number	1B	CDB state for this reassembly
Reserved	1 bit	–
Timer Generation Field	3 bits	CDB state for this reassembly
Reserved	2 bits	–
Timestamp	18 bits	CDB state for this reassembly
Reserved	4B	–

Table 10 GetStatus Format – Select 1 (Memory Usage, GetStatus_reasmPrioMem)

Field	Size	Description
Status	1B	Status byte. See definitions in the *Status Byte Encoding* table.
Reassembly Index	3B	Reassembly index specified by the task
Spare	3 Bits	–
Priority Memory Usage	29 Bits	PAB memory usage (in 256-byte units) for the priority of the indicated reassembly
Spare	3 bits	–
Total Memory Usage	29 bits	PAB memory usage (in 256-byte units) total for all priorities
Reserved	4B	–

Table 11 GetStatus Format – Select 2 (Memory Usage GetStatus_prioMem)

Field	Size	Description
Spare	35 bits	This format does not refer to a reassembly or CDB location. There is no reassembly/CDB-specific state to report.
Priority Memory Usage	29 bits	PAB memory usage (in 256-byte units) for the priority specified by the PAB input task parameter
Spare	3 bits	–
Total Memory Usage	29 bits	PAB memory usage (in 256-byte units) total for all priorities
Reserved	4B	–

PAB Commands

PAB commands are passed to the PAB as part of the input task parameter. The first 8 bits of the PAB input task parameter are the operator, and the 5 Most Significant Bits of the operator are called the command. The remaining three bits are sticky bits used for inter-engine communication.

The command field specifies the PAB engine instruction. There are only 14 valid instructions which can be passed through the command field. Do not pass any other pattern other than these valid 14 instructions. The following table shows the instruction name, its binary codes, and a synopsis of its functionality.

Table 12 PAB Commands

Binary Pattern	Command	Function
00000	GetStatus_reasmState	Returns PAB status to the client.
00001	GetStatus_reasmPrioMem	Returns PAB status to the client.
00010	GetStatus_prioMem	Returns PAB status to the client.
00100	Discard	Discards data from the indicated reassembly, starting at the beginning of the reassembly, for the specified length. If all of the data is discarded, then the reassembly is deleted.
00101	Cleanup	Discards all reassembly data, initialize CDB state. Used only when the state of the assembly is unknown, and you want to clear it.
01000	Transmit	Transmits from the specified offset for the specified length, retains reassembly packet data, and sends pointers to the original reassembly packet data in the output task.
01010	TransmitCopy	Same as Transmit command, but duplicates the reassembly packet data in memory, and send the pointers to the duplicate data in the output task.
01100	Transmit with Discard	Same as Transmit command, sends original reassembly pointers in output task, but discards reassembly packet data starting from the beginning of the reassembly, up through the last byte transmitted.
10000	Enqueue	Enqueues input data to reassembly, at the specified offset, for the specified length.
10100	Sticky Discard	Marks the assembly as dirty, freeing all memory and causing all subsequent enqueued data to be dropped until there is a full Transmit with Discard, Enqueue Transmit with Discard, standalone Discard, or Cleanup command.
11000	Enqueue with Transmit	Same as Enqueue command, followed by Transmit command, sends original reassembly pointers (updated by Enqueue) in output task.
11010	Enqueue with Transmit Copy	Same as Enqueue with Transmit command, but duplicates packet data in memory and sends the pointers to the duplicate data in the output task
11100	Enqueue with Transmit with Discard	Same as Enqueue with Transmit command, then transmits the entire reassembly and discards source data
11101	Passthrough	Passes input data and pointers to output

NOTE The behavior for all command encodings not described here is undefined.

PAB commands have parameters that may have different meaning depending on the command. The following table lists the parameters, their default values, and their sizes.

Table 13 PAB Command Parameters

Parameter Name	Size	Default Value
ReassemblyIndex	3 Byte	0xFFFFFF
Length	2 Byte	0
StartOffset	1 Byte	0
FirstOffset	1 Byte	None
SizeoutThreshold	2 Byte	0xFFFF
WriteOffset	2 Byte	0xFFFF
TimerValue	3 Byte	0
AdvancedFlags bit 7 - ExeptionNotify	1 Bit	0
AdvancedFlags bit 6 - BMIgnore	1 Bit	0
AdvancedFlags bit 5 - DiscardOnTimeout	1 Bit	0
AdvancedFlags bit 4 - DiscardOnSizeout	1 Bit	0
AdvancedFlags bit 3:2 - TimerControl	2 Bits	0
AdvancedFlags bit 1 - SeqCheckEnable	1 Bit	0
AdvancedFlags bit 0 - BlockSequenceCheck	1 Bit	0
BitFields bit 7:5 - LengthBitOffset	3 Bits	0
BitFields bit 4:2 - DestBitOffset	3 Bits	0
BitFields bit 1:0 - Reserved	3 Bits	0
SeqNumber	1 Byte	0

- Parameters whose meaning is dependent on the PAB command.
- Parameters whose meaning is independent of the PAB command.

Independent PAB Command Parameters

The following sections discuss the command parameters that do not depend on the PAB command. The parameters that do depend upon the PAB command are described as part of the behavior of the specific command.

ReassemblyIndex

This parameter is the 24-bit reassembly ID to which the segment is to be reassembled. The value of the reassembly ID should be within the list of the configured reassembly IDs.

For more information about configuring reassembly IDs, refer to the *AXX2500 Family of Communication Processors Axxia Software Environment (ASE) Reference Manual*. The default value of 0xFFFFFF does not have any special meaning; it is interpreted as the reassembly ID 16777215.

SizeoutThreshold

This parameter specifies the maximum number of bytes for a reassembly and is also the highest byte number location allowed on a particular reassembly. If a reassembly segment Enqueue command attempts to write to an address higher than the SizeoutThreshold, the reassembly terminates and the data is discarded from the reassembly. This means the next segment enqueued with the same reassembly ID starts a new reassembly. The default value of 0xFFFF indicates that no SizeoutThreshold is specified.

The reassembly packet data may be transmitted depending on the information specified in DiscardOnSizeout parameter.

TimerValue

This parameter provides timer information. The 24 bits of time information are calculated as follows:

- Bit 23 sets the unit. When it is 1, the unit is seconds; otherwise it is microseconds.
- Bits 22:16 are a left shift operand. This is a signed value and a negative value means right shifting. If the time unit is seconds, the valid left shifting value must be less than or equal to 10. A left shifting value greater than 10 refers to the maximum delay the timer engine can provide.
- Bits 15:0 are the actual timer value.

The Axxia architecture defines a 32-bit timer delta. However, the PAB only uses a 24-bit parameter. These 24 bits are used as the 24 MSBs of the 32-bit timer delta; it is assumed the eight LSBs are all zero. For example, a PAB input parameter of 0x810001 is interpreted as a timer delta of 0x81000100, which translates to 512 seconds.

The default value of 0 indicates that no timer is used.

ExceptionNotify

The PAB engine can receive the following command types from another engine.

- Commands that direct the PAB engine to create an output task
- Commands that trigger a PAB action, but do not create an output task.

The ExceptionNotify parameter permits the PAB engine to generate an output task upon an exception during processing which otherwise does not generate an output task.

The default value of 0 indicates that no output task is generated when an exception event is detected on a particular reassembly. Otherwise, a task is generated when an exception occurs.

NOTE The ExceptionNotify parameter does *not* control the generation of the output task due to a time out or a reassembly size limit. An example of exception event is an event caused by the buffer management. You should only set this bit to 1 if the engine sequence terminates with an engine which can properly interpret the exception event, that is, the MPP or CPU.

BMIgnore

This parameter controls buffer management checking. The default value of 0 indicates buffer management checking is performed for the incoming task. When set to 1, the buffer management checking is not performed for the incoming task.

DiscardOnTimeout

If this bit is set to 1, when a time out occurs in the middle of the reassembly the PAB does not generate a task to the next engine. When a time out occurs, all of the existing segments are removed from the reassembly, therefore the next segment is treated as the initial segment for the reassembly. The default is 0, which causes the PAB to generate a task when a reassembly times out that sends the time out packet to the next engine.

DiscardOnSizeout

The default value of 0 causes the PAB to generate a task when the number of bytes in a reassembly exceeds the SizeoutThreshold value. The PAB also sends the oversize packet to the next engine. However if 1, the PAB engine does not generate a task for the next engine.

NOTE When a time out or oversized assembly is detected, all of the segments are removed from the reassembly. Therefore, the next segment is treated as the initial segment for the reassembly.

TimerControl

Use the 2-bit TimerControl parameter with the timer value to do the following actions.

- Enable or disable the timer.
- Start the timer based on the initial segment or any designated segment (every, current, last, etc.) to arrive in the reassembly.

The following table describes the various TimerControl settings.

Table 14 TimerControl

TimerControl Value	Action
00	No operation. (Default)
01	Stop Timer The PAB does not time out the reassembly even though the timer value is expired.
10	Start Timer unconditionally when the new segment arrives. The PAB starts the timer with the time out value specified in the TimerValue parameter. Therefore, you can specify a new timer value any time these control bits are 10.
11	Start Timer for the initial segment of the reassembly. Use when the time-out needs to start on the arrival of the initial segment in the reassembly, rather than any segment.

When a reassembly is terminated either by a command, such as Discard, or an exception, such as oversize PDU, the PAB automatically resets the timer. For example, the PAB may start a reassembly and the timer. If the oversize exception occurs because the size of the reassembly exceeds the sizeoutThreshold, the PAB resets the reassembly. In this case, the PAB does not generate an additional output task due to the reassembly time out.

The following two examples illustrate using the TimerControl parameter.

Example 1: In this example, the time-out value is based on the arrival of the initial segment into the reassembly. There are three Enqueues with TimerControl 11 and DiscardOnTimeout = 0, which cause the reassembly to be transmitted when the timer (started by the first enqueue) expires.

1. Segment 1: **Enqueue** with **TimerControl**: 11
2. Segment 2: **Enqueue** with **TimerControl**: 11
3. Segment 3: **Enqueue** with **TimerControl**: 11

Example 2: In this example, the time-out value is based on the arrival of the last segment onto the reassembly. Assuming there are three segments for the reassembly and on the last segment there is a Transmit with Discard command.

1. Segment 1: **Enqueue** with **TimerControl**: 10
2. Segment 2: **Enqueue** with **TimerControl**: 10
3. Segment 3: **Transmit** with Discard with **TimerControl**: 01

NOTE Since the reassembly is terminated with **Discard** command, setting **TimerControl** to 01 is not mandatory.

SeqCheckEnable

Both the SeqCheckEnable and the BlockSequenceCheck control sequence number checking in the PAB. Complete details on how these two parameters work together to control sequencing checking are in the BlockSequenceCheck section. The default value is 0.

BlockSequenceCheck

The combination of the BlockSequenceCheck and SeqCheckEnable parameters, plus the current state of the reassembly controls sequence checking in the PAB. The sequence number update only occurs when the state of the CDB shows that the previous segment had sequence checking enabled.

You can use the combination of the BlockSequenceCheck and SeqCheckEnable to setup a sequence number check for a particular reassembly ID. The following table shows how the combined settings of BlockSequenceCheck and SeqCheckEnable control PAB sequence checking.

Table 15 Sequence Check Control

SeqCheckEnable	BlockSequenceCheck	Update the reassembly CDB with SeqNumber value	Perform sequence check
0	0	No	No
0	1	Yes, except for the following commands: GetStatus PassThrough	No
1	0	Yes, except for the following commands: GetStatus PassThrough	Yes, except for the following commands: Cleanup Passthrough GetStatus
1	1	Yes, except for the following commands: GetStatus PassThrough	No

To initialize the reassembly sequence number, you must enable the BlockSequenceCheck using the SeqCheckEnable and BlockSequenceCheck combination 01 or 11. To run the sequence check for the incoming segment, use the combination 10.

SeqNumber

The PAB checks the value of parameter depending on the values of the BlockSequenceCHeck and SeqCheckEnable parameters, and the state of the reassembly, if the previous command had sequence number checking enabled. The default value is 0.

Sticky

Sticky bits are bit-wise ORed for the Enqueue, Enqueue with Transmit, Enqueue with Transmit Copy, and Enqueue with Transmit with Discard commands.

The PAB engine saves the current sticky bits in the CDB for the reassembly. A new reassembly is initialized with the sticky bits of the initial segment sticky value. The PAB sends a reassembly's sticky bits with its output task(s) as part of the PAB status byte. Software may use sticky bits to pass or accumulate information from input tasks to output tasks. The PAB does nothing to modify or interpret them except as described within the individual command descriptions. The sticky bits stored in the CDB are not affected by the GetStatus and Passthrough commands.

Using the Enqueue Commands

The Enqueue commands add the incoming task data to a reassembly. The incoming task is generally associated with a *packet*, and the data from the packet that is enqueued is a *segment*.

Command parameters permit you to add the incoming task at any point in the reassembly. You can also select a range of data from the packet for enqueuing. The result of reassembling segments is a reassembly packet. The following figure illustrates the segment, Enqueue command, and reassembly packet.

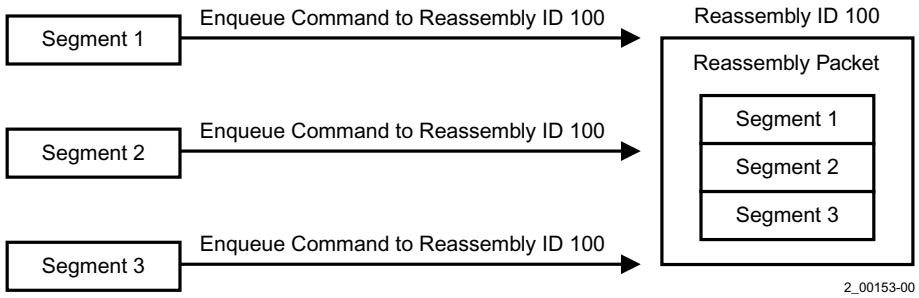

Figure 11 Segment, Enqueue Command, and Reassembly Packet

In the preceding figure, there are three tasks associated with segments 1, 2, and 3. Each task is sent to the PAB with an Enqueue command. The command parameter for the reassembly ID is 100. The result is a part of reassembly 100. The resulting reassembly packet consists of segment 1, 2, and 3.

Enqueue Commands

The PAB engine uses the following Enqueue commands.

- Enqueue
- Enqueue with Transmit
- Enqueue with Transmit Copy
- Enqueue, Transmit, and Discard

Enqueue Command Parameters

You must configure the following parameters for the Enqueue commands.

- Length
- Startoffset
- FirstOffset
- WriteOffset
- LengthBitOffset
- DestBitOffset

There are special conditions that affect how the PAB interprets the Enqueue command parameters. These conditions are described with each of the following individual parameters.

Length

The PAB uses the Length value, along with the StartOffset, FirstOffset, LengthBitOffset, and the actual segment length to determine which part of the segment to enqueue and where to enqueue it. The PAB applies the StartOffset data to every segment except the segment that begins the reassembly. This segment is called the initial segment and, for the initial segment, the PAB uses the FirstOffset instead of the StartOffset.

The default value of 0 means that the PAB should enqueue the entire segment (assuming the LengthBitOffset value is also 0). The following figure illustrates the actual segment size the PAB reassembles as a function of the actual segment length, StartOffset, LengthBitOffset, and Length parameters.

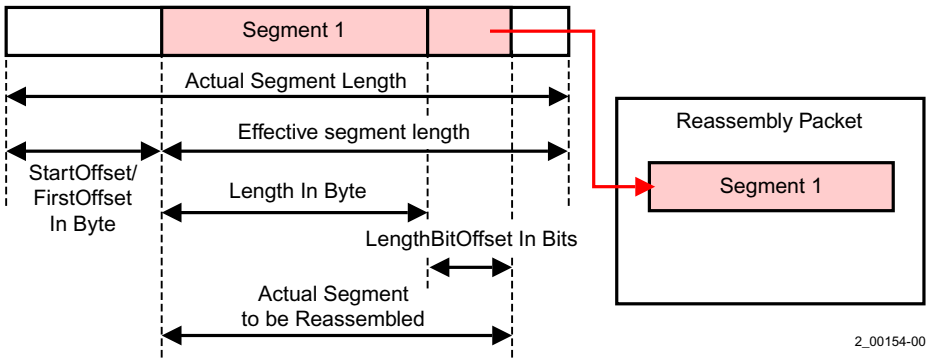

Figure 12 Actual Segment to Be Reassembled

When the sum of Length and LengthBitOffset is 0; then the length to be reassembled is the effective segment length.

StartOffset

StartOffset is the number of bytes to exclude from the original segment. For the initial segment that starts the reassembly, the information in FirstOffset is used and the value in StartOffset is ignored. Generally, the StartOffset value should be less than the actual segment length.

Note the following conditions for the StartOffset parameter.

- When the value of StartOffset is greater than or equal to the actual segment length, the length of the actual segment to be reassembled is 0. The input task is treated as having no data.
- When the value of StartOffset is less than the actual segment length and the sum of StartOffset and Length and LengthBitOffset is greater than the actual segment length, the actual segment to be reassembled is the effective segment length. In the previous *Actual Segment to Be Reassembled* figure, the difference between the actual segment length and StartOffset is called the effective segment length. A sticky interrupt status bit is set to record this unexpected event.

The StartOffset default value is 0.

FirstOffset

The FirstOffset is the number of bytes to be excluded from the original initial segment. The initial segment is the segment that is received by the PAB at the start of the reassembly. Generally the FirstOffset value should be less than the actual segment length.

Note the following conditions for the FirstOffset parameter.

- When FirstOffset is greater than or equal to the actual segment length, the Actual segment to be reassembled is 0. The input task is treated as having no data.
- If FirstOffset is less than the actual segment length and the sum of FirstOffset and Length and LengthBitOffset is greater than the actual segment length, the actual segment to be reassembled is the effective segment length. In the previous *Actual Segment to Be Reassembled* figure, the difference between actual segment length minus FirstOffset is called the effective segment length. A sticky interrupt status bit is set to record this unexpected event

WriteOffset

The default value of 0xFFFF indicates this segment is to be enqueued at the end of the current reassembly. Otherwise, this value specifies the location where the PAB writes the first byte of the segment. Because a segment can be enqueued in any location of the reassembly packet, the following conditions are possible.

- Leaving an empty space in front of a reassembly packet.
- Leaving one or more empty spaces in the middle of a reassembly packet.
- Overwrite a previously enqueued segment.

The following figure illustrates the previous conditions.

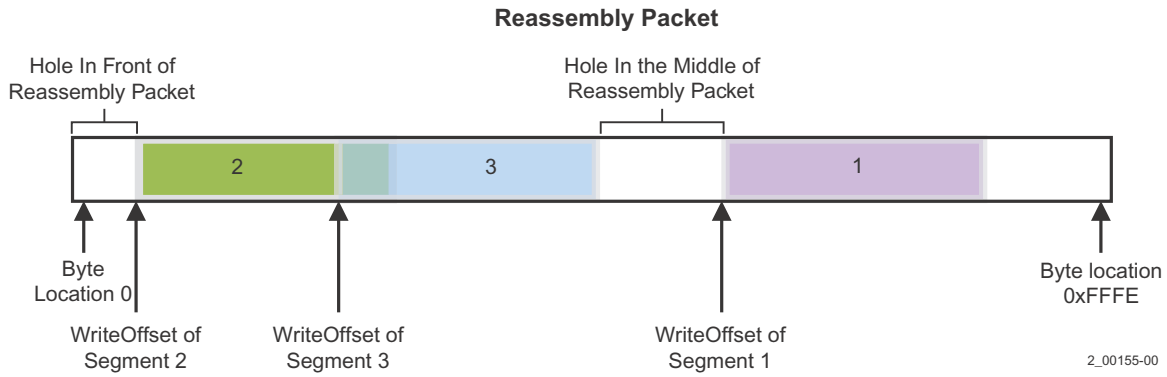

Figure 13 WriteOffset on Enqueue Commands

In the preceding figure, there are three Enqueue commands associated with segment 1 (purple), segment 2 (green), and segment 3 (blue). Segment 1 is the initial segment enqueued to the reassembly packet. Then segments 2 and 3 come to the reassembly. The Segment 1 WriteOffset creates a hole in the reassembly packet. Since the Segment 3 WriteOffset lies within the segment 2 data; data is overwritten. The preceding figure also shows that holes can exist in front *and* in the middle of the reassembly.

For the case when you need to transmit two or more times from a single active reassembly, the PAB restricts the minimum value for the WriteOffset parameter, requiring that the minimum value of WriteOffset be greater than or equal to the MinEnqueueOffset value. The MinEnqueueOffset is a value that points to the location immediately after the last byte of the previously transmitted packet from the reassembly. In other words, if a reassembly is transmitted up through byte n, then the MinEnqueueOffset points to byte n+1. For example, if there are 1000 bytes in a reassembly, then a Transmit command (01000) is called with length information of 500 bytes. In this case, the minimum value of the following WriteOffset must be greater than or equal to 500. This prevents data from a reassembly being referenced by two or more output tasks, which can cause data to be unintentionally overwritten.

LengthBitOffset

The LengthBitOffset command parameter and the Length parameter permits you to specify the length of the segment to be enqueued in bit units. Generally, for byte level reassembly, you should leave this field 0, in which case the length of the segment to be reassembled is specified by the Length command parameter in byte units. The default value is 0.

DestBitOffset

This DestBitOffset command parameter along with the WriteOffset parameter allows you to specify the write location of the segment to be enqueued in bit units. If the WriteOffset parameter is set to 0xFFFF, the information in the DestBitOffset is ignored. Generally, for byte level reassembly, you should leave this field as 0, in which case the write location of the segment to be reassembled is specified by the WriteOffset command parameter in byte units. The default value is 0.

Enqueue with Transmit Command

The Enqueue with Transmit command appends an incoming segment to a particular reassembly ID and then transmits the entire reassembly. To transmit only part of the reassembly, use a Transmit command.

Enqueue with TransmitCopy Command

The Enqueue with TransmitCopy command appends the incoming segment to a particular reassembly ID and then transmits a copy of the entire reassembly to the next engine.

Enqueue with Transmit with Discard Command

The Enqueue with Transmit with Discard command is used to append the incoming segment to a particular reassembly ID, transmit the entire reassembly, and remove it from the reassembly.

Using the Transmit Commands

The Transmit commands transmit a packet from a particular reassembly ID. These commands allow you to select all of a reassembly or a section of the reassembly to be transmitted as the packet sent with the Output Task.

The Transmit commands use the same parameters as the Enqueue commands to define the information about the offset and the length, but instead of defining the range of data for the segment, the Transmit commands use them to define the range of data of the reassembly to be transmitted.

The transmit command does not discard the information from the reassembly. To start a new reassembly, discard the information in the reassembly with the Transmit with Discard command.

The following figure shows four segments in a reassembly. In the figure, the reassembly packet consists of part of segment 2, all of segment 3, and part of segment 4. The WriteOffset, DestBitOffset and the Length parameters identify the part of the reassembly to transmit.

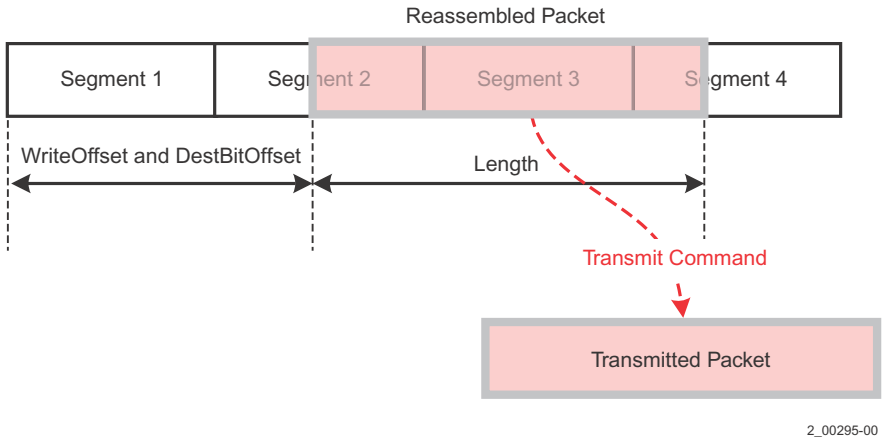

Figure 14 Transmit Command Illustration

Transmit Commands

The Transmit commands include:

- Transmit
- TransmitCopy
- Transmit and Discard

You must configure the following parameters for the Transmit commands.

- Length
- WriteOffset
- LengthBitOffset
- DestBitOffset

Length

The Transmit Length parameter requests the PAB to transmit the entire reassembly from the specified WriteOffset and DestBitOffset (assuming the LengthBitOffset value is 0). When Length is not zero, the PAB uses this value, along with the WriteOffset, DestBitOffset, and LengthBitOffset to determine which part of the reassembly to transmit.

There are two special conditions:

- If the sum of WriteOffset and DestBitOffset and Length and LengthBitOffset is greater than the actual reassembly length, the PAB transmits no data and sets a sticky interrupt status bit to indicate this unexpected event.
- If the sum of Length and LengthBitOffset equals 0, the PAB transmits the entire reassembly starting from the location specified by the sum of WriteOffset and DestBitOffset.

The default value is 0. The following figure illustrates the relationship between these parameters.

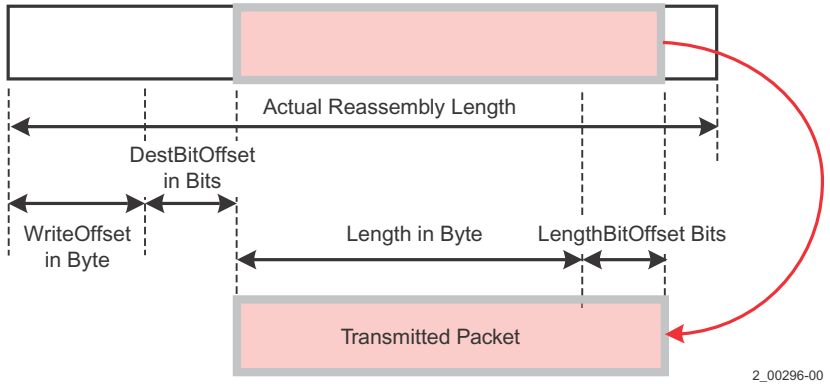

Figure 15 Reassembly Packet

WriteOffset

The default value is 0xFFFF and the PAB engine transmits the reassembly PDU from byte location 0. The WriteOffset and DestBitOffset parameters indicate the location from which the PAB should transmit the reassembly PDU.

The PAB engine can generate one of the following exceptions related to the WriteOffset parameter.

- When the sum of WriteOffset and DestBitOffset is greater than the actual reassembly length, the PAB generates an exception and does *not* transmit data.
- When the sum of the WriteOffset, DestBitOffset, Length, and LengthBitOffset is greater than the actual reassembly length, the PAB generates an exception and does not transmit data.

LengthBitOffset

The LengthBitOffset parameter and the Length parameter permit you to specify the length in bits of the reassembly to be transmitted. Generally, for byte level reassembly, leave this field as 0 so the Length parameter specifies the length of the segment to be reassembled in bytes. The default value is 0.

DestBitOffset

The DestBitOffset parameter and the WriteOffset parameter permit you to specify from where to transmit the reassembly. If the WriteOffset parameter is 0xFFFF, the PAB ignores information in the DestBitOffset parameter. Generally, for byte level reassembly, leave this field as 0 so the PAB can use the location specified by the WriteOffset parameter in bytes as the destination for the packet. The default value is 0.

TransmitCopy Command

When using the Transmit command, if you transmit data more than once from the same reassembly, two or more tasks result that point to the same packet data. If different tasks reference the same packet in memory, there is the potential for an unintentional overwrite of data as the tasks are processed independently by other downstream engines. Also, editing packet data in the PAB after it has been transmitted can cause a similar issue.

To transmit packet data more than once without a potential conflict, the TransmitCopy command duplicates the packet data for the reassembly in memory, and transmits the duplicated packet data from a reassembly ID with a new task. The original data is left in its reassembly, and its state does not change. Similar to the Transmit command, this command includes the information about the offset and the length of the data in the reassembly to be transmitted.

This command is useful when a packet needs to be edited and transmitted multiple times. For example you can send multiple versions of a packet, each one with the header edited slightly differently.

Transmit with Discard Command

The Transmit with Discard command transmits a packet from a particular reassembly ID. The command includes the information about the offset and the length of the reassembly to be transmitted. The Transmit with Discard command is identical to the Transmit command as it sends the original reassembly pointers in an output task. But the Transmit with Discard command discards reassembly packet data, starting from the beginning of the reassembly, up through the last byte transmitted.

NOTE The PAB uses special handling for Discard and Transmit with Discard commands that specify a non-zero LengthBitOffset. If the LengthBitOffset ends at a non-byte boundary, the PAB discards only up to the prior byte boundary–that is, the discard is rounded down. If you do not end LengthBitOffset on a byte boundary, you can get unpredictable results.

Discard Command

The Discard command discards data from the front of a reassembly. The Length and LengthBitOffset parameters determine the number of bytes or bits to be discarded from the reassembly. If these parameters specify the entire reassembly is to be discarded, the PAB terminates reassembly. If the discard command is called and there is data left in the reassembly, the PAB treats the first byte of data which is not discarded as being at offset 0 for any future commands and sets the MinEnqueueOffset parameter to zero.

Application software must track this value for future commands on the reassembly to specify a non-default WriteOffset in the future for this reassembly. If the software does not specify a WriteOffset parameter in the future, there is no need to track this value.

Use the following parameters to configure the Discard command.

- Length
- LengthBitOffset

Length

The default value is 0, which indicates that the entire reassembly is to be discarded. Otherwise, the Length and LengthBitOffset parameters specify the number of bytes and bits the PAB should remove from the beginning of the reassembly.

An exception condition, shown in the *Discard the Data from the Reassembly* figure, applies to the Length parameter when used with the Discard command. If the sum of the Length and LengthBitOffset is greater than the actual reassembly length, the PAB discards the entire reassembly and generates an exception event since it tries to access data that does not exist.

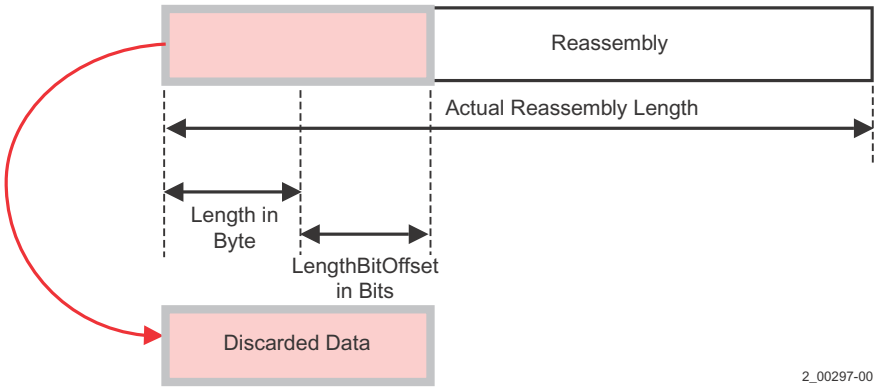

Figure 16 Discard the Data from the Reassembly

LengthBitOffset

The LengthBitOffset and the Length parameter permit you to specify the amount of the reassembly to be discarded in bits, starting from the beginning of the reassembly. Generally, for byte level reassembly, you should leave this field as 0 so the amount of the reassembly the PAB discards is specified by the Length parameter in bytes. The default value is 0.

NOTE The PAB uses special handling for Discard and Transmit with Discard commands that specify a non-zero LengthBitOffset. If the LengthBitOffset ends at a non-byte boundary, the PAB discards only up to the prior byte boundary–that is, the discard is rounded down. If you do not end LengthBitOffset on a byte boundary, you can get unpredictable results.

Sticky Discard Command

The Sticky Discard command marks the packet assembly as corrupted (dirty). This command directs the PAB to free the memory for the reassembly ID and drop all subsequent enqueued data. The PAB maintains the reassembly in this state until the PAB receives a Transmit with Discard, Cleanup, Enqueue with Transmit with Discard, or Discard command.

The Reassembly Index is the only parameter required for this command. Leave all other parameters as default values.

Cleanup Command

The Cleanup command clears a reassembly. Use the Cleanup command only when a particular reassembly index is in an unknown state. This may occur if software is unable to track the state of a reassembly that has experienced an exception. The Cleanup command is equivalent to a Discard command, which discards the entire reassembly, with the following distinctions:

- The PAB never performs Sequence checking.
- The PAB does not check the DiscardOnTimeout parameter value for consistency with the existing DiscardOnTimeout value in the CDB.

The Reassembly Index is the only command parameter required for this command. Leave all other parameters at their default values.

Passthrough Command

The Passthrough command is a special command that does not use any command parameters. The PAB simply passes the task on to the next data path accelerator engine. Therefore, you can leave all command parameters at their default values.

GetStatus_reasmState Command

The GetStatus_reasmState command is a debugging command that lets you check the CDB state for a particular assembly. The ReassemblyIndex is the only command parameter needed for this command. This command does not change any state or status of any reassembly. You can leave all other command parameters at their default values.

GetStatus_reasmPrioMem Command

The GetStatus_reasmPrioMem command lets you check the PAB memory usage for the priority used by a particular reassembly. The command returns information on PAB memory usage (in 256-byte units) for the priority specified by the indicated reassembly, as well as PAB memory usage (in 256-byte units) total for all priorities. The only parameter needed for this command is the ReassemblyIndex. This command does not change the state or status of any reassembly. Leave all other parameters at their default values.

GetStatus_PrioMem Command

Te GetStatus_PrioMem command is a debugging command which allows you to check the memory usage for a specified priority. The command returns information on PAB memory usage (in 256-byte units) for the priority specified by the PAB input task parameter, as well as PAB memory usage (in 256-byte units) total for all priorities. This command does not change the state or status of any reassembly. Leave all command parameters at their default values.

PAB Exception Handling

PAB detects internal hardware inconsistencies and errors, and records them in sticky bits or counters which can be accessed using the configuration.

NOTE The PAB exception handling described in this section applies to all commands *except* GetStatus.

Input tasks are checked for inconsistencies which may have been caused by hardware errors in engines sending tasks to the PAB. Note that such errors can also be software errors. The PAB checks the consistency of the incoming input tasks with respect to Axxia standards. Input task parameters are checked for consistency with respect to PAB expectations. The exceptions are recorded or reported as described previously in PAB command and parameter descriptions.

Hardware Exception Handling

PAB records error exceptions in sticky bits or counters which may be read by the CPU. You must check these bits and counters if you suspect a system malfunction, or you want to determine if the system is operating correctly after processing packets.

Error Containment

The PAB does not retain system state, so you must explicitly clear the state after an error is detected. PAB dynamically allocates memory blocks to hold reassembly data. In a system running without exceptions these blocks are deallocated as the reassemblies are transmitted. If an exception is detected which may interfere with this process, PAB transmits an output task indicating this event and frees up the reassembly memory immediately. Software is then responsible for terminating the reassembly with a Discard or Cleanup. PAB marks the reassembly as corrupted so that a subsequent Transmit command of the reassembly does not result in referencing memory which has been deallocated.

Error Recovery

The PAB does not retain system state, so you must explicitly clear the state after an error is detected. The PAB does not have any error recovery mechanisms. The PAB engine *depends* on the application software to clear the reassembly for the error recovery process.

Error Syndrome

The PAB records error events primarily in sticky bits, not counters. The sticky bits should be explicitly cleared by CPU writes when attempting to identify or isolate faults in a debugging environment.

The only events stored in counters are the exceptions logged in the output tasks. These counters are read-only and are cleared by hardware using hardware reset.

PAB Performance Monitoring Facilities

The PAB maintains performance statistics that a user can read with a software application. The Axxia PAB engine supports the following statistics.

- System memory blocks – Used for reassembly data, per priority and across all priorities. This value is the number of blocks, of all sizes, managed by the MMB.
- System memory bytes – Used for reassembly data, per priority and across all priorities. Use a software application that invokes the GetStatus command to retrieve the System memory bytes.
- Two SMON counter instances – Used to monitor variety of conditions.

Chapter 5: Modular Traffic Manager (MTM) Engine

The MTM engine performs scheduling and buffer management. It supports configurable and programmable scheduling and traffic management algorithms.

Overview

The Modular Traffic Manager (MTM) accelerator engine performs buffer management and scheduling. It supports configurable and programmable scheduling, plus traffic management algorithms. The MTM engine can also manage flow control.

The MTM engine features unicast and enhanced multicast output. It can generate multiple tasks without duplicating the packet data, resulting in effective device resource usage. A single input task can generate multiple new tasks using the multicast feature. The MTM engine processes each of the newly created tasks separately with its own data set. The packet data does not need to be duplicated because each task can point to the same packet data.

Features

The MTM engine has the following feature categories.

- Buffer Management
- Traffic Scheduling and Traffic Shaping
- Flow Control
- Multicast Support

Buffer Management

The MTM engine is equipped with a Traffic Manager, which determines whether to accept or discard a packet and writes the task into the MTM task queue. You can use a script to perform buffer management algorithms and to keep statistics. The Traffic Manager supports up to 64 scripts (each with 1K instructions) to implement different buffer management algorithms.

Use the Traffic Manager to perform the following buffer management functions.

- Select a discard decision for the received task. The Buffer Traffic Manager uses the buffer algorithm programmed in the script to determine whether it enqueues or discards the task.
- Maintain the state of the task by sending the task information for enqueuing.
- Generate backpressure messages to other engines or apply backpressure at any level of the scheduling hierarchy.
- Remove backpressure.

You can program the Buffer Traffic Manager engine policy to keep or discard a packet based on the following criteria.

- Current global and flow traffic thresholds.
- Rate policing algorithms.
- Packet priority.

Traffic Scheduling and Traffic Shaping

The MTM uses up to seven levels of a hierarchy for packet rate shaping and scheduling and contains a Traffic Shaping compute engine. You can use either the supported scheduling or shaping modes or write your own algorithm within a Traffic Shaping compute engine script. You can also use the script for maintaining additional statistics. Similar to the Traffic Manager compute engine, the Traffic Shaping engine can have up to 64 scripts, each with 1 K instructions.

The MTM scheduler supports the byte-accurate, dual-leaky-bucket algorithm based shaping with Smoothed Deficit Weighted Round Robin (SDWRR), Deficit Weighted Round Robin (DWRR), and strict priority scheduling of eligible queues and hardware schedulers. It also supports peak and sustained rate shaping or peak and minimum rate shaping.

Flow Control

The MTM engine can receive and generate flow control with backpressure signals received from and sent to other engines.

The MTM engine uses the backpressure mechanism to perform the following functions.

- Pause scheduling of a queue or scheduler based on external backpressure. For example, backpressure can be applied by the EIOA output ports when they receive PAUSE frames from a client.
- Pause scheduling of a queue or scheduler based on internal script processing.
- Backpressure an engine upstream in the processing path based on internal script processing; for example, to backpressure a flow at the start engine in the processing path.

Both the Traffic Manager and Traffic Shaper compute engines can send backpressure messages on the backpressure ring.

Multicast Support

A single input task can generate multiple new tasks using the multicast feature. Each new task is processed separately with its own data set in the MTM. Because each task can point to the same packet data, the packet data does need to be duplicated.

MTM Processing

MTM does not process the packet data. The MTM can either make a discard decision on the task in a packet or schedule out the tasks to the next engine in the pipeline. For multicasting, new tasks are created and point to the same packet data.

The MTM engine processes tasks using the following steps.

1. The MTM engine receives packet tasks from accelerator engines that need task scheduling. When a task arrives, the Buffer Traffic Management script runs and the MTM makes a decision to keep or discard the task.
2. For expanded unicast or multicast packets, the MTM creates new tasks and uses buffer management for the discard decision.
3. When the MTM engine discards a task, it deletes the task and the packet data from memory.
4. The MTM engine enqueues the tasks it keeps.
5. The MTM engine selects output tasks based on the scheduling structures and algorithms.
6. The MTM engine removes the task from the queue, generates an output task for the packet, and sends it to the next engine in the pipeline.

The MTM scheduler supports byte-accurate dual-rate shaping with Smoothed Deficit Weighted Round Robin (SDWRR), Deficit Weighted Round Robin (DWRR), and strict priority scheduling of eligible queues and schedulers. It supports peak and sustained, or peak and minimum rate shaping.

MTM Functional Description

Scheduling Packets

To achieve the scheduling behavior that you want for multiple traffic flows, you create a scheduling hierarchy of up to seven levels consisting of MTM queues and schedulers:

- An *MTM queue* is an entry point for packets into the scheduling hierarchy. When a packet is enqueued for scheduling, it is assigned to an MTM queue. A queue can be assigned to a scheduler that is at any level of the scheduling hierarchy you create, except the top level, level 0.
- An *MTM scheduler* is a configurable arbiter and rate enforcer for packets assigned to queues and schedulers from one level to the next higher level. Each scheduler can support up to 64,000 entities: queues and schedulers.

Of the seven scheduling hierarchy levels, level 0 is the top level and has a single scheduler, called the *Root Scheduler*. The Root Scheduler can have a maximum of 32 children, all of which must be schedulers. Level 6 can only have queues, since it is the lowest level. Queues can be assigned to levels 2 through 6.

When building a scheduling hierarchy, you begin with level 0 and add the levels you need, up to a total of six, to support your application. The following figure illustrates a sample scheduling hierarchy using six levels.

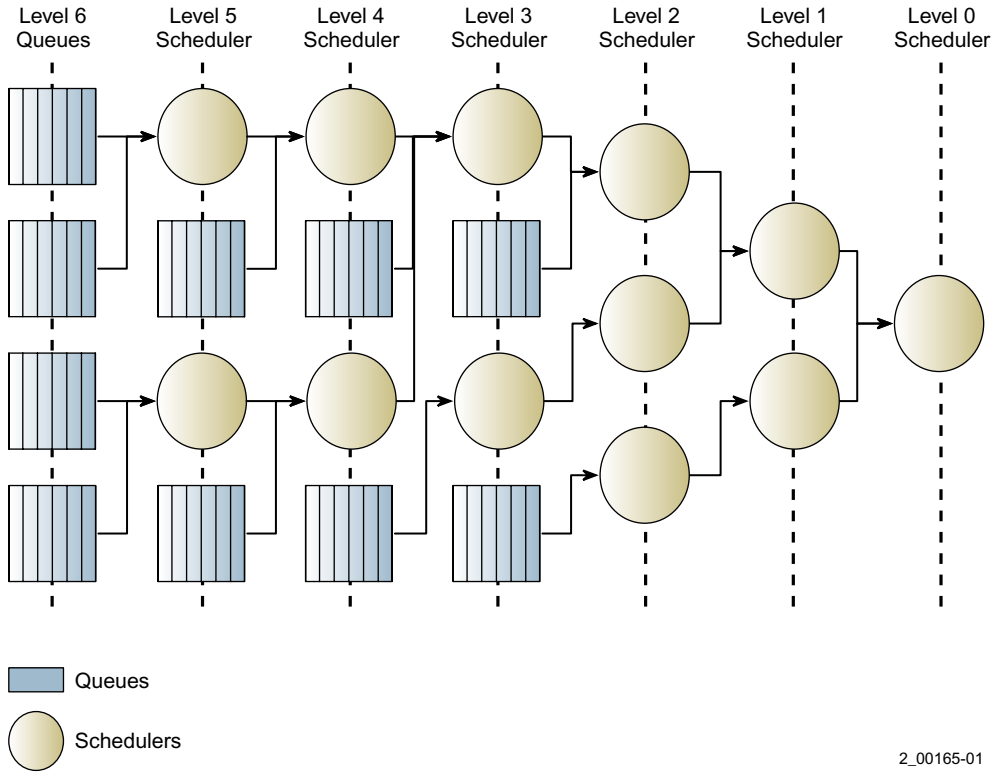

Figure 17 MTM Scheduling Hierarchy Structure

Root Scheduler

The Root Scheduler is the top-level scheduler and is the only scheduler in level 0. It is the last scheduler in any scheduling path, and determines the packet to be sent out of the MTM to the next engine in the engine sequence. The Root Scheduler only supports schedulers; queues cannot be Root Scheduler children.

As the last scheduler, it does not have a scheduling mode, and uses the SDWRR arbitration mode. The Root Scheduler is work conserving. The rate for the Root Scheduler, defined by the sum of the rates of its children, is the maximum output rate for your application.

The Root Scheduler polls its level 1 children for traffic, and when traffic is available, it transmits a packet. As the level 1 children are serviced, they service their level 2 children, and the process continues down to each level of the hierarchy.

Adjusting Packet Size

For both buffer management and scheduling, you can define a packet size adjustment to be used in the MTM calculations. For example, when scheduling the packet in the MTM engine before sending it to the SED engine to add a VLAN tag, you can add four bytes to use in the buffer management and scheduling rate calculations. Similarly, when planning to remove bytes from the packet using the SED engine, you can subtract that amount from the size for scheduling calculations.

The packet size adjustment can be an input task parameter, or from the expander flow table, or from the Buffer Manager script. In addition, each scheduler level has a configured adjustment that enables the adjustment to be different at every level of the hierarchy. The adjustments are cumulative.

Arbitration and Scheduling Modes

Schedulers can support up to 64,000 children, and the children can be a combination of queues and schedulers. The arbitration mode of the scheduler and the scheduling mode of the children controls the behavior of the scheduler and its children.

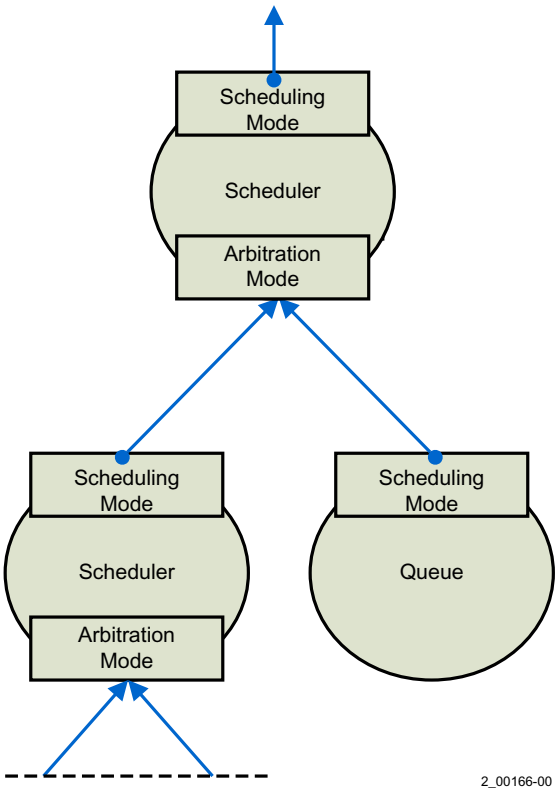

Figure 18 Arbitration and Scheduling Modes

The arbitration mode determines the algorithm used to service the children, while the scheduling mode defines how the child is serviced within the parent scheduler's algorithm.

Each scheduler maintains either one or two lists that track the child queues and schedulers that have traffic. Depending on the arbitration mode you select for the scheduler, the list or lists are managed and serviced differently. The following figure shows the basic structure of a two-list scheduler.

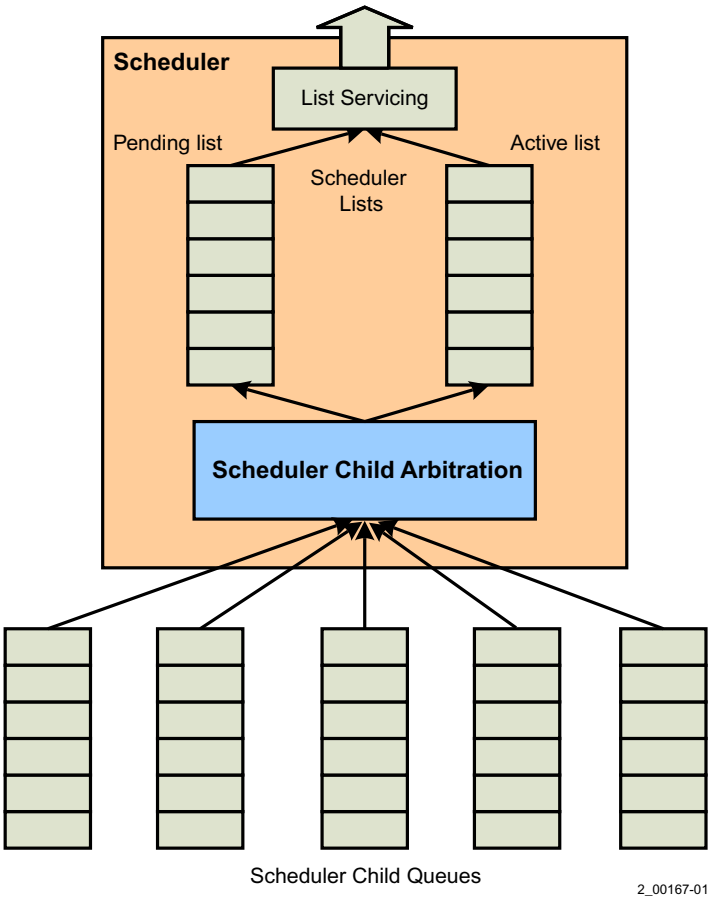

Figure 19 Sample Scheduler Structure

Lists are not used for strict priority, and a single list is used for DWRR. You can define a scheduler's lists to be managed by hardware or by a Traffic Shaper Compute Engine script. For any given path through the hierarchy — starting with a queue and ending with the root scheduler — script-managed arbitration can only be configured for a single scheduler in that path.

You can independently configure each scheduler to support one of the following arbitration modes for its children.

- Smoothed Deficit Weighted Round Robin (SDWRR)
- Script-managed SDWRR
- Deficit Weighted Round Robin (DWRR)
- Script-managed DWRR
- Strict Priority
- Script-managed Strict Priority

Just as the arbitration mode defines how the scheduler arbitrates its children, each child node has a scheduling mode that defines how the child is scheduled within the arbitration mode. For each arbitration mode, the scheduler's children must be assigned a compatible scheduling mode.

The following scheduling modes are used.

- SDWRR – Required for SWDRR arbitration.
- DWRR – Required for DWRR arbitration, defines the priority the child uses.
- Strict Priority 0-3 – Required for strict priority arbitration, defines the priority, set at 0-3, and the list entry point for the child, such as at the top or tail of lists 0 and 1.

Rate Shaping Traffic

The Axxia device employs the token bucket algorithm to enforce rates by shaping traffic. Every queue and scheduler has a programmable dual-rate, dual-bucket mechanism available for controlling its rate and traffic shaping behavior.

The two buckets, called primary and secondary, are configured a with the following parameters.

- Peak rate and sustained rate (shaping mode = PR_SR)
- Peak rate and minimum rate. Peak and minimum rate shaping is not available for strict priority schedulers. (PR_MR)
- Peak rate only (PR)
- No rate shaping (NONE)
- Script (SCRIPT)

Both buckets operate identically for the peak and sustained rates, using the token bucket algorithm. You set the bucket type, rate, and size to define the rate-managing behavior for the bucket.

The minimum rate is specified differently, in bytes of credit per unit.

These following parameters define the behavior of the bucket.

- Bucket type:
  - Peak and sustained rate buckets – Calculate the delays used for traffic shaping rate enforcement.
  - Minimum rate buckets – Determine when a queue or scheduler gets additional round robin weight credit to help it maintain its minimum rate.
- Bucket rate – Sets the ratio of time/bytes that determines how quickly the bucket fills up relative to the packet size.

  For example, a rate of 100 MB/s defines a ratio of 10 ns/byte. Each time a packet is sent from a scheduler, the size of the packet is converted from bytes to time and added to the bucket, and the elapsed time is subtracted from the bucket.
- Bucket size – Defines how much traffic can pass before the bucket rate adjusting action takes place.

  For example, when using peak and sustained rates, the peak bucket size defines how much traffic can burst at the maximum line or media rate before the peak rate shaping is enforced. Similarly, the sustained bucket size defines how much traffic can pass at line and peak rates before the sustained rate shaping is enforced.

  With this behavior, the two bucket sizes define how much traffic a queue or scheduler can send to catch up after a period of no traffic by using the line rate and the peak rate shaping before the sustained rate shaping is enforced.

The following figure showing packets on the wire over time, illustrates the relationships between bucket size, rates, and packet rate shaping, assuming a steady stream of packets after an idle time.

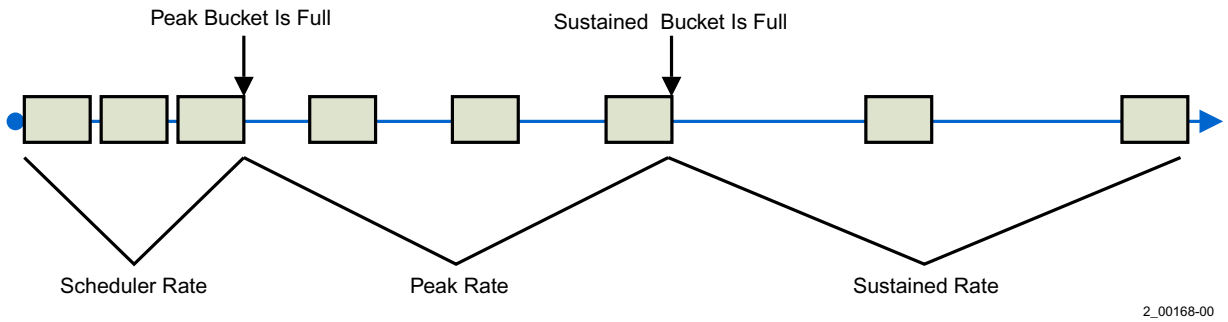

Figure 20 Packet Shaping Using Peak and Sustained Buckets

In this example, after an idle time, the peak and sustained rate buckets are empty. When traffic first arrives, it is sent at the maximum parent scheduler rate until the peak rate bucket fills up, causing the peak rate delays to be applied to packets to maintain the peak rate. Once the sustained rate bucket fills up, the longer sustained rate delays are used, to maintain the sustained rate.

Scheduler Operation

The Axxia architecture supports a scheduling hierarchy of up to seven levels, composed of schedulers and queues. You can define the hierarchy at configuration, and modify it dynamically. A scheduler can have both queues and other schedulers as its children.

Each scheduler services its children at its defined rate, using the servicing algorithm defined by its arbitration mode.

Every time the scheduler sends a packet from one of its children, after the packet is sent, the parent scheduler checks the status of two calculations:

- The child's assigned dual-bucket rates, if any, calculated based on the packet size and the elapsed time since the last packet was sent from the child.

  If the peak or sustained rates have been exceeded, the child is assigned to the timer block for a time equal to the amount of time that the child is ahead in scheduling for the exceeded rate. If it is over both peak and sustained rates, the greater value is used. Once it has been in the timer block for the allotted time, it is returned to its appropriate list:

  — For SDWRR, that is the tail of the pending list.
  — For DWRR, the child is returned to the tail of its list.
  — For strict priority, the child is returned to its assigned location.
- The child's current round robin weight limit, in bytes, after the packet size is subtracted from it.

  Strict priority scheduling mode queues and schedulers do not use the DWRR rate limit except when they are configured for starvation avoidance. When a node has sent traffic equal to or greater than its current weight, it will allow the lower priority siblings to transmit one packet.

  — If the current round robin weight is positive, the queue or scheduler stays at its current spot for DWRR or strict priority scheduling. For SDWRR scheduling, the child is placed at the tail of the active list.
  — If the current round robin weight is negative or zero, the queue or scheduler is placed at the tail of either the pending list for SDWRR or its entry list location for DWRR, and its round robin weight limit is added to its current weight.

These two calculations are performed independently, with the decision to put a child in the timer block taking priority over the round robin limit decision of where to place the child. Placing a child in the timer block does not affect the adding or not adding of the weight to the child's current round robin weight limit.

SDWRR Scheduling

The SDWRR algorithm treats the two scheduler lists as an active list that is being serviced, and a pending list that holds queues and schedulers with traffic that are waiting to be serviced. See the previous *Sample Scheduler Structure* figure.

When a child queue or scheduler receives traffic, it is entered into the pending list. When the active list has completed servicing its queues and schedulers, it becomes the pending list, and the pending list becomes the active list. Once the pending list becomes the active list, the children are serviced in sequence from the head of the list to the tail of the list.

A queue or scheduler at the head of the list transmits a single packet when its turn arrives. Then, if it has additional traffic, it is placed at the tail of the active list. The child remains in the active list until its round robin weight goes negative or zero, or it has exceeded one of its shaping rates.

The minimum SDWRR rate used, measured in bytes, should be equal or greater than the Maximum Transfer Unit (MTU) size for the connection.

NOTE Schedulers configured with SDWRR arbitration require that all children are configured with the SDWRR scheduling mode.

DWRR Scheduling

The DWRR algorithm uses a *single* list for servicing children. The child at the head of the list is always serviced.

When a child is being serviced at the head of the list, it is serviced until its round robin weight is negative or zero, or it has exceeded a shaping rate, then it is put at the end of a list.

Strict Priority Scheduling

A Strict Priority scheduler has up to four children and they are serviced according to priority.

The children are assigned a priority, 0-3, with 0 as the highest priority and 3 as the lowest. The highest priority child traffic is always serviced. You can configure a starvation avoidance mode that permits lower priority traffic to be serviced at defined intervals.

Controlling Arbitration with Scripts

You can control one level of scheduling arbitration in a traffic path through the scheduling hierarchy using a Traffic Shaper script. After the scheduler has scheduled a queue, the task associated with the queue is read out from memory and the traffic shaping script associated with that path in the hierarchy is run. Each path can have only one scheduler associated with a script.

You must always define a Traffic Shaper script for any given path through the scheduling hierarchy. The Traffic Shaper script always runs even if it does not affect the scheduling at any level in the hierarchy. If you choose to use the hardware scheduling at every level of the hierarchy, you can define a null script, or use the Traffic Shaper script to perform additional functions, such as statistics gathering.

The following figure shows an example of a scheduling path and a script-based scheduler.

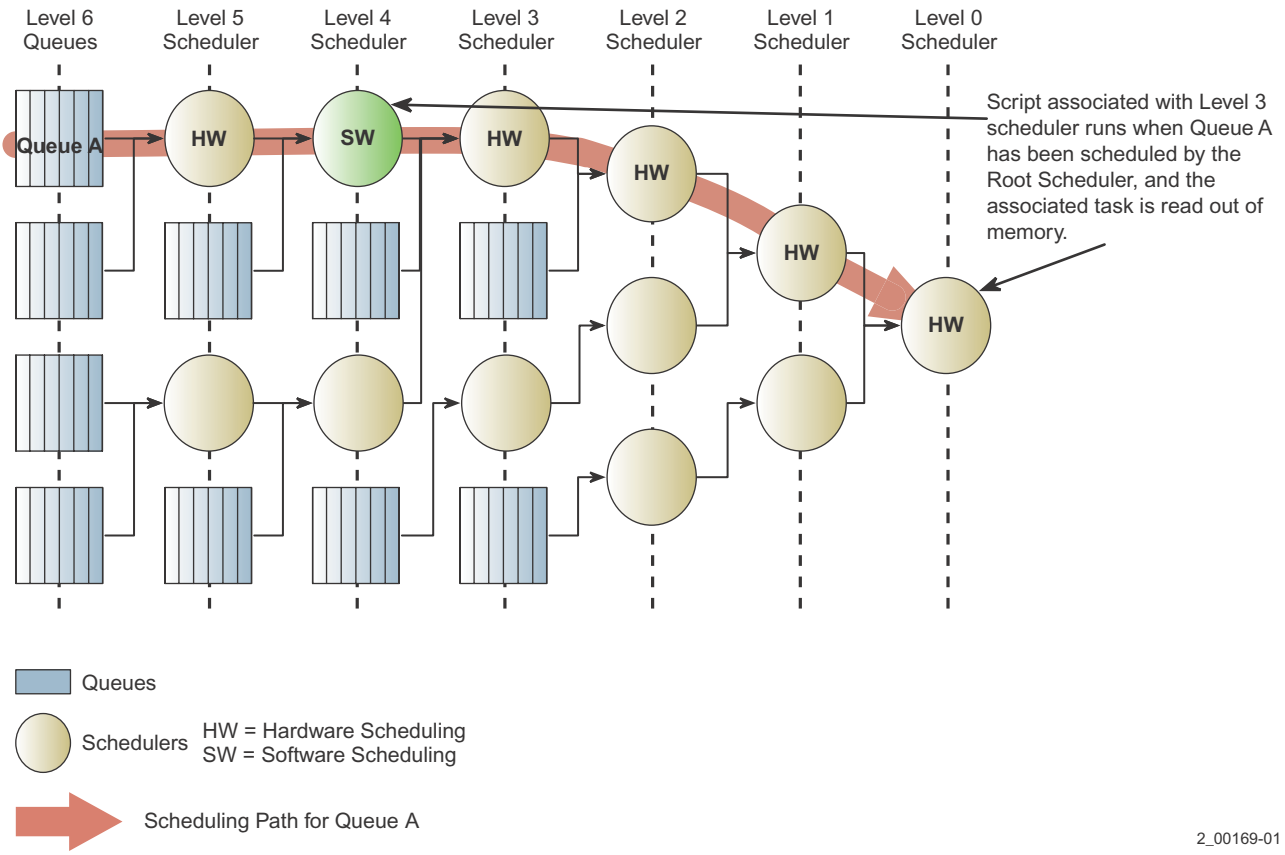

Figure 21 Scheduling Path with Script-based Scheduler

When using a Traffic Shaping script to control arbitration, the structure and default servicing of the lists is still controlled by the arbitration mode selected for the scheduler, such as SDWRR or DWRR. By associating a Traffic Shaper script with a scheduler, you can use the script to control how its child queues and schedulers are entered into the scheduler lists.

The software scheduling model allows a traffic shaper script to control the scheduling at any level of the scheduling hierarchy, for only one level. When the Traffic shaper script is controlling the scheduling at any level, it is responsible for both enforcing the arbitration policy at this level and any shaping of the children of the scheduler.

From the Traffic shaper script, you can control where a child should be placed in the scheduler lists after scheduling:

- For SDWRR arbitration, the script determines if the child is placed in the active or pending list.
- For DWWR arbitration, the script determines if the child is placed at the head or tail of the list.

The script also controls when a child is sent to the timer block and defines the delay for the timer block.

Applying and Receiving Backpressure

The MTM engine can receive and send backpressure for the following purposes:

- To pause scheduling of a queue or scheduler, based on external backpressure. For example, backpressure can be applied by the EIOA output ports when they receive PAUSE frames from their downstream client.
- To pause scheduling of a queue or scheduler based on internal script processing.
- To backpressure an engine earlier in the processing path based on internal script processing. For example, to backpressure a flow at the start engine in the processing path.

The MTM can be backpressured at any level in the scheduler hierarchy. Backpressure can be received from other engines on the Backpressure Ring or can be asserted to the Backpressure Ring from within the MTM from the Buffer Manager or Traffic Shaper compute engines. The backpressure command is sent with the ID of the queue or scheduler, for inband signaling using the backpressure ring.

Backpressure causes the scheduling entity to be removed from the scheduling structure after it has sent the next packet out. When backpressure is turned off, the entity re-enters the scheduling process.

The MTM can also generate messages to be sent on the backpressure bus. The messages can be generated either from the Buffer Manager or Traffic Shaper compute engine scripts.

The MTM also has the ability to generate backpressure messages based on global buffer usage.

Defining Buffer Management Policies

You can program the Buffer Traffic Management Compute Engine policy to keep or discard a packet based on a number of factors, including global and flow traffic threshholds, packet priority, and rate policing algorithms.

Building the Scheduling Hierarchy

You create the scheduling hierarchy using the Axxia Software Environment (ASE). To create the hierarchy, you add a MTMSchedulingLevel element for each level you want in your hierarchy, and add the number of schedulers you want at each level.

The schedulers and queues for a level are kept in an internal scheduling engine table that you define in the ASE. You can also use the RTE to dynamically modify the scheduling hierarchy.

MTM Task Receive Queues and Buffer Management

Task Receive Queues

The MTM supports four task receive queues. There is a low and high priority queue for unicast packets. And there is a low and high priority queues for multicast packets. The separate queues are necessary because multicast packets can take longer to enqueue, since multiple copies are generated from a single task.

The MTM provides a programmable (Strict/WRR) service policy between the Unicast and Multicast task queues. Within a type, the high priority task queues are always serviced before the low priority task queues.

MTM and Expander Engines

In engine sequences, the MTM is represented by two accelerator engines: the MTM engine and the Expander engine:

- The MTM engine is an intermediate engine, used for unicast traffic. Received tasks are passed to the next engine in the engine sequence, with any included parameters for downstream engines.
- The Expander engine is a start or end engine in an engine sequence. Received tasks end at the Expander, and new tasks are generated. The Expander is used for multicast and expanded unicast traffic.

  Each multicast task has a group ID, that has a set of assigned flow IDs that represent the multicast destinations. Each flow ID is used as an index to a namespace table that defines the parameters for a new task generated by the Expander.

Expanded unicast traffic is unicast traffic that uses a flow ID with the Expander to assign parameters and generate a new task, the same mechanism used for multicast traffic.

In engine sequences, the Expander performs the following functions.

— As an end engine, it receives either multicast or expanded unicast tasks. Based on a flow ID, it assigns parameters and creates a new task for each packet to be scheduled.

— As a start engine, it launches the new multicast or expanded unicast tasks into the MTM engine for buffer management and scheduling. It does not, however, send tasks to the MTM engine task receive queues; the resulting tasks of the Expander remain internal to the MTM and go directly into buffer management and then scheduling.

The Expander engine is a subset of the MTM, and, when used as the start engine in a sequence, it must always be followed by the MTM engine.

For example, when developing an application that supports all three traffic types, you might use the following engine sequences.

Table 16 Engine Sequences for Traffic Types

Engine Sequence	Traffic Type
MPP → MTM → EIOA	Unicast
MPP → Expander	Expanded unicast
MPP → Expander	Multicast
Expander → MTM → EIOA	Launching the expanded unicast and multicast traffic task

Both the MTM and the Expander engines share the same configuration information defined under the MTM engine configuration under Engines → MTM in the ASE.

Task Arrival and Enqueuing Arbitration

The MTM and Expander engines receive the following task types.

- Unicast – These tasks, received by the MTM engine, represent a single packet, and the task parameters arrive within the task.
- Multicast – These tasks, received by the Expander engine, represent a single packet that is sent to multiple destinations by generating new tasks for each destination. The task parameters for each of the generated multicast tasks come from a namespace table. Each multicast group defines a set of flow IDs that are used to index the namespace table for the task parameters.
- Expanded unicast – These tasks, received by the Expander engine, represent a single packet, and, like multicast generated tasks, the scheduling task parameters are supplied from the namespace table based on the flow ID.

The tasks that arrive at the MTM are arbitrated in three configurable ways:

- Task receive queues for all tasks. The MTM has four task queues:
  — High and low priority queues for unicast and expanded unicast traffic
  — High and low priority queues for multicast traffic

  To determine the queue for a packet, you map the task priorities to high and low priority queues.
- Arbitration between expanded unicast and multicast tasks for entry into the expansion engine that assigns the parameters to the tasks. This arbitration is between the expanded unicast flow IDs, and the flow IDs assigned to the group IDs of multicast tasks.
- Arbitration between unicast tasks, including both unicast and expanded unicast tasks, and multicast tasks for entry into the buffer management, TM compute engine.

The algorithms used for each of these arbitrations are independently configurable and ca be strict priority or weighted round robin. The following figure shows the arbitration points and the paths to them.

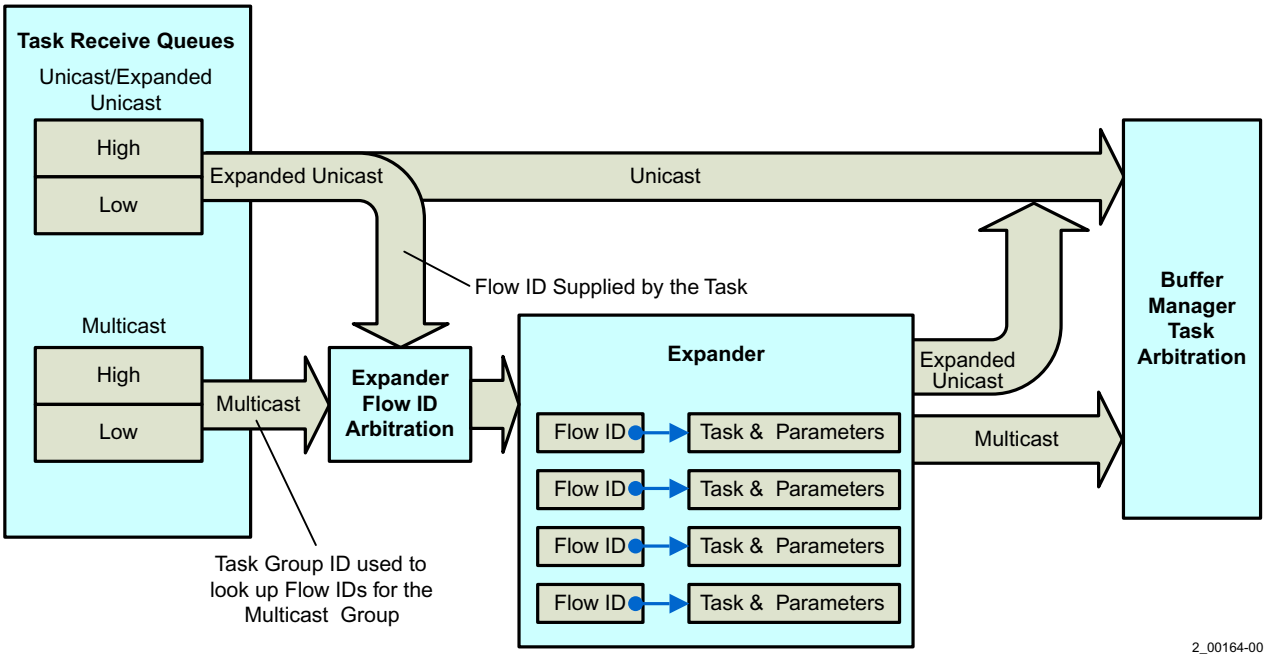

Figure 22 MTM Enqueuing Arbitration

As shown in the previous figure, each of the three task types have different paths to buffer management task arbitration:

- Unicast tasks arrive in the unicast task receive queues, and when they are serviced from their task queue, they enter task arbitration for buffer management.
- Expanded unicast tasks arrive in the unicast task receive queues, and when they are serviced, their flow ID enters the Expander flow ID arbitration. Each expanded unicast flow ID is arbitrated with the multicast flow IDs, generated from the arriving multicast tasks. When they are assigned a task and parameters in the Expander, the newly generated tasks enter task arbitration for buffer management.
- Multicast tasks arrive in the multicast task queues, and when they are serviced, the Expander engine looks up the flow IDs associated with the task group ID. These IDs are entered into the Expander flow ID arbitration. When an ID is selected, it is assigned a task and parameters and the new task enters the Buffer Manager task arbitration.

The unicast and expanded unicast tasks arrive at the Buffer Manager arbitration in the order that they arrived in the task queues. The unicast and the expanded unicast of the same priority are queued in the same task receive queue, therefore their order is maintained.

Queue, Scheduler, and Global Parameters

The MTM supports 32 bytes of parameters per queue and 16 bytes of parameters per scheduler. These parameters are common to both the Buffer Traffic Manager and the Traffic Shaper scripts. Changes made by the Buffer Traffic Manager are visible to the Traffic Shaper compute engine.

When a queue uses a hardware scheduler instead of a script, 16 of the 32 queue parameters are used by the hardware scheduler and are not available for the script in either the Buffer Traffic Manager or the Traffic Shaper.

In the Buffer Traffic Manager compute engine, the 32 bytes of queue parameters are available, in addition to the 16 bytes of parameters of its parent scheduler and another 16 bytes of parameters from another level in the scheduling hierarchy, defined in software. It is possible to configure this level of the hierarchy for each queue.

In the Traffic Shaper compute engine, 16B of parameters from the queue is available along with 16B of parameters from the level at which the Traffic Shaper script is running and 16B of parameters from one level below where the traffic shaper is running. If the traffic shaper is running one level above the queue and so the queue is not using the hardware scheduler, all 32B of queue parameters are available. The other 16B are presented as the parameters from one level below where the traffic shaper is running. Another 16B are available from level in the scheduling hierarchy which is programmable for each queue.

The MTM also supports 32 bytes of Global Parameters. 16 bytes of Global parameters are writeable by the Buffer Manager and 16 bytes are writeable by the traffic shaper. All 32 bytes are readable by both of the compute engines.

Buffer Management

For each task that arrives at the MTM, the Traffic Buffer Manager compute engine runs to determine if the packet should be enqueued or discarded. You can define up to 63 buffer management scripts. Each task is assigned to a queue and each queue is assigned a buffer management script, so the script used for a task depends on its queue.

The decision made by the script is based on the following information provided to the script using the register file.

- The following resource usage items involved.
  - Global buffer threshold
  - Global buffer size
  - Total free memory
  - Packets in the queue
  - Blocks in the queue
  - Memory used by the queue's parent scheduler
  - Memory used by a higher level scheduler, indicated by the queue structure by selecting the level higher than the queue for traffic shaper and buffer management parameters
- The following packet and queue items involved.
  - Packet length
  - Packet length correction – When the size of the packet changes due to processing that occurs after the MTM processing, and you want the packet buffer management decision to be based on the final packet size, you can add or subtract a packet length correction value.
- The following parameters involved.
  - 16 bytes of task parameters
  - 16 or 32 bytes of queue parameters. If the parent scheduler does not use hardware scheduling, 32 bytes are available.
  - 16 bytes of parameters of the queue's parent scheduler
  - 16 bytes of parameters of the selected higher level scheduler
  - 32 bytes of shared queue parameters. These parameters are visible to both the Traffic Shaper and Buffer Traffic Manager scripts, and can be defined for groups of queues, so that when any queue in the group is processed by the Buffer Traffic Manager or the Traffic Shaper, the script can access the 32 bytes of parameters for the group.
  - 16 bytes of Traffic Manager global parameters

Draining Traffic from Queues and Schedulers

APIs handle all of the tasks required to drain a queue or scheduler. The drain operation involves discarding all packets in the queue or scheduler.

Perform the following steps to drain a queue or scheduler.

1. Stop all enqueues to the queue or scheduler you want to drain.

2. Check to determine if the queue or scheduler has any memory associated with it. If it does, you can drain the queue or scheduler in a register. The hardware reads all of the tasks in a queue, or all of the tasks in the queues associated with a scheduler and frees the memory associated with those tasks. When the draining is complete, the hardware signals by setting a bit. The MTM can also optionally assert an interrupt when tit completes draining.

When emptying a queue, the MTM engine reads and empties the queue as fast as it can. If a scheduler is set to be drained, the queues are drained one at a time. The scheduling rate of the queue does not affect how fast the queue is drained. The MTM waits until one packet is scheduled from the queue, so it can be cleared from its parent's scheduling structure, then reads and deletes the other tasks in the queue.

NOTE The MTM engine cannot drain a scheduler if one or more nodes under the hierarchy is backpressured. The MTM engine relies on one schedule event for each queue before draining can proceed.

MTM Input and Output Parameters

MTM Input Parameters

Table 17 MTM Input Parameters

Parameter	Size	Description
command	1B	Identifies the packet type to unicast, expanded unicast, or multicast.
queueId	3B	Queue ID number. This 3-byte value specifies the MTM queue on which to place the unicast packet. This is an index into a internal scheduling table that based on the MTM namespace. All tasks sent with a queueId of 0xFFFFFF are dropped.
packetLenCorrection	1B	Packet length correction. A signed number for correcting the packet length for scheduling. The MTM computes the packet length based on the task and then adds or subtracts the value of this field. Default: 0 (no correction) Range: –128 through 127 (negative values remove and positive values add)
sharedParamIndex	3B	Shared parameter index. You can define 32 bytes of parameters that are shared among a group of queues, and visible to both Buffer Traffic Manager scripts and Traffic Shaper scripts. This field specifies an index into a table that holds the shared parameters. When a task is loaded in the Buffer Traffic Manager compute engine, the 32 bytes of shared parameters, if used, are loaded into it. This index points to an entry in a namespace, and specifies which shared parameter block is used. Default: 0xFFFFFF (no shared parameters are loaded)
parameters	32B	Task parameters available to the Buffer Traffic Manager engine and Traffic Shaper engine. The Buffer Traffic Manager script can modify these parameters before they are passed to the Traffic Shaper engine.

MTM Output Parameters

Table 18 MTM Output Parameters

Parameter	Size	Description
finalOutputTarget	2B	A 16-bit value written by a script, or from the first two bytes of the current_sch_params block or the queue_params block of the register file. To use the queue or scheduler parameters, set this value with the **outputIdSelect** attribute in Namespaces > Namespace > MTMQueues > Queue or Namespaces > MTMSchedulersLevel > Scheduler. For example, the finalOutputTarget could be used to pick the packet's physical Ethernet port.
tsOutput	30B	Up to 30 bytes of output data from the Buffer Traffic Manager or the Traffic Shaper engine.

Setting the Final Output Target

The final output target is the output port destination for the packet. The Output ID value in the Traffic Shaper register file is used as the finalOutputTarget value. This value replaces the first two bytes of the input task parameters when they are passed to the next engine in the output parameters.

You can define the final output target for a packet in one of two ways.

- When you create a queue or scheduler, check the outputIdSelect box in its Namespace definition. The hardware then assigns the queue or scheduler an Output ID, using the first two bytes of the 16 or 32 bytes of queue or 16 bytes of scheduler parameters, respectively. Before the script is run, the hardware writes these two bytes into the Output ID.
- In the Traffic Shaper script, write the Output ID register file value to define the output target for the packet. You can modify the value the hardware wrote, using any parameters available to the script you want, such as input task parameters, or scheduling component parameters.

NOTE You cannot dynamically change the Output ID value of a queue or scheduler. However, you can use a script or the RTE to change the Output ID value used by the queue or scheduler by changing the first two bytes of their parameters.

If a packet's path through the scheduling hierarchy does not include any queues or schedulers with the Output ID value set and the script does not write a value, zero is used for the Output ID.

If the packet's path includes a single queue or scheduler with the Output ID value set, the hardware writes the output ID defined for that queue or scheduler.

If the packet's path includes more than one queue or scheduler with the Output ID value set, the hardware writes the output ID for the scheduling component lowest in the scheduling hierarchy with the Output ID value set. For example, if both a queue and its scheduler have output IDs enabled, the queue's Output ID is written by the hardware and the scheduler's is not.

For an example of using the outputIdSelect, you could configure level two schedulers to map to output ports. By checking these scheduler's outputIdSelect box, you can provide the value that sends all traffic for each of these schedulers to a the specific output port they are scheduling for.

Alternately, you could have the Output ID assigned by an upstream engine, such as the MPP, in the input task parameters, and use the Traffic Shaper script to write that parameter value in the Output ID field. In this case, it would not matter if any queues or schedulers had the outputIdSelect checked, since the script would overwrite the hardware-written value.

Unicast Processing

For unicast tasks, the upstream engine from the MTM engine knows what MTM queue the task needs to be placed in and this is specified as part of the task parameters.

The MTM looks up the queue information and runs the Buffer Traffic Management script for this task. If as a result it decides to accept the task, the task is placed at the tail of the corresponding MTM queue. Otherwise, the task and its corresponding data are discarded. For every unicast task received, the MTM places at most one task in one of its task receive queues.

Expander Input and Output Parameters

Expander Input Parameters

Table 19 Expander Input Parameters

Parameter	Size	Description
command	1B	The packet traffic type: ■ expandedUnicast ■ multicast
flowIdMcastId	3B	For expanded unicast packets, this value is the flow ID. For multicast traffic, this value is the multicast group ID.
virtualPipeline	1B	A Virtual Pipeline instance for the packet destination. The Virtual Pipeline instance must exist under the Expander entry. This value determines which of the 128 Virtual Pipeline instances to use during task generation for expanded unicast or multicast packets
exportParams	16B	Input task parameter Copied to Expander output parameters

Expander Output Parameters

Table 20 Expander Output Parameters

Parameter	Size	Description
importParams	16B	Contains a copy of the exportParams parameter, if provided in the incoming task.

Expanded Unicast Processing

Expanded unicast tasks are tasks that describe unicast packets. The engine sequence which the expanded unicast tasks are sent from terminates in the MTM, and a new task is generated and sent down a new engine sequence that originates in the MTM. The Expander engine launches the new task.

The MTM engine maintains a 64-entry Virtual Pipeline instance table that contains the base address to a region in memory that contains the data required for generating the new task. The incoming task has a flow ID which is an index that points to an entry in this memory.

The MTM engine fetches the data for the new task from system memory and creates a new task, based on the table data and the parameters defined for the Virtual Pipeline instance the packet is assigned to. If the input task provided the 16 bytes of parameters, you can define which of these bytes you want to include in the output task. The format of the resulting task is a regular unicast task. The task is handled just like a unicast task: buffer management followed by either an enqueue into an MTM queue or drop.

Multicast Processing

Multicasting takes an incoming task and duplicates it into a number of expanded unicast tasks. The Expander duplicates the multicast tasks sent to the MTM engine within the MTM.

Multicast tasks describe a set of unicast packets. The origin engine sequence that sends the multicast tasks terminates with the MTM engine. A set of new tasks are generated and sent down new engine sequences that originate in the MTM. The Expander engine launches the new tasks.

An expanded unicast input task points to a flow ID table entry that defines a new unicast task. Multicast input tasks have a multicast group ID that points to a set of flow ID table entries, which defines all of the new unicast tasks generated by the multicast task.

The MTM fetches the data for each new task from memory and creates a new task, based on the table data and the parameters defined for the Virtual Pipeline instance to which the packet is assigned. If the input task provides the 32 bytes of parameters, they are also included in the task. The format of the resulting task is a regular unicast task. The task is handled just like a unicast task: buffer management followed by either an enqueue into an MTM queue or discard.

You can use Virtual Pipeline instance and the multicast group ID input parameters for multicast tasks to select a list of Flow IDs for the outgoing unicast packets. The Virtual Pipeline instance is used as an index into the Multicast Flow Table, to select a set of Flow IDs. The multicast ID indexes the Flow ID set structure to a sequence of expanded unicast Flow IDs that are used when duplicating the packet. The received McastId task parameter points to a list of FlowEntries in the Multicast Flow Table. Each FlowEntry points to an EngineSequence within an egress Virtual Pipeline instance.

The multicast function ensures tasks sent to the same multicast group ID, are put into a given MTM queue in order.

MTM Exception Handling

The MTM engine handles these exceptions with the following actions.

- Tasks with a queue ID = 0xFFFFFF are dropped and the MTM increments a counter.
- Tasks sent to an illegal queue ID are dropped and the MTM increments a counter.
- All MTM queues can have a maximum size (1M packets) and any tasks that exceed this threshold the MTM drops. An MTM counter records the number of tasks dropped because of queue overflow.
- The MTM engine ignores backpressure commands to non-existent scheduler points.
- Task errors received with the input task are passed on in the output task. The MTM does not generate and the Expander does not generate any task errors.
- The MTM engine reports all other error types.

MTM Performance Monitoring Facilities

The MTM engine records statistics for the following performance elements.

- Number of input tasks received
- Number of tasks sent to illegal Queue IDs
- Number of packets (tasks) accepted by Buffer Management
- Number of packets (tasks) dropped by Buffer Management
- Number of tasks sent to another engine
- Number of tasks dropped by the Traffic Shaper
- Amount of memory used by the MTM engine
- Number of multicast packets dropped
- Total number of packets processed
- Total number of packets dropped
- Total number of packets dropped that have a Queue ID = 0xFFFFFF
- Number of packets (tasks) dropped in each of the input task queues
- The compute engine in the buffer manager is capable of maintaining any user-defined statistics in the Queue Structure or in System Memory by using the Shared Parameter blocks.

Chapter 6: Timer Manager (TMGR) Engine

This chapter describes the Time Manager (TMGR) engine that provides flexible timers and timing services to the Axxia CPUs and accelerator engines.

Overview

The Timer Manager (TMGR) engine provides flexible options for tracking Axxia time and using timers that have multiple time formats and various degrees of accuracy and precision. The TMGR engine provides timers and timing services to the CPUs and the accelerator engines.

The Axxia Software Environment (ASE) refers to the Timer Manager as the *Timer*.

Features

The TMGR engine supports timers for the following purposes.

- Scheduling – Timer fires after a defined time.
- Time-out – Timer prompts for an action.
- Process Expiration – Timer indicates a dead or timed-out process.

The TMGR engine has the additional following features.

- The TMGR engine operations include create, delete, start, and stop.
- A task-based timer operation requires very little CPU overhead.
- Task-based timers fire by sending a task (with the ID and tag the requesting engine supplied at timer creation) to the engine that requested the timer.
- Commands to the TMGR engine in a pipeline can be sent by a preceding engine using the task input parameters.
- The CPU subsystem can access the TMGR engine using the RTE APIs. In this case, the CPU does not use the timer input and output task parameter formats.

The TMGR engine supports two input task queues, one high priority and one low priority. You can configure arbitration, weights, and thresholds for these queues in the ASE.

The TMGR engine contains the Axxia Time Accumulator for maintaining the Axxia Time and for generating timestamps. This accumulator can be used for frequency synchronization and time-of-day synchronization. Additionally, the TMGR engine supports interrupt generation to the host processor based on Axxia Time values.

Time Representations

The Axxia architecture stores and manages Axxia Time in a consistent way across all engines. The Axxia architecture uses the following representations of time.

- Axxia Common Time – A 64-bit accumulator that counts microseconds in a fixed-point representation, with the eight least-significant bits representing fractional microseconds.

  This accumulator is the native time that Axxia architecture uses. Each increment of Axxia Time represents 1/256 microseconds. Axxia Common Time therefore counts at an effective rate of 256 MHz.

- Axxia Delta time – Specifies time relative to current time; for example, to start a timer at a specified interval from the current time. Software uses this format most often, because software often applies delta values to timers and to the scheduling of events.
- Axxia Absolute time – An absolute time using a smaller number of bits that gives up precision and range of time (relative to current time). The Axxia architecture uses it internally; it is not visible to application software.

Formats

Axxia Time

The 64-bit Axxia Time representation uses 56 bits to represent microseconds and the least-significant 8 bits to represent fractional microseconds. Each increment of Axxia Time represents 1/256 microseconds. Therefore, Axxia Time counts at a rate of 256 MHz.

The ratio between the core clock and Axxia Time has roughly one part in 2^{58} accuracy. This is helpful for cases where the ratio is set at startup time and not changed. In this way, once the internal and external absolute time are synchronized, the relationship between them will not change rapidly enough that resynchronization is required. The relationship may still slowly change because the ratio between the core clock and Axxia Time is not 100 percent accurate, but only slips one count every several months.

Axxia Delta Time

The Axxia Delta Time format specifies time relative to current time. The following situations use the Delta Time format.

- All timer requests made to the TMGR engine by other accelerator engines.
- FPL timer requests to the MPP hash engine timer structure.
- Traffic shaping within the MTM engine.

The following table describes how the device represents Axxia Delta Time.

Table 21 Axxia Delta Time Representation

Bits	Field	Description
31	Unit	0: Time specified in microseconds 1: Time specified in seconds.
30–24	Scale	Seven-bit signed shift that is applied to the Value field. If Unit=1 (seconds), then Scale field should be less than ten; otherwise, the Value field is treated as a maximum delta time.
23–0	Value	Delta

Delta Time format can be used in shortened forms (16 and 24 bits) as needed by setting the LSBs of the Value field to 0. A 16-bit delta time has bits [15:0] of Value as 0 and uses the Scale to put the resulting 8 bits in the right place. Conversely a 24-bit delta time format has 16 bits of Value and the least-significant 8 bits are 0.

To represent a Delta Time as microseconds, set the upper byte to 0 and the lower 3 bytes specify the value in microseconds. In other words, a value in microseconds can be directly used as a 32-bit value, as long as the value is less than 2^{24}.

To represent a Delta Time in seconds, compute the number of seconds and then set the most significant bit.

The scaling factors either represent time in fixed-point seconds (by choosing seconds as the units and specifying a negative shift), or to keep the width of the math reduced (for example, in an MTM traffic shaping script) by doing all the scheduling interval math at a scaling which keeps the width of the math less regardless of shaping rate.

Axxia Compressed Absolute Time

The Axxia devices uses the Compressed Absolute Time format to store absolute times in timer implementations. This representation gives up precision and range of time (relative to current time) and the Axxia architecture uses it internally only so it is not visible to software.

The format of this time is represented in 18 bits. The 18 bits are a floating point representation of Absolute Time. There is a 14-bit mantissa and a 4-bit exponent.

The Axxia Compressed Absolute Time stores all timestamps within 3 percent of the delta used when the new Absolute Time is generated. Furthermore, the Absolute Time will be correctly interpreted within any bound that is no more than 32 times as long as the timer that was asked for. So if a Delta Time of 1 microseconds was specified, the Compressed Delta Time will correctly map to Absolute Time within 320 microseconds. If this Compressed Delta Time was accessed more than 320 microseconds after it was set, the representation may wrap and the stored Absolute Time would then be interpreted incorrectly. The Axxia compressed format can express delta times up to approximately 38 hours. This represents the maximum delay that can be supported by Axxia hardware timers, used for any purpose.

Accuracy and Precision for Axxia Time

The TMGR engine scales the internal core clock to microseconds using a configurable scaling factor. This scaling mechanism supports high precision fractional scaling (roughly one part in 2^{58}). This scaling factor is the one place in the system where the core clock frequency affects time. All other modules use time in the units of fixed point microseconds.

The basic time unit used in the Axxia architecture is microseconds, but certain applications require more precision than one microsecond. There are a few applications (for example, network time) where the protocol specifies time in nanoseconds. To convert Axxia Time to nanoseconds, the Axxia Time can be multiplied by the factor 1000/256.

Software can adjust the ratio between the core clock and Axxia Time with a single 32-bit update. The update is used to adjust the Axxia Time when receiving timing over packet (ToP) updates or when using an input reference clock. For these cases, the full precision is not needed because Axxia Time is updated periodically (if only because the core clock reference crystal may shift frequency over time).

Timers

The Axxia architecture implements two types of timers: non-cancellable and cancellable.

Non-cancellable Timers

Non-cancellable timers are unstoppable timers that run until completion.

Cancellable Timers

Cancellable timers are timers that can be changed (cancelled or time-out value reset) once they start. The Axxia architecture implements cancellable timers internally using non-cancellable timers along with an additional data structure that holds the updated time when a timer is supposed to expire.

An example of using cancellable timers involves analyzing performance metrics. When a timer is cancelled in the MPP engine, a task (which is automatically deleted by built-in FPL software) still arrives at the MPP at the originally scheduled time-out time. Application software accesses cancellable timers using an index which it supplies with the timer when the timer is created or modified.

Using the TMGR Engine

The TMGR engine provides large-scale (millions of timers) timer services to the CPUs and accelerator engines, plus supports timers for the following purposes.

- Scheduling – A timer that fires after a defined time.
- Time-out – A timer that prompts for an action.
- Process Expiration – A timer that indicates that a process is dead or has timed-out.

Characteristics

The TMGR engine has the following characteristics.

- The TMGR engine can be an intermediate engine in an engine sequence, or it can provide timing services directly through the timer ring. The MPP, MTM, and PAB engines can access the TMGR engine directly via the timer ring. Timer events arriving on the timer ring take precedence over those arriving on the task ring.
- It supports timer create, delete, start, and stop operations. Task-based timer operation requires very little CPU overhead. Task-based timers fire by sending a task to the client accelerator engine. The task carries the ID and tag that the client supplied when the timer was created. The task helps the client quickly determine what semantic action it associated with the timer expiration.
- Efficient support for millions (large-scale) of simultaneously active timers.

Task Input Queue Configuration

The TMGR engine supports two input task queues for high and low priority tasks. You can configure queue arbitration, weights, and thresholds similar to other engines.

Starting TMGR Processing

The TMGR engine starts when it receives a command over the dedicated timer ring or the task interface. The CPU application software uses the `ncp_timer_*` RTE APIs to manage timers over the task interface. As long as there are active timers its internal logic will continuously run to notify clients of timer expirations as they occur.

Task Input Parameters

The following table shows the TMGR engine task input parameters.

Table 22 TMGR Task Input Parameters

Input Field	Size	Description		
command	1B	Timer operation. Bit 0 and 1 of the byte contain the encodings.		
		00	Update timer (Create timer if it does not exist.)	
		01	Delete timer	
		10	Stop timer	
		11	Reserved	
deltaTimeout	4B	Delta Timeout value in standard Delta Time format. This time-out value is the delay from the time the task is serviced. This pseudo floating-point number can be interpreted as follows:		
		Bits	**Field**	**Description**
		31	Unit	0: Microseconds 1: Seconds
		30–24	Scale	7-bit signed shift that is applied to value
		23–0	Value	Delta (mantissa)
		The 16 LSBs [15:0] of the deltaTimeout can also be overloaded to provide an additional two bytes of flow ID. If this is used, adjust the Scale value so the MSB is set to maximize the accuracy.		
tag	8B	Data passed back to the timer creator.		
timerId	3B	Timer identification value.		

Because the CPU subsystem uses the `ncp_timer_*` RTE APIs to manipulate timers, the specifics of the timer input and output task format are usually not relevant to application software. The exceptions are when the CPU is not using the RTE APIs or when an engine accesses the TMGR engine by using the task interface rather than the timer ring interface.

Task Output Parameters

Table 23 TMGR Task Output Parameters

Output Field	Size	Description
firedDeltaLowBytes	2B	Lower two bytes of the **deltaTimeout** parameter. If these indicate flow ID, the input parameter should have the value (mantissa) left justified and Scale adjusted accordingly so the MSB of the mantissa is 1.
firedTag	8B	Metadata associated with this timer during creation.
firedTimerId	3B	Cancellable timer index.

Exception Handling

The TMGR engine supports the standard Axxia architecture mechanisms and methodology for reporting exceptions. The TMGR engine checks for interface and internal errors and registers these error indications along with additional error syndrome information in *sticky* status registers that can be read and cleared through the configuration interface. System software can configure exception reporting hardware to determine which of the error indications will generate interrupts.

These are common timer errors and how the TMGR engine handles the following exceptions.

- Timer set to expire within next time tick – Timer is treated as if it has expired in 0 time and the appropriate timer expiration action takes place.
- Out of memory – For creating a non-cancellable timer, the timer request is returned with the Alloc_failed indication set. If there is no memory for a cancellable timer, then the task is returned with the mem_alloc_failed bit set. When this condition happens, a counter increments.

The TMGR engine does not pass on task errors in engine sequences. When the TMGR engine receives a task in an engine sequence, it is being used as a cancellable timer. When the timer expires, the TMGR engine sends an output task to the next engine in the sequence, without any parameters or flags from the input task. The output task serves as a signal that the timer has expired. The typical use of the TMGR engine in an engine sequence has the requesting engine that sends the task to the TMGR engine also serving as the engine that receives the output task from the TMGR engine. For example, an engine sequence from the CPU to the TMGR engine and then back to the CPU (CPU → TMGR → CPU) permits the CPU to use the TMGR engine as a cancellable timer.

External Timing Reference Interface

Axxia device has the following external timing reference interfaces.

- External Signal Definition
- Interface Configuration

Performance Monitoring Facilities

The TMGR engine can support 10+ million timer events per second using a 400-MHz Axxia device.

Timer Support

Although the Axxia architecture supports a small number of traditional hardware timers in the CPU subsystem (for example, MPIC timers and PowerPC 476FP® timers), it uses a separate TMGR engine to provide timer services to the CPU and to the other accelerator engines. The TMGR engine can provide millions of simultaneously active timers to the CPUs and accelerator engines. Accelerator engines request a time and a corresponding duration from the TMGR engine. When a timer expires, the TMGR engine notifies the requesting accelerator engine or CPU.

The TMGR engine provides timers to requesting accelerator engines using the following mechanisms.

- Timer Ring – A dedicated ring connected to the accelerator engines that use large numbers of timers, including the MPP, the MTM, and the PAB.
- Task Ring – A task is sent to the TMGR engine on the task ring by an accelerator engine that requests a timer. The CPUs often use this mechanism.

Timer Support for Resource Management

The MPP natively accesses the TMGR engine with the timer ring interface using its hash engine. A single hash engine call can atomically allocate a namespace ID, start a timer, and associate both the namespace ID and the timer with a hash key so that they can quickly be found later using that hash key.

For more information on using the MPP hash engine timer, refer to the *Axxia Communication Processor Functional Programming Language (FPL) Programming Guide*.

Similarly, the PAB engine associates a timer with an assembly ID. Under task control, such timers can be *watchdog* timers to recover memory for broken reassemblies or to transmit reassembled frames at a fixed interval.

Timer Support for Scheduling

The MTM accesses the TMGR engine using the timer ring interface and requests timers to implement its traffic shaping functions. For example, when a queue or scheduler is ineligible for scheduling during a certain interval, the MTM can set a timer that starts when the queue or scheduler is once again eligible for scheduling.

Timer Support for General CPU Processing

The CPU subsystem accesses the TMGR engine using the task interface. The TMGR engine delivers time-out events on the specified task queue. When timer events are associated with an application software-specified flow, the specified task queue can be an ordered task queue that enables a fast executive run-to-completion model to include timer events. Because the CPU subsystem manipulates timers by sending and receiving tasks over NCA producer-consumer queues in a cached CPU memory, this requires very little CPU overhead.

Support for External Clock Generation and Synchronization

The Axxia architecture provides an interface which is useful for clock generation and synchronization. The TIME_REF and TIME_SYNC signals are multifunction pins which permit generating output reference signals, and using an input reference.

Generating an Output Reference Clock from Axxia Time

Axxia devices can generate a reference clock output (for example, a T1 or E1 clock) at a tick rate that is derived from Axxia Common Time when you configure the TIME_REF pin as an output. This feature might typically be used when Axxia Common Time is synchronized dynamically from sources such as a timing over packet protocol. The ratio between Axxia Common Time and the output reference clock would typically be set at startup time and would not be expected to change dynamically. The Axxia architecture supports an output clock frequency range from less than 10 kHz up to 1/8 of the core clock speed (or up to 1/64 of the core clock speed for lowest jitter). Edges of the generated output clock have 1/2 of an Axxia core clock tick of jitter.

Using an Input Reference Clock to Synchronize Axxia Time

Use the input reference clock feature when an external device performs global timing recovery. In this case, common time should remain synchronized with the reference clock. This is done by configuring the TIME_REF pin as an input and setting it to capture Axxia Time every configured number of TIME_REF edges. The CPU can then compare the measurement of elapsed Axxia Time with the number of elapsed external clock ticks and adjust the Axxia Common Time generation process to maintain frequency synchronization. The maximum recommended frequency for external time input to an Axxia device is 1/8th of the Axxia core clock frequency (for example, 50 MHz for an Axxia device running at a 400-MHz core clock).

Mapping an Input Strobe to Synchronize Axxia Time with an External Device

When synchronizing the Absolute Time with an external device use the TIME_SYNC pin as an input to capture the Axxia Common Time. For example, external logic could assert the TIME_SYNC strobe. Application software can communicate with the external time reference to note what absolute external time the strobe was asserted. By knowing the Absolute External Time and the Axxia Time that the signal was captured, application software can then apply the time difference to future time computations. Since the frequencies are exactly synchronized, this delta will not change.

Using an Output Strobe to Synchronize Axxia Time with an External Device

This feature uses the TIME_SYNC pin as an output to capture the Axxia Time when the pin is asserted. You can compare the Absolute External Time when the TIME_SYNC pin was asserted with the time captured internally when the pin was asserted. In this way, you can know the delta between the Axxia Time and the External Absolute Time. Again, because the frequencies between the Axxia Time and the external time are synchronized, this delta does not change.

Support of Timing over Packet (ToP) Protocols

When using a packet network to convey timing information, a server sends timing information encapsulated into packets to one or more clients. Based on the time the packets are received and their contents, clients can recover the frequency or absolute time at the server. Two well known protocols which Axxia supports for performing this function are NTP and IEEE 1588.

In general, Axxia applications partition timing over packet functionality between the host processor(s) and the acceleration subsystem. The acceleration subsystem, which can run either the LSI Axxia Development Kit (ADK) software, customer software, or some combination, performs the following operations.

- Receives incoming packets.
- Identifies (using classification) which packets are related to timing over packet protocols.
- Delivers these packets (along with the common time that they arrived) with appropriate QoS treatment by the proper NCA queue to customer application software running on the host processor(s).

Customer application software acts on the received packets, performing the following operations.

- Scales back and forth between Axxia Common Time and the time format used by the timing over packet protocol (for example, TAI for IEEE 1588).
- Sends response or follow-up messages.
- Retrieves information about the common time at which previous packets were sent out.
- Serves as a client, occasionally adjusting the tick rate at which common time advances to better match the frequency at which the time server's clock is advancing (using the appropriate RTE commands).

Using EIOA Engines to Capture Network Time from Packets

The hardware timestamping function of the EIOA engines can be configured on a per Virtual Pipeline instance basis for ingress and egress packets. The hardware timestamping process uses the current Axxia Time value, which is carried as metadata along with the received packet. The metadata timestamp value is made available to downstream engines in the Virtual Pipeline instance as needed.

Similarly, EIOA engines can capture the Axxia Time when a packet of interest (for example, a ToP-related packet) is sent out. There are three 64 x 64 timestamp tables in the following EIOA engines.

- One for EIOA0
- One for EIOA1 (For the XGMAC and the 4 SGMIIs)

To specify that a particular packet or task being sent from the EIOA should trigger a timestamp capture, the application software provides a 1-byte timestampCapture parameter associated with the packet or task. This parameter enables the capture and specifies a -bit table index where the associated captured time entry is held. Application software can then use the `ncp_eioa_tx_ts_get()` function to retrieve the specified stored timestamp from the EIOA. The application software can use this information (perhaps converted to a different time base) to generate subsequent timing over packet control messages.

A basic timing over packet algorithm performs the following functions.

- Captures the Axxia arrival time of packets at the EIOA.
- Compares delta times contained in subsequent ToP packets with the delta time of measured arrival times.
- Uses a low-pass filter algorithm (that you provide or third-party application software) to adjust the frequency of Axxia Time generation as needed to track network time frequency.

The above algorithm can be enhanced to correlate absolute time by performing the following actions.

- Using both ingress and egress hardware timestamps of ToP packets to accurately measure round trip ToP delays
- Using the round-trip delay to compute the one-way delay used to modify the Axxia Time value to yield absolute network time
- Applying the calculated delta time to future time computations

Chapter 7: Stream Editor (SED) Engine

The Stream Editor (SED) engine is an efficient and flexible editor for fixed-function and general-purpose packet editing. For clarity, a packet is defined as a complete PDU, and a fragment is a segment of a PDU.

Overview

The Stream Editor (SED) engine can block-edit packets and is an intermediate accelerator engine in a Virtual Pipeline instance sequence. The SED engine can edit a header, payload, and trailer of a packet, in 128-byte blocks using a Very Large Instruction Word (VLIW) compute engine. The SED engine operates on a 256-byte register file that contains up to 128 bytes of packet data and 128 bytes of parameters.

Features

The SED engine performs the following packet-editing functions.

- Adds or deletes data to the packet head or tail.
- Modifies data anywhere in a packet by using a script (C-language subset), to calculate TTL, updates checksums, and so on.
- Fragments or segments a single packet into multiple packets.
- Performs CRC calculations and timestamp packets.
- Discards packets according to script conditions.

You can use the SED engine to edit packet data with the following techniques.

- Edit only the first block (128 bytes) of the packet. With this approach, you typically edit common packet headers that are less than 128 bytes.
- Edit only the first two blocks (256 bytes) of the packet. You typically use this technique when additional header information (exceeding 128 bytes) needs to be added at the start of the packet (prepended).
- Edit every block of the packet.

The SED can modify a packet trailer, even when it is operating in the first one-block or two-block mode. For many applications, the SED engine efficiently modifies, updates, or adds header information.

For example, the SED engine can perform the following typical operations.

- Prepend layer 2 header.
- Insert one or more VLAN tags.
- Perform Network Address (Port) Translation.
- Swap in a new header (layer 2, layer 3, and layer 4).
- Decrement TTL.
- Update header checksums.
- Perform IPv4 or IPv6 fragmentation.

SED Architecture

The SED architecture consists of the following two components.

- SED Core Logic – Receives the task, loads the compute engine, unloads the compute engine, and sends the task to the next engine in the sequence.
- SED Compute Engine – Runs a C-NP language script to perform programmable packet editing.

These two SED components operate as a three-stage pipeline processor that perform the following functions.

- Receives the task and loads the Compute Engine register file with data and parameters.
- Runs the selected Compute Engine script. The script modifies the data in the register file, and sets predicates for any unload processing.
- Unloads the results of script processing from the register file and sends the task to the next engine in the engine sequence.

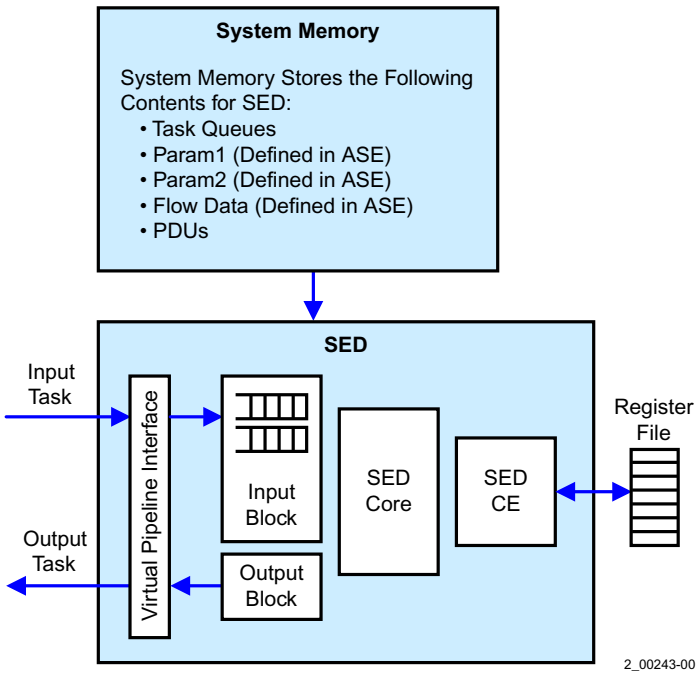

Figure 23 SED Block Diagram

SED Task Receive Queue Configuration

The SED has two task receive queues, for high and low priority. Use the task priority to map tasks to low and high priority packet queues. The SED does not require or support ordered task queues, since the SED must process all blocks or fragments of the packet in order.

The SED Engine receives one task per packet. This means internal task packet data plus all packet pointers/data sizes are guaranteed to contain one and only one packet. The maximum packet in the system is defined as 64 KB. For completeness, when the SED Engine fragments and outputs multiple tasks, these are considered as standalone packets contained within those tasks.

SED Functional Description

The SED edits packets through parameter- and script-based actions. For clarity, a packet is defined as a complete PDU, and a fragment is a segment of a PDU.

SED Capabilities

The SED is a block editor that allows programmable editing of up to 128 bytes of packet data at a time, loaded into its Compute Engine register file with processing parameters. It can be used to edit:

- The first block of a packet only
- The first and second block of a packet only
- Every block of a packet

In any of these three modes, the SED can add padding and trailer bytes to the end of the packet. The script defines the padding data and length, and trailer information as output parameters. The script signals for that information to be written to the end of the packet by setting a predicate flag in the register file of the last packet block processed, whether that is the first, second, or last block of the packet.

SED editing options include:

- Adding up to 255 bytes to the head of the packet, or deleting up to 384 bytes from the head of the packet prior to script processing.
- Deleting up to 765 bytes from the end of a packet prior to script processing
- Adding up to 255 bytes of padding and trailer information to the packet
- Performing modifications to the packet data via the Compute Engine script.
- Segmenting a packet into multiple fragments that are transmitted as independent tasks.

In addition to editing the packet, the SED can send up to 32 bytes of parameters for use by downstream engines.

SED Packet Editing Parameters

The SED engine modifies packet data in 128-byte blocks, using the following parameter types.

- Preprocessing – SED core logic uses these parameters to modify the block before a SED Compute Engine script executes. These parameters define what is loaded into the compute engine.
- Compute Engine processing – SED Compute Engine uses these parameters to modify the block.

SED Parameter Processing

SED processing begins when a packet editing task arrives. The core logic first determines the processing parameters for the packet, merging the stored parameters with the task parameters to identify the processing options for the packet, which include:

- The packet data to be loaded. The parameters defines offsets, called the start and end delta. You can define up to three start and three end delta values using task parameters, parm1 and parm2. When more than one start or end delta is defined, they are added together, respectively, to determine the number of bytes added or deleted from the head or tail of the packet. For example, if three start deltas are defined, 12, 20, and 10, the start delta used is 42 bytes.

  - Use the start delta to add bytes to, or delete bytes from, the head of the packet. The start delta value is a signed number from -128 to 127, with negative numbers defining the number of bytes deleted, and the positive numbers defining the number of bytes added to the head of the packet. You can use up to three start deltas, allowing up to 384 bytes to be deleted from the head of the packet, and up to the hardware maximum of 255 bytes to be added to the head of the packet using multiple summed start deltas.

    Additionally, you can define up to 240 bytes of prepend data in namespace table parameters. If prepend data is defined in the parameters, that data is loaded into the register file ahead of the packet data. Prepend data is inserted before start delta bytes. Up to 255 bytes of data can be prepended to the packet. If the prepend data length, up to 240 bytes, plus the start delta length, up to 255 bytes, exceeds 255 bytes, then the start delta is reduced to fit the 255 byte limit.
  - Use the end delta parameter to define the number of bytes to be deleted from the end of the packet. The end delta cannot be used to add bytes to the end of the packet. The value of the end delta parameter has a maximum of 255. To define a larger end delta, you can define one or two additional namespace end delta parameter values that are added to the task end delta to achieve the 765-byte maximum.
- IP fragmentation information, including fragmentation header length, maximum transfer unit size, and minimum fragment length.
- Name of the compute engine script to run. The SED supports up to 128 scripts.
- 32 bytes of task parameters for script processing. These parameters, also called task inline parameters, arrive with the task and are loaded when the first block is processed, and are stored in the register file and used by the compute engine script for all blocks. This block can include trailer data and padding bytes, if needed, and otherwise can be defined to meet application requirements. In the register file, these bytes start at byte 128.
- Pointers to two optional parameter sets in namespace table memory, Parm1 and Parm2. Optionally, these sets can include:
  - An additional set of the task parameters, with a flag byte indicating which values to use.
  - 16 or 32 bytes of parameters, loaded for each block processed. In the register file, these bytes start at byte 160.
  - Up to 240 bytes of data to be prepended to the packet.

SED Input and Output Parameters

SED Input Parameters

Table 24 SED Engine Input Parameters

Parameter	Size	Description
param1Ptr	3B	Pointer (parm1) to an entry in a SED namespace.
param2Ptr	3B	Pointer (parm2) to an entry in a SED namespace.
startDelta	1B	The number of bytes to add to or delete from the start of the PDU. Possible values are –128 to +127. Negative values remove and positive values add.
endDelta	1B	The number of bytes to delete from the end of the PDU. Range: 0 through 255
script	1B	The SED script. Select a script defined under the **SED > ComputeEngine** element.
fragmentHeaderLen	1B	The number of bytes of the packet to repeat for each fragment.

Parameter	Size	Description
fragmentationSize	2B	The number of bytes in each fragment.
minFragmentLen	2B	The maximum length of a packet that is not fragmented.
inlineData	32B	Task parameters. Use these parameters for application-specific tasks, such as trailer information or pad data.

SED Output Parameters

The SED outputs a single 32-byte parameter. These 32 bytes are placed in the task_param_block parameter in the SED register file.

For more information about SED Output Parameters, refer to the *Axxia Communication Processor Compute Engine Programming Guide*.

SED Parameter Structure

The SED provides great flexibility in how you define and use its packet processing parameters.

Parameter Data Structure

You define SED parameters in the following sections.

- Define standard input task parameters in the engine sequence configuration, supplied from the following sources.
  - Inline task parameters – Outputs from upstream engines in the pipeline or flow table entries.
  - Virtual pipeline merge template task parameters – Static values used for all Virtual Pipeline instance traffic and defined using the Axxia Software Environment (ASE).
- Additional parameters defined in one or two SED namespace entries: Parm1 and Parm2. The parameter index (pointer) for the namespace entries is provided by the task parameters.

The task parameters used in processing can be overwritten or added to parameters defined in the Parm1 or Parm2 SED namespace entries.

Parm1 and Parm2 Namespace Table Parameters

These tables are optional; setting the parameter index for Parm1 (param1Ptr) or Parm2 (param2Ptr) to all ones turns off the index pointer.

These tables can contain the following information.

- Up to 32 bytes of additional script parameters
- Replacements or modifications to the standard task parameters
- Up to 240 bytes of data to prepend to the packet

Table 25 Engine Table Types and Defined Entries

Engine Table Type	Table Entries Can Define
Direct Mode	16 or 32 bytes of additional script parameters
Indirect Mode	A set of task parameters, including start and end delta, script ID, and fragmentation parameters.
	Prepend data
	Additional 32 bytes of script parameters
	A flag byte that indicates which table entry fields are used

Parm1 and Parm2 can be the same or different types.

The maximum additional script parameters available from Parm1 and Parm2 namespace tables is 32 bytes, and with the Task script parameters, make a maximum 64 bytes of script parameters.

If both Parm1 and Parm2 define additional script parameters, either set can be selected, or they can be concatenated or merged to create the 32 bytes that are used.

Indirect Mode Characteristics

The Indirect Mode has the following characteristics.

- Standard task parameters – Can be enabled in the indirect parameters using the flag byte. This byte permits you to select specific parameters to use from the indirect table entry.
- Table Entry Memory – Table entries must be 256 bytes or less, to fit a cache line, so that when using indirect table types, the task parameters and flag byte take up 16 bytes, resulting in the maximum of 240 prepend bytes.
- Prepend and additional 32-bytes of Script Parameters have the following characteristics.
  - If prepend data is not enabled, the table entry may still provide 16 to 32 bytes of additional script parameters, if the size of the table provides the space.
  - If the length of the prepend data is less than 240 bytes, the SED can read up to 32 bytes from the end of 256-byte table row as additional script parameters. The SED starts at the next 16-byte border after the prepend data to read up to 32 bytes, based on the number of bytes available.
  - If both Parm1 and Parm2 define prepend data, Parm1 prepend data is used.

Table 26 SED Indirect Parameter Table Structure

Field	Bytes	Description
Flag Byte	1	Selects the valid fields for the table: Bit 7: Start Delta Byte is valid Bit 6: End Delta Byte is valid Bit 5: Script ID Byte is valid Bit 4: Fragment Header Length is valid Bit 3: Frag Size is valid Bit 2: Reserved (Unused) Bit 1: Min Frag is valid Bit 0: PrePend Data Length is valid
startDelta	1	The number of bytes to add to or delete from the start of the PDU. Possible values are -128 to +127. Negative values remove and positive values add.
endDelta	1	The number of bytes to delete from the end of the PDU. Range: 0 through 255
scriptId	1	Script ID number.
fragmentHeaderLen	1	The number of bytes of the packet to repeat for each fragment.
fragmentationSize	2	The number of bytes in each fragment.
minFragmentLen	2	The maximum length of a packet that will not be fragmented.
Prepend Data Length	1	Parameter data block prepend: 1 byte value specifying how many bytes of the parameter data block should be prepended to the packet instead of being given to the SED script. Currently limited to 255 bytes maximum. (1 to 255 range; 0 means no prepend data)
Reserved Bytes	6	Pad to 16B boundary
Parameter Data Block	n	PrePend Bytes (0 to 255B) + SED Parm data (0 to 32B) Max Case: 1 Flag Byte + 9B Parms + 6B Pad + 255B + 32B = 303B – 256B = 47 B Overage Therefore, PrePend size or SED Parm size or both must account for the 47 byte reduction to a Cache Line Size of 256 Bytes.
Prepend Pad	m	REMAINDER Base 16 (nB) = Pad to 16B boundary. That is, MOD 16 (nB + mB) = 0 where nB = number of bytes in the parameter data block, and mB = the number of bytes in the prepend pad.
SED Parm Data	p	0 to 32 Bytes of Parm Data (starts at 16B boundary) SED Engine determines if parameters exists by computing and comparing the “fetched” block size and sum of bytes thru PrePend Pad.
Total	256	

Additional notes on Indirect parameter tables are listed here.

- Indirect Blocks always start at a 16B boundary.
- Large Indirect Blocks should always start at a 256B Cache line boundary to avoid inefficiencies. Spanning cache lines should always be discouraged.
- PrePend Data always starts at a 16B boundary.
- PrePend is a byte length and PrePend data comes first in Parameter Data Block.
- If PrePend used, then PrePend Pad is used to get to a 16B boundary.
- SED Parm Data, if present, always starts at a 16B boundary.

Merging Script Parameters

Merge the direct, indirect, and standard task parameters in the following ways.

- If start delta or end delta parameters are enabled in an indirect table entry, their values are added to the respective Task parameter start or end delta values. If both Parm1 and Parm2 provide start or end delta values, all three values are added.
- If script ID or Fragmentation parameters are enabled in an indirect table entry, their values replace the Task parameter values. If both Parm1 and Parm2 provide values, Parm1 values are used.

Both Parm1 and Parm2 table entries can provide up to 32 bytes of additional script parameters. Even if both Parm1 and Parm2 are used, only 32 bytes total can be loaded from the two tables, so either Parm1 or Parm2 parameters must be selected, or a combination of bytes can be used as bytes from both sources are merged. The Parm1 parameters are loaded first, then merged with the Parm2 parameters to determine the final 32-byte set.

You define how the two sets of 32-byte parameters merge by setting a byte offset, the Parm2 Offset, a configurable parameter for each script. The Parm2 Offset, a value from 0 to 31, defines where in the 32 byte parameter set that the Parm2 parameters begin.

For example, as shown in the following figure, with a Parm2 offset of 7, bytes 0-6 are the first 7 bytes of Parm1, and Parm2 bytes start at byte 7.

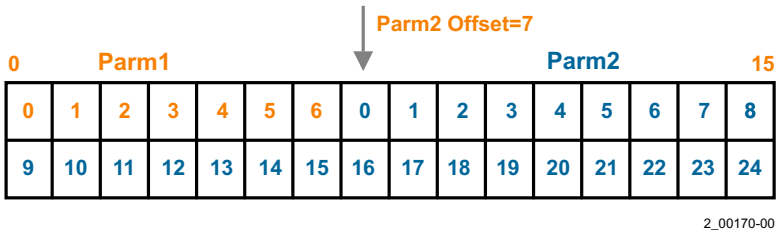

Figure 24 Parm1 and Parm2 32-byte Parameter Merge Example

Setting the Parm2 Offset to 32 selects the Parm1 parameters; setting it to zero selects the Parm2 parameters.

Using Parm1 and Parm2 Parameter Combinations

When Parm1 and Parm2 are both used in the Indirect mode, the start delta and end delta values from both table entries can be enabled, which causes their values to be added to the standard task parameter values for both start and end delta to determine the final value used for processing.

For example, use this feature to implement different headers or trailers for Parm1 and Parm2, to create parameter sets that use multiple combinations of Parm1 and Parm2 parameters without the need to create a unique parameter set for each.

For example, you could define a Layer 2 header set in the Parm1 table, and a Layer 3 header set in Parm2. Using this configuration, you could then use any Layer 2 header with any Layer 3 header, since the length for each header is in the Parm1 and Parm2 table start delta values.

Using SED Parameters

The SED parameters provide flexibility to maximize memory use and to provide for easier reuse for multiple applications.

Depending on application requirements, you can choose to use all or a subset of the parameter options. Here are some sample application requirements and how they can met using the SED parameters.

- *Editing a packet using 32 bytes or less of parameters, without requiring additional prepend data.*

  The standard task parameters, that include 32 bytes of parameters are all that is necessary, without the need for using the Parm1 or Parm2 namespace tables for parameters.
- *Editing a packet using 64 bytes or less of parameters, without requiring additional prepend data.*

  The standard task parameters that include 32 bytes of parameters can be used with Parm1 in the direct mode to provide an additional 32 bytes of parameters for a total of 64 bytes.
- *Editing a packet requiring 64 bytes or less of parameters, and up to 202 bytes of additional prepend data.*

  The standard task parameters, that include 32 bytes of parameters, and Parm1 in Indirect mode, with a maximum of 202 prepend bytes defined, leaving the maximum of 32 additional parameter bytes available in the table entry.
- *Editing a packet requiring 32 bytes or less of parameters, and up to 240 bytes of additional prepend data.*

  The standard task parameters, that include 32 bytes of parameters, and Parm1 in Indirect mode, with a maximum of 240 prepend bytes defined, leaving the maximum of 32 additional parameter bytes available in the table entry.
- *Editing a packet requiring 64 bytes or less of parameters, and up to 240 bytes of additional prepend data.*

  The standard task parameters, that include 32 bytes of parameters, and Parm1 in Indirect mode, with a maximum of 240 prepend bytes defined, and Parm2 in Direct mode, providing 32 bytes of additional parameters.

Exception Handling

Any received taskError is passed on using the outgoing task. The SED does not generate taskErrors.

When the SED receives a task with erroneous packet editing parameters, such as start and end deltas that delete all of the packet, a PDU of zero bytes is sent forward in the output task. However, if PrePend data is specified, it is sent regardless; even if the PDU is completely deleted. The same behavior applies to Pad or Trailer, if specified; the Pad or Trailer bytes are sent at the end of the packet stream.

The SED supports the standard SMON capabilities in the Axxia architecture.

Chapter 8: Security Protocol Processor (SPP) Engine

The Security Protocol Processor (SPP) engine provides decryption and encryption, packet integrity, and anti-replay protection.

Overview

The Security Protocol Processor (SPP) is a multithreaded, pipelined, accelerator engine that provides features needed for security protocol processing, including authentication, encryption, decryption, and editing. The SPP performs encryption and integrity (hashing) checks for packets and performs the necessary security processing. The SPP always functions as an intermediate engine in a pipeline sequence.

The SPP engine provides hardware acceleration offload to both the data and control path. For example, when a host must set up a secure connection, the SPP encrypts and decrypts all traffic in the hardware engine for this secure connection. After the SPP sets up the secure connection, it maintains all context data internally to allow complete hardware operation of future data traffic for this connection.

The SPP engine performs the following operations.

- Accelerates the decryption and encryption of packets for many standard security protocols.
- Provides the following protocol processing security services.
  - Packet integrity authentication.
  - Anti-replay protection.
  - Programmable security protocol processing that includes packet modifications and error checking.

For more information on the SPP, refer to the *Axxia Communication Processor Security Protocol Processor (SPP) Programming Guide*.

Supported Protocols and Features

While the SPP engine is programmable, LSI provides all firmware for it. The SPP engine has the following features.

- IPv4 and IPv6 protocol processing.
- Pseudo-random Number Generator (PRNG) to generate initialization values.
- Support for a packet input length of 0xFFFF to indicate the usage of the entire packet.
- Programmable packet processing for the following items.
  - Security header (for example ESP, AH, and TLS) insertion and removal.
  - Addition and removal of padding.
  - Next packet header retrieval.
  - Extended sequence numbers.
  - MAC (ICV) truncation, appending, and verification.
  - Insertion, replacement, or removal of packet data directly or indirectly related to security processing.
  - Anti-replay checking for window sizes up to 1024 packets.
  - Processing packets that are not byte aligned in memory.
  - Hash result retrieval after decryption.
  - IP header checksum and length modification.
  - Sequence number, hash result, and padding verification.
  - Security context word (record) updates and modification.

The SPP engine supports the following encryption and combined mode algorithms.

- DES (ECB, CBC).
- 3DES (ECB, CBC).
- AES-128 (ECB, CBC, CTR, GCM, CM, CCM).
- AES-192 (ECB, CBC, CTR, CM).
- AES-256 (ECB, CBC, CTR, GCM, CM, CCM).
- ARC4-40, 128, using stateless (RFC 2617) and stateful (RFC 4559) mode.
- Kasumi f8.
- SNOW 3G f8.
- ZUC f8.
- A5/4, A5/3, and A5/1.

The SPP engine supports the following hashing algorithms.

- MD5 (and SSL MAC and HMAC).
- SHA-1 (and SSL MAC and HMAC).
- SHA-224, SHA-256, SHA-384, or SHA-512 (and HMAC).
- AES-GMAC (used with AES-GCM and stand-alone).
- AES-CMAC.
- Kasumi f9.
- SNOW 3G f9.
- ZUC f9.

The SPP supports the following protocols.

- IPsec, Encapsulating Security payload (ESP), and Authentication Header (AH)
- TLS
- SSL
- DTLS
- 3GPP
- SRTP
- MACsec
- IEEE 802.11 WiFi
- IEEE 802.16e WiMAX

Configuring Secure Connections

The host processor is responsible for negotiating the setup of secure connections. To assist this process, there are two hardware-based accelerators that are not part of the SPP. These blocks are available to the host processors via the peripheral bus

- True Random Number Generator (TRNG)
  - — Can be used for initial vector (IV) generation for key management protocols such as the Internet Key Exchange (IKE) protocol.
  - — Compliant with the NIST 800-90 standard.
- Public Key Accelerator
  - — Offloads RSA, Diffie-Hellman, DSA and ECC functions from host processor.
  - — Supports up to 4096-bit exponents.

The following figure shows the interface between the host processor (CPUs) and the TRNG and PKA.

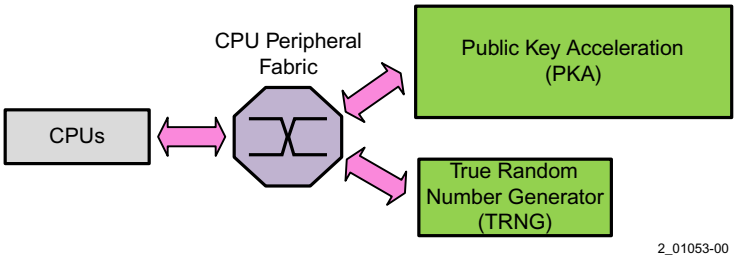

Figure 25 Interface between Host and PKA and TRNG

For more information about the PKA and the TRNG, refer to the *Public Key Accelerator* chapter and the *True Random Number Generator* chapter in the *Axxia AXX2500 Family of Communication Processors Multicore Reference Manual*.

SPP Architecture

The SPP is a multithreaded, pipelined engine consisting of the Multiprotocol engine (MPE) and the Virtual Pipeline Interface. The MPE is the main processing core of the SPP and controls security processing. The Multiprotocol engine has the following components.

- Input Task Manager (ITM) – Receives the task, retrieves the security context information, and starts the pipelined processing.
- SPP Micro-Engines – Microcoded (by LSI) customized security processors which perform all security protocol processing and packet formatting.
- PRNG – Creates random Initialization Vectors (IVs) for insertion into protocol packets.
- Crypto Service Engine (CSE) – A transaction driven cryptography engine that performs all cryptographic and hashing operations. Controlled by the SPP Micro-Engines.
- Security Context L1 Cache – Holds up to 16 security context parameter sets for security processing.

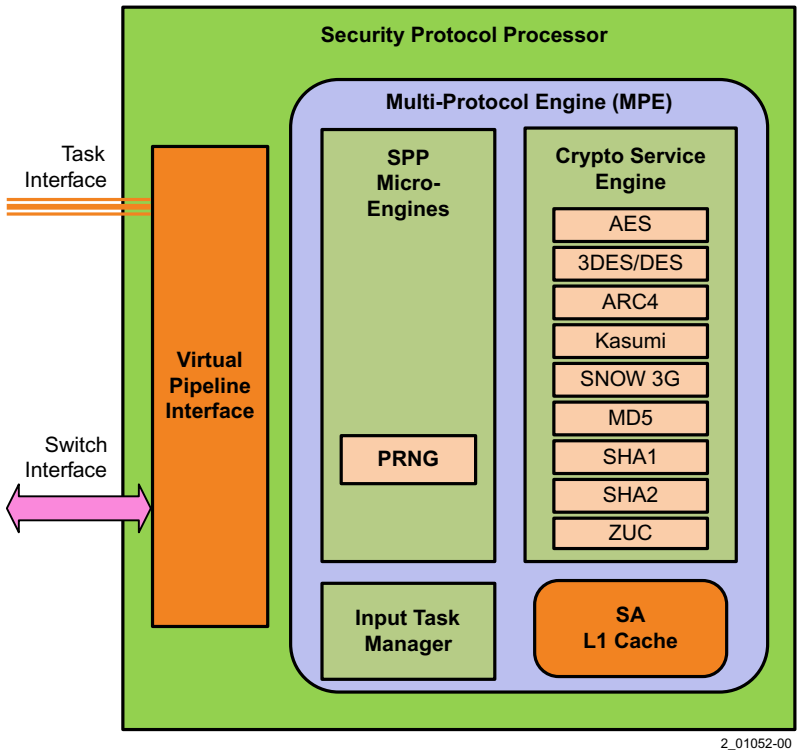

Figure 26 SPP Block Diagram

Virtual Pipeline Model

The SPP is an intermediate engine. The SPP typically receives tasks from the following accelerator engines.

- MPP – Packets may be first processed by the MPP for classification or other processing such as header compression or decompression and then sent to the SPP for security processing.
- PAB – Packets can be reassembled or may be built by merging header and data by the PAB (RLC AM mode) and sent to the SPP for security processing.
- SED – Packet headers may need to be added or packet segmented or fragmented in the SED prior to security processing.
- CPU complex – A CPU can send a packet to the SPP for processing.

The SPP typically sends tasks to the following accelerator engines.

- MPP – After security processing, it is necessary to save statistics for erroneous packets and it may be necessary to reclassify the packet, particularly in the decryption case, or perform other processing such as header decompression or header additions.
- PAB – The PAB can be a retransmission buffer (RLC AM mode) to store and copy the packet after security processing.
- SED – Packets may need editing or additional L2 headers added or to be segmented or fragmented after security processing.
- PIC – Packets may require recalculations of CRC and checksums before being transmitted (for example, UDP checksum for SRTP).
- CPU complex – Packets are sent to the CPU after security processing because the CPU originally sent the packet, or some exception processing is required.

The SPP outputs status indicating errors such as authentication failure, and downstream engines can check this status and take appropriate action such as discarding packets that failed authentication.

NOTE The SPP engine does not receive backpressure messages from downstream engines (ones that receive SPP output tasks) or process the messages.

SPP Virtual Pipeline Operation

The SPP processes packets using up to four threads, depending on the device, in a multi-stage pipeline with the following steps.

1. When a task arrives at the Virtual Pipeline interface block, the SPP engine enqueues the task into one of eight task queues.
2. When a task is at the head of its queue, it is dequeued and sent to the Multiprotocol Engine (MPE). The task includes a pointer to the packet security context. The SPP requests that the security context for the packet be loaded into the L1 cache. The SPP can process up to 16 packets simultaneously.
3. The Input Task Manager (ITM) indicates to the SPP Micro-Engines that the security context cache entry is available and protocol processing starts.
4. The SPP Micro-Engines perform the following protocol processing.
   - Initial value generation, insertion, and removal
   - Packet padding and depadding
   - Security header insertion, modification, verification, and removal
   - Sequence number generation and estimation
   - Anti-replay checking
   - IP header updates (for IPsec) and error checking
5. The SPP Micro-Engines also handle all formatting of data after receiving a packet from the Crypto Service Engine (CSE) for encryption services.
6. The CSE performs necessary ciphering and hashing operations. It can perform multiple operations on a single packet if needed.
7. The security context data structure in the L1 cache is updated with any new values and written back to system memory if no other packet for that security context is currently active in the SPP.
8. The processed packet is written to memory, or to the task if the packet length is less than 96 bytes.
9. The task is forwarded to the next engine in the engine sequence.

Chapter 9: Deep Packet Inspection (DPI) Engine

This chapter defines the Deep Packet Inspection (DPI) engine that performs pattern matching on packet data.

Overview

The Deep Packet Inspection (DPI) engine uses regular expression (RegEx) comparison that performs pattern matching on packet data. It receives the packets from other accelerator engines or CPUs in the pipeline.

Use the DPI engine to scan data for specific preprogrammed patterns and to return results when it finds the patterns. You can scan packet data for the purpose of application recognition, intrusion detection and prevention, virus detection, SPAM filtering, or, if required by an application, accelerated pattern matching.

DPI Features

The DPI engine supports many packet inspection capabilities, including the following features.

- Support for up to 3 Gb/s throughput
- Ability to perform scans across multiple packets
- Support for industry-standard regular expressions
- Support for character class, quantifiers, anchors, subexpressions, and alternation
- Support for up to one million concurrent flows
- Simultaneous evaluation of up to one million rules

Internal Architecture

You must configure the DPI engine, which consists of three processing subengines, before processing any tasks. After the DPI engine is configured, the subengines can receive tasks to scan using a predefined ruleset loaded in system memory.

The following figure shows how the DPI engine fits in with the overall system architecture.

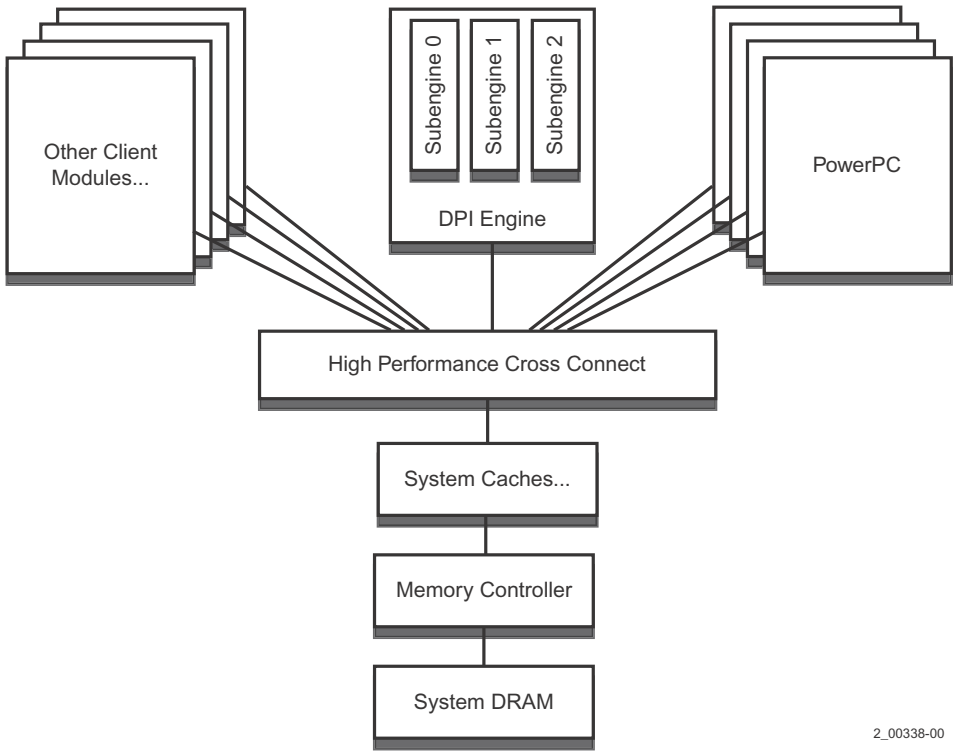

Figure 27 DPI Engine Overview

Task-specific control information is passed to the DPI engine through the task parameters. The input task parameters include control information such as whether the packet will be scanned in cross-packet or non-cross-packet mode, and the position of the packet in a flow (first, intermediate, or last).

Cross-packet scanning is required when a scan consists of the payload inside packets of the same flow. For example, a large file transfer may include many packets, and the file itself needs to be scanned. You can achieve this with cross-packet inspection. Packets from the same stream are identified by earlier engines or the CPU, and a unique identifier is assigned in the input task parameters. A unique identifier is required because packets from different streams can be arbitrarily interleaved and passed to the DPI engine. Any packet with that same identifier is scanned in cross-packet mode until the engine is instructed that the last packet has arrived for this stream. Along with the unique identifier, there are input task parameters for first, intermediate, and last packet in a cross packet inspection. These flags help the engines identify when a cross-packet state (context) must be created, updated or deleted as an individual packet is scanned.

Packets that are scanned in non-cross-packet mode do not need to store any context, and they ignore input task parameters related to cross-packet inspections. Packets scanned in non-cross-packet inspection mode can be scanned for Intrusion Detection and Intrusion Prevention System applications, application recognition, or when the layer 7 payload does not require more than one packet to be transported.

The input task parameters determine the following processing actions.

- The DPI output format, whether the DPI output is token only, or token with packet.
- The number of input packet bytes to skip before processing.

The output token indicates which rule matches the scanned data. Depending on the output parameters, it can also have two variables that describe the position of the match in the scanned data, the beginning and end of the match.

For cross-packet scans, the DPI engine stores context data in memory. Context data is flow-specific packet data that is stored in memory to be rescanned along with the next packet in the flow. There are two types of context data buffers:

- Small context records (SCRs), which can be configured to 32, 64 (default), 128, or 256 bytes at initialization time
- Large context blocks (LCBs), which can be configured to 512 bytes, 1 KB, 2 KB, or 4 KB (default)

Initially, an SCR is automatically allocated. When the amount of context data exceeds the size that can be stored in an SCR, the DPI engine automatically allocates an LCB. The DPI engine allocates LCBs when needed and de-allocates them when the last packet in the flow has been processed.

Compiling Rulesets to Configure the DPI

You can compile rulesets with a stand-alone compiler or with the compilation APIs. After the ruleset compilation, use the load ruleset API to load the resulting binary ruleset. Note that the compiler adds the output format information into the binary ruleset. However, the DPI engine ignores this information. The DPI engine only uses input task parameters for output formatting.

Use the human-readable text ruleset and the compiled binary ruleset to configure the DPI engine.

DPI Task Input Queue Configuration

You must configure the DPI engine before processing any packets. You configure the DPI by using the API to load the compiled ruleset (binary configuration image). The ruleset can be loaded as part of the configuration file you create in the ASE or use an API to dynamically load the compiled ruleset.

The DPI engine receives input tasks., and processes tasks for a given task queue in order. The input task parameters contain the packet-specific control information, such as the position of the packet in a flow (first, intermediate, or last), and the start and end offset.

The input task parameters include one of the following DPI output formats. which indicates whether the DPI output is token only, token with packet, or token ID.

- Token only
- Token with packet
- Token ID with start pointer and end pointer

The following table describes the DPI function blocks and registers.

Table 27 DPI Function Blocks and Registers

Function Block	Register Name	Description
Buffer Management	taskStartFifoHighWaterMark	High water mark for the task start FIFO. Can be programmed to a value from 0 to 7. Default = 5
	taskContentsRecvIntFifoHighWaterMark	High water mark for the task contents receive interface FIFO. Can be programmed to a value from 0 to 15. Default = 13
	taskContentsFiFoHighWaterMark	High water mark for the task content FIFO. Can be programmed to a value from 0 to 7. Default = 5
	taskOutputFifoHighWaterMark0	High water mark for the DPI task output FIFO. Can be programmed to a value from 0 to 31. Default = 29
	taskOutputFifoHighWaterMark1	High water mark for the DPI task output FIFO. Can be programmed to a value from 0 to 31. Default = 29
	taskOutputFifoHighWaterMark2	High water mark for the DPI task output FIFO. Can be programmed to a value from 0 to 31. Default = 29

Function Block	Register Name	Description
Memory Configuration	largeContextRecordSize	Size of each large context block (LCB). Can be set to 512 bytes, 1 KB, 2 KB, or 4 KB (default).
	smallContextRecordsize	Size of each small context record (SCR). Can be programmed to 32, 64 (default), 128 or 256 bytes.
	numLargeContextRecords	Total number of large context records supported. The value ranges from 0 to 2^{24}. Default = 0
	numSmallContextRecords	Total number of small context records supported. The value ranges from 0 to 2^{24}. Default = 0
	rulesetMemorySize	Total memory allocated for rulesets. The value ranges from 0 to 128 MB. Default = 64 MB
	rulesetBaseAddress	Ruleset memory base address. Calculated during system memory allocation. Read only.

Context Across Packets Within a Flow

The DPI engine has access to the system memory where the RegEx ruleset instructions and the cross-packet contexts are stored. Context data (flow-specific task packet data) is stored in memory for rescanning along with the next packet in the flow. Context data includes context length, start condition, start condition stack, and packet data. The DPI engine synchronizes the context reads and writes for a given flow ID.

Initiating DPI Processing

There are three independent threads in the DPI engine. Each thread has its own task parameters and task packet data that control the DPI engine's processing and output. Task parameters also define the output configuration of the DPI engine.

The DPI engine typically receives tasks from the following accelerator engines.

- Modular Pattern Processor (MPP) – The MPP classifier can send packets directly to the DPI engine.
- Packet Assembly Block (PAB) – Packets reordered using IP-Defragmentation, TCP termination, or similar protocols can be sent directly from the PAB to the DPI engine.
- CPU complex – The CPU can invoke the DPI engine directly.

The output target module is also specified in the Virtual Pipeline path. The typical target for tasks leaving the DPI engine is an entity that is capable of doing classification and making decisions.

The DPI engine typically sends tasks to the following accelerator engines.

- CPU complex – The packets have been classified and the CPU processes the answers.
- MPP – Performs additional classification of the DPI engine outputs.

DPI Task Input Format

The DPI engine receives input tasks. Tasks for a given task queue are processed in order. The system arbitrates between the two logical queues and then between the three task queues of the selected logical queue using the configuration and task start interface engine backpressure signals. For task data referenced by descriptors in an input task, the data is fetched from system cache or memory.

The following table shows the input task format for the DPI engine.

Table 28 Input Task Format

Name	Value	Description		
ResultFlags	1B	This byte specifies the format of the output task.		
		Bit	**Field**	**Description**
		3	RetainPacket	If this bit is set, the original packet is forwarded with the DPI engine results appended to the packet. If this bit is clear, the incoming packet is discarded and the outgoing task only contains the DPI engine results.
		2:0	ResultFormat	000: Single result only – These are always 12-byte formats. The output template logic selects which bytes are of interest. 001: Bit vector result. 100: All results, 4-byte result format. See the *Result Format* section for more information. 101: All results, 8-byte result format. 111: All results, 12-byte result format.
StreamFlags	1B	This byte specifies the stream state for this packet. It can indicate that this is the first packet on the stream or the last. If bits 0:2 are clear, the packet is neither first nor last and therefore is assumed to be intermediate. This byte is ignored for non-cross-packet cases.		
		Bit	**Field**	**Description**
		2	End of Stream, discard results	Any stored context is released. No output task is generated.
		1	End of Stream	Causes any stored context to be released.
		0	Start of Stream	For cross-packet cases, this bit indicates that this is the first packet of the context. Any previous state should be discarded. If a large context record is associated with this stream, it should be freed.
StreamId	24b	The index into the context table specifying which cross packet context this packet belongs to. If it is set to all 1s (0xFFFFFF), then no cross packet inspection is being done.		
StartCondition	16b	The start state for the RegEx processor. For cross-packet inspection, this parameter is only relevant when StartOfStream is set.		
MaxOutputs	2B	For the All Results output case, specifies the maximum number of result bytes permitted. Any results in addition to this are dropped.		
StartOffset	2B	Specifies how many initial bytes of the incoming packet to skip before processing. Can be used to skip over headers in the packet.		
EndOffset	1B	Specifies how many bytes from the tail of the packet to not process. Can be used to avoid RegEx processing on packet trailers.		

Load Balancing

The DPI engine has two task receive queues, a high priority queue and a low priority queue. Internally, each of the three processor subengines has a high and low priority queue for incoming tasks. The task receive queue high and low priority tasks are assigned, respectively, to the subengine' high and low priority queues.

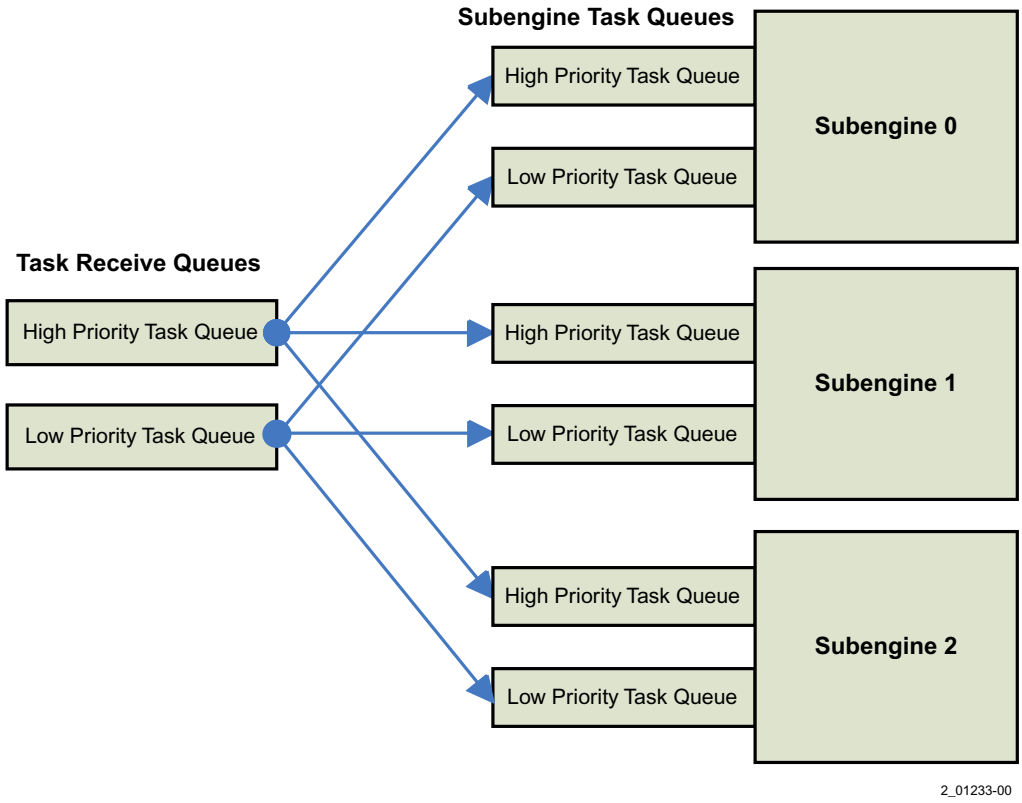

Figure 28 DPI Input Task Queues

As tasks arrive, they are assigned to subengines based on the load balancing algorithm you select. The load balancing of the subengines can be configured to one of the following methods.

- Load balancing by process subengine – DPI uses high and low priority queues as a single resource. New tasks are given to the subengine with the lowest load in both high and low priority queues. For example, a high priority incoming task is sent to the high priority queue of the subengine with lowest total number of tasks in the sum of its high and low priority queues.
- Priority load balancing – DPI sends high priority tasks to the high priority queue with the lowest load and low priority tasks to the low priority queues with the lowest load, without regard to the subengine. For example, a high priority task is assigned to the subengine with the fewest high priority tasks.

An exception to the load balancing algorithms occurs when the incoming task has a task order ID, used to define packets that are part of a flow for cross-packet inspection. If a task arrives with the same task order ID as another task already in a subengine queue, that packet is automatically assigned to the same subengine queue. This ensures that packets are processed in order, so that cross-packet inspection results are based on the correct order. If no tasks with the same task order ID are present in any queue, the selected load balancing algorithm assigns the incoming task to a subengine.

The following table shows how the subengine task queues are mapped to the task priority backpressure signals.

Table 29 Task Queue Backpressure Priority Mapping

Subengine Number	High Priority Queue	Low Priority Queue
0	Priority 0	Priority 3
1	Priority 1	Priority 4
2	Priority 2	Priority 5

The DPI engine does not use priorities 6 and 7 for backpressure for subengine load balancing.

Result Format

The following section describes the format of the results that are returned.

- **First Result or Single Result** – This option enables a special feature in the DPI engine that reports only the first match of a regular expression in the packet.
- **All Results** – If this option is enabled, every match is reported even if it is a subset of, or partially overlaps, another match. If two or more expressions match the same input, all are reported. The following are the different All Results format option actions.
  - **All Results, 12-byte result format** – Returns the token ID, start pointer and end pointer of the matched character's location.
  - **All Results, 8-byte result format** – Suppresses the output of a pointer to the location of the last character (byte) matched by any regular expression.
  - **All Results, 4-byte result format** – Disables reporting of both the start and end pointers associated with each token returned by the DPI engine.
- **Bit vector format** – DPI engine returns a bit vector of token ID matches. Each bit corresponds to a token.

  Instead of outputting the actual token values, you can set up the DPI engine to return a bit vector indicating which token ID values were matched during the scan. A 1 in a bit position indicates that the corresponding token ID value was matched at least once during the scan. A 0 indicates that the corresponding token ID was not matched. The indexing for the bit vector output is 0-based, so the 0th bit of the first Dword corresponds to a token ID value of 0, the first bit corresponds to token ID 1, etc.

  You cannot use the bit vector format if there are more than 128 tokens in the ruleset, because this format can only send up to 128 tokens in the output parameter. Although the ruleset could contain many more than 128 rules, the token values must be in the range 0 to 127. Otherwise, you must use the All Results format.

Start Conditions

The start condition mechanism is a unique and powerful feature that has two complementary uses. First, it allows multiple rulesets to be organized into a single compiled binary image in which each scan job can dynamically select the ruleset to use at run time. Second, you can change the active ruleset by matching certain specified patterns. If you change the active ruleset while in match all mode, the change itself is *greedy*. This means that when a pattern is matched that causes a change in the start condition, the hardware execution immediately switches start conditions without doing any further matching within the text that matched.

The DPI engine supports up to 16K start conditions. If it is initialized with no start condition specified, it defaults to start condition two.

For more information about start conditions and the different scan modes, refer to the *Axxia Communication Processor Deep Packet Inspection Engine Programming Guide*.

Maximum Output

The maximum output value limits the total number of reported items for the input task. When you use the All Results format, the DPI output has token output followed by the retained packet data in the output packet. The maximum size is limited to 64 KB. If there are more token outputs, the output is clipped at 64 KB.

NOTE MaxOutputs is the maximum number of bytes (not results), and MaxOutput results are truncated to the nearest 8-byte boundary.

DPI Task Output Format

After it finishes a packet inspection, the DPI engine sends results through an output task. The system creates an output task based on the information passed in the associated input task. To transmit tasks that require data copies (for example, sending matching data with results) the DPI engine writes the reassembly data to memory and requests new memory blocks as needed. After creating the task and completing any related memory writes, the DPI engine sends the task to the next engine.

There are three different output task formats – FirstResult, VectorResult, and AllResult – as explained in the Input Task format. The DPI output task format depends on the input task ResultFormat configuration. The following table shows the output task formats.

Table 30 Output Task Formats

<table>
<tr><th>Byte</th><th>FirstResult</th><th>VectorResult</th><th>AllResult</th></tr>
<tr><td>0</td><td rowspan="12">FirstResult</td><td rowspan="16">VectorResult</td><td>NumResults</td></tr>
<tr><td>1</td><td rowspan="15">Unused</td></tr>
<tr><td>2</td></tr>
<tr><td>3</td></tr>
<tr><td>4</td></tr>
<tr><td>5</td></tr>
<tr><td>6</td></tr>
<tr><td>7</td></tr>
<tr><td>8</td></tr>
<tr><td>9</td></tr>
<tr><td>10</td></tr>
<tr><td>11</td></tr>
<tr><td>12</td><td rowspan="4">Unused</td></tr>
<tr><td>13</td></tr>
<tr><td>14</td></tr>
<tr><td>15</td></tr>
<tr><td>16</td><td colspan="3">ResultFlags</td></tr>
<tr><td>17</td><td colspan="3">ErrorFlags</td></tr>
</table>

FirstResult and AllResult Formats

The following figure shows the FirstResult and AllResults formats. For more information about the available AllResult format options, see the *Result Format* section.

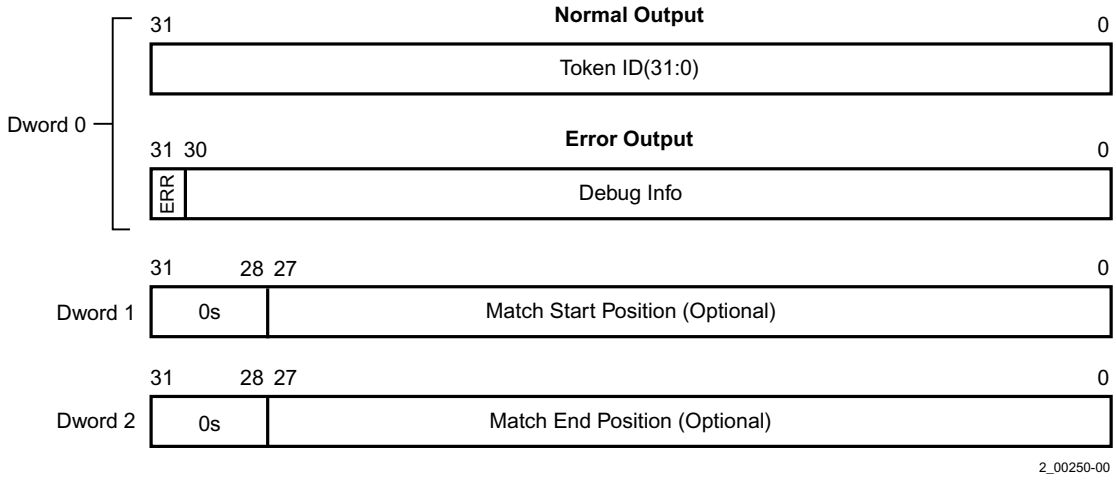

Figure 29 Result Format

Bit 31 of the Dword0 is the Error Token bit, which is set if it is an error token. When an error token is sent, the start pointer is whatever was stored for the start pointer when the error occurred, the end pointer is the current file position when the error occurred, and the Debug Info field contains the current start condition during the error. The following list shows the conditions when the error token is sent.

- A backup error
- An invalid instruction error
- A stack overflow error

In each case, the task that created the error is dropped.

Byte 16 of the ResultFlags output parameter consists of the following fields.

Table 31 ResultFlags Output Parameter - Byte 16

Bit	Field Name	Description
2:0	Result Format	This is a copy of the ResultFormat flag from the input task. 000: Single result only. 001: Bit vector result. 100: All results, 4-byte result format. 101: All results, 8-byte result format. 111: All results, 12-byte result format.
3	Overflow	In the case of All Results, overflow occurs when the number of results is larger than the specified maximum in the input task.
4	Large Context	This bit is set if a large context is associated with this stream.
5	Job Terminate	The job was stopped as the result of a halt instruction in the DPI program.
6	ctx alloc error	A large context was needed, but it could not be allocated.
7	Error	Some other error occurred, as recorded in ErrorFlags. This bit is set if any bit in ErrorFlags is set.

Byte 17 of the ErrorFlag output parameter consists of the following fields.

Table 32 ErrorFlag Output Parameter - Byte 17

Bit	Field Name	Description
0	IR_ERROR_C	Illegal DPI instruction.
1	Backup too far error	The hardware stops trying to match any pattern if it is larger than 3,580 bytes by default. This backup rescans the same byte or bytes. If the hardware attempts to backup more than 3,580 bytes, an error flag is set in the status returned with the job that attempted it.
2	SC stack overflow	Start condition stack overflow.
3	SC stack underflow	Start condition stack underflow.
4	Uncorrected parity error	RAM parity error.

Vector Result Format

The Vector Results format uses the first 16 bytes, bytes 0 through 15, of the task results as a bit vector map. Result tokens numbered 0 to 127 are indicated by bits set in the 128 bits of the 16 bytes, starting with 0 as the first bit of the first byte that indicates token 0, and ending with the last bit of byte 15, 127, that indicates the result token 127.

For this format, you cannot use more than 128 result tokens, and they must be numbered 0 through 127.

DPI Data Structures, Commands, and Exception Handling

For more information on the DPI configuration registers, context record formats, API data structures, commands, performance monitoring, and exception handling, refer to the *Axxia Deep Packet Inspection Programming Guide*.

Chapter 10: Packet Integrity Check (PIC) Engine

Overview

The Packet Integrity Check (PIC) accelerator engine acts as an intermediate engine that receives an input task from a preceding engine. It efficiently calculates a cyclic redundancy check (CRC) or checksum for a packet without needing the MPP or CPU engines, then sends its result to another engine in the pipeline instance. You define the preceding and succeeding engines as part of a pipeline.

The PIC supports two task receive queues; one high priority and one low priority. The PIC processes tasks and maintains task order within a given task priority in each task queue as it performs the operations you request. The PIC can simultaneously operate on multiple tasks, but preserves the order of the issued tasks in each task queue.

Optionally, the PIC can update the packet with the calculated CRC or checksum. The PIC can modify the IP header checksum, IP L4 checksum, the SCTP CRC or checksum, and the trailer CRC.

The PIC can perform only a single CRC or checksum option per task, except for IPHeaderAndL4Checksums, where two checksums are calculated.

Features

The PIC engine can perform a number of CRC and checksum calculations that many IP Layer 4 protocols use for packet verification.

CRC and Checksum Operations

Use the CRC or checksum calculation to verify an incoming packet, or calculate a new value for a modified packet. The input task includes pointers to the packet in memory, start and end offsets, and the designated CRC or checksum calculation to perform. By specifying the start and end offsets, you can use the PIC engine to calculate a CRC or checksum on an *arbitrary* block of data. Optionally, the PIC engine can update the packet with the calculated CRC or checksum. The PIC engine can modify an IP header checksum, an IP Layer 4 checksum, a Stream Control Transmission Protocol (SCTP) CRC or checksum, or a trailer CRC.

The PIC engine supports the following common CRC algorithms.

- CRC16-USB
- CRC16-X25 and PPP.
- CRC16-UTRAN Frame Protocol.
- CRC32-AAL5.
- CRC32-Ethernet (Frame Check Sequence).
- CRC23C-RAW
- SCTP-RFC3309 (CRC32C)

The PIC engine supports the following common checksum algorithms.

- IP Header checksum
- IP Layer 4 checksum
- IP Header checksum and the IP Layer 4 checksum, if they exist in the packet
- Generic 16-bit checksum
- Generic 16-bit checksum using XOR of ones (checksum inverted)

- SCTP-RFC2960 (Adler32 checksum)
- BIP (Bit Interleaved Parity).

Additionally, you can configure the PIC engine with up to four user-defined and software-programmable CRC polynomials.

NOTE A software-programmable CRC is less efficient to use than a hardware-supported CRC.

Supported IP Layer 4 Protocols

The PIC engine supports several IP Layer 4 protocols. The NextHeader (NH) or protocol value identifies each Layer 4 protocol, part of the IPv4 or IPv6 header. The following table identifies each supported IP Layer 4 protocol, its NextHeader value, support for IPv4, IPv6, or both, and the corresponding Request for Comments (RFC) document number.

Table 33 PIC Supported IP Layer 4 Protocols

IP Layer 4 Protocol	NextHeader Value	IP Support	RFC Document
Authentication Header	0x33	IPv6	RFC 4302
Destination options	0x3c	IPv6	RFC 2460
Fragment Header	0x2c	IPv6	RFC 2460
Hop by Hop options	0x00	IPv6	RFC 2460
ICMP-v4	0x01	IPv4	RFC 792
ICMP-v6	0x3a	IPv6	RFC 4443
IGMP	0x02	IPv4 and IPv6	RFC 3376
MLD	0x3a	IPv6	RFC 3810
Mobility Header	0x87	IPv6	RFC 3775
No Next Header	0x3b	IPv6	RFC 2460
Routing Header	0x2b	IPv6	RFC 2460
TCP	0x06	IPv4 and IPv6	RFC 793
UDP	0x11	IPv4 and IPv6	RFC 768
UDPLite	0x88	IPv4 and IPv6	RFC 3828

PIC Engine Processing

The PIC supports three modes of operation for processing an incoming packet, as described in the following table.

Table 34 PIC Packet Processing Modes

Mode	Description
Check	Verify the CRC or checksum of the packet, based on the type supplied by the task.
Generate	Generate the CRC or checksum for the packet, based on the type supplied by the task.
Modify	Generate the CRC or checksum for the packet and modify the corresponding fields in the packet, based on the type supplied by the task.

The PIC type is a one byte value supplied by the task that defines the CRC or checksum to be performed by the PIC on the packet.

Table 35 PIC Type Values for Available Algorithms

Value	CRC or Checksum Setting
0x21	CRC16 UTRAN Frame Protocol
0x22	CRC16-USB
0x23	CRC16-X25/PPP
0x24	CRC32-AAL5
0x25	CRC32-Ethernet
0x26	CRC32C
0x28	Variable CRC #0 (software configured)
0x29	Variable CRC #1
0x2a	Variable CRC #2
0x2b	Variable CRC #3
0x01	IP header checksum
0x02	IP L4 checksum (Only L4 checksum)
0x03	Both IP header and L4 checksum
0x04	Generic 16 bit checksum
0x05	Generic 16 bit checksum with XOR of 1's
0x08	SCTP-RFC2960 (Adler32 checksum)
0x09	SCTP-RFC3390 (CRC32C)
0x10	BIP16
0x00	No-op (do nothing)

The PIC processes incoming packets using the parameters passed to it using the task.

After the CRC or checksum calculation, depending on the mode of operation, either the packet is modified or not and sent to the output task queue, along with the task parameters for the successive engine in the Virtual Pipeline. You can mask a subset of the errors detected by the PIC during processing.

Because of the nature of protocols and error checking, not every combination of errors is possible. The PIC parses the packet to calculate header and embedded checksums until either the parsing is complete, or a fatal packet error occurs. Packet errors can be fatal or non-fatal, with the following behaviors:

- If the PIC encounters one or more fatal errors it stops parsing the packet, so that any additional errors that might have been discovered with further parsing are not reported.
- Non-fatal error conditions can be reported simultaneously, potentially together with one or more fatal errors. The following conditions are considered non-fatal, if you have configured the error mask to ignore the checking of the corresponding errors:
  - Detected illegal TCP flags
  - Detected Mobility Header NH is not NONEXT
  - Detected IPv4 TTL=0 or IPv6 HL=0
  - Detected a fragmented packet
  - Detected CHK=0000 IPv6 UDP

The PIC can process multiple tasks at the same time, and the task order is always maintained in each task queue so that the output task queue contains the packets in the same order they arrived in the input task queue.

PIC Input and Output Parameters

PIC Input Parameters

The PIC input parameters configure the PIC for a single operation.

Table 36 PIC Input Task Parameters

Parameter	Size	Description
flags	1B	Bits 7:6 – The upper 2 bits show the type of relative offset. ■ 00: Relative to end of packet ■ 01: Relative to start offset ■ 10: Relative to start of packet Bit 5 ■ 1: Generate a checksum. ■ 0: Validate a checksum Bit 4 ■ 1: Modify the packet. ■ 0: Do not modify. Bits 3:0 are reserved.
type	1B	The type of checking to perform. See the following table for the types of checking performed.
startOffset	2B	The offset of the first byte of the packet data to examine.
endOffset	2B	Based on the endOffsetType parameter.
chk16Seed	2B	Initial seed for generic 16-bit checksum. Default: 0x0000.

Table 37 ASE Label and PIC Checking Performed

ASE Label	PIC Operation Performed
UTRAN_16	CRC16 UTRAN Frame Protocol
USB_16	CRC16 USB/XMODEM
CPPP_X25_16	CRC16 X25/PPP
ETH_32	CRC32 AAL5
CRC32_Ethernet	CRC32 Ethernet
CRC32C_Raw	CRC32C
VariableCRC_0 VariableCRC_1 VariableCRC_2 VariableCRC_3	Four CRCs for which you define the parameters.
IPHdrChecksum	IP header checksum
IP_L4Checksum	IP Level 4 checksum (only Level 4 checksum)
IPHdrAndL4Checksums	Both IP header and Level 4 checksum
Generic_16bit	Generic 16-bit checksum
Generic_16bit_XOR1s	Generic 16-bit checksum with XOR of 1s
Adler32_SCTP	SCTP-RFC2960 (Adler32 checksum)

ASE Label	PIC Operation Performed
CRC32C_SCTP	SCTP-RFC3309 (CRC32C)
BIP16	Byte Interleaved Parity 16
NO_OP	Do nothing

Setting Input Parameters

The following table shows the effects of setting the **generate** and **modifyPacket** parameters for different CRC types.

Table 38 Input Parameter Settings Affects on CRC Operations

CRC Type	generate	modifyPacket	Description of Parameter Setting Impact
CRC16_UTRAN CRC16_USB CRC16_X25_PPP CRC32_AAL5 CRC32_Ethernet CRC32C_Raw	F	F	The CRC is calculated as a check. The calculated CRC and pass/fail are returned in the task parameter.
	T	F	The CRC is calculated as a generate. The calculated CRC is returned in the task parameter.
	T	T	The CRC is calculated as a generate. The calculated CRC is returned in the task parameter. The calculated CRC is always appended to the end of the PDU.
IPHdrChecksum IP_L4Checksum IPHdrAndL4Checksum	F	F	The IPv4 header and/or Level 4 checksum is calculated as a check. The calculated checksum and pass/fail are returned in the task parameter.
	T	F	The IPv4 header and/or Level 4 checksum is calculated as a generate. The calculated checksum is returned in the task parameter.
	T	T	The IPv4 header and/or Level 4 checksum is calculated as a generate. The calculated checksum is returned in the task parameter. The calculated checksum replaces the data in the PDU.
Adler32_SCTP	F	F	The Adler checksum is calculated. The SCTP header checksum is extracted and compared with the calculated value. The calculated Adler checksum and pass/fail are returned in the task parameter.
	T	F	The Adler checksum is calculated. The calculated Adler is returned in the task parameter.
	T	T	The Adler checksum is calculated. The calculated Adler checksum is returned in the task parameter. The calculated Adler checksum is replaced in the SCTP header.
CRC32C_SCTP	F	F	The SCTP CRC is calculated. The SCTP header CRC is extracted and compared with the calculated value. The calculated CRC and pass/fail are returned in the task parameter.
	T	F	The SCTP CRC is calculated. The calculated CRC is returned in the task parameter.
	T	T	The SCTP CRC is calculated. The calculated CRC is returned in the task parameter. The calculated SCTP CRC is replaced in the SCTP header.
BIP16	T or F	F	The calculated BIP is returned in the task parameters. Note that the BIP cannot modify a packet.

CRC Type	generate	modifyPacket	Description of Parameter Setting Impact
VariableCRC_0 VariableCRC_1 VariableCRC_2 VariableCRC_3	F	F	The generic CRC is calculated as a check. The calculated CRC is returned in the task parameter. Note that the generic CRC cannot modify a packet.
	T	F	The generic CRC is calculated as a generate. The calculated CRC is returned in the task parameter. Note that the generic CRC cannot modify a packet.
Generic_16bit Generic_16bit_XOR1s	F	F	The generic checksum is calculated as a check. The calculated checksum is returned in the task parameter. Note that the generic checksum cannot modify a packet.
	T	F	The generic checksum is calculated as a generate. The calculated checksum is returned in the task parameter. Note that the generic checksum cannot modify a packet.

PIC Output Parameters

The PIC engine has two output parameters, status and computedValue. The 32-bits of the status output parameter are the first five rows of in this table.

Table 39 PIC Output Parameters

Field	Bit	Width (Bits/Bytes)	Description
ALLGOOD	63	1b	Pass or fail. ■ 0: The task failed one of the error code checks ■ 1: No error
Status Code	62-46	17b	See the *PIC Status Code Error Bits* table for the information stored in these seventeen bits. Valid only when the type of checking is IPHdrChecksum, IP_L4Checksum, or IPHdrAndL4Checksum. These bits are set to zero if not operating on an IP checksum.
errorCode	45-43	3b	■ Bit 45 – Set if the Layer 3 IP header checksum is good or the packet is IPv6. Valid only when the type of checking is IPHdrChecksum, IP_L4Checksum, or IPHdrAndL4Checksum. Setting the **generate** input parameter sets this bit ■ Bit 44 – For each type of PIC operation, this bit is set if the check is successful. Setting the **generate** input parameter sets this bit ■ Bit 43 – Set if the Layer 3 IP header and the Layer 4 payload are recognized. Valid only when the type of checking is IPHdrChecksum, IP_L4Checksum, or IPHdrAndL4Checksum.
NextHeader	42-35	8b	The IPCHK layer4 protocol. This is returned if input task parameter type is 01, 02, 03.
errorCode	34-32	3b	■ Bit 34 – DATAERR: used to identify an error on the data fetch for the packet. ■ Bit 33 – PARAMERR: used to identify an error with the input task parameters. ■ Bit 32 – TASKERR: used to identify an error on the input task.
computedValue		4B	The computed result based on the configured operation. When both the IP header and L4 checksum are being output, then this is the header checksum followed by the L4 checksum. The output most significant bits are padded with zeroes as needed. On an illegal task or malformed packet the computed value is undefined.

PIC Status Parameter

The following table shows the meaning of the 17 error status bits in the **Status Code** parameter.

Table 40 PIC Status Code Error Bits

Bit	Description
62	Detected IPv6 jumbogram (packet exceeding standard MTU
61	Detected Mobility Header NH is not NONEXT
60	Detected CHK = 0000 IPv6 UDP *or* Detected illegal TCP flags
59	Not enough data to process extension header
58	Detected IP TL is greater than PDU length
57	Detected fragment
56	Detected ODD length IPv6 ext authentication or routing header
55	Detected UDP length or UDPLITE length is greater than IP TL
54	Detected UDPlen or UDPLITElen is less than 8 (not UDPLITE=0) *or* Detected TCP Data Offset is greater than IP TL
53	Detected IP TL not large enough to support Level 4 protocol
52	Detected an unsupported protocol
51	Detected IPv4 Time to Live = 0 or IPv6 HopLimit = 0
50	Detected IPv4 TL is less than IPv4 HL
49	Detected IPv4 HopLimit is less than 5
48	Detected header length is greater than PDU length
47	Detected packet that is not IPv4 or IPv6
46	General failure (hardware problem)

PIC Exception Handling

The PIC engine discards all incoming tasks that have the task error bit set and does not set the task error bit. The PIC engine reports the following kinds of errors by logging sticky bits, which you can configure to trigger an interrupt and transmit in the errorCode output parameters.

- Data hardware errors reported from system memory
- Illegal or malformed task parameters
- Malformed packets, by enforcing packet consistency checks and field value checks as defined by the applicable standard references documentation
- When PIC Type (1B) is *not* one of the supported CRC/checksum setting, PIC flags the output task as a FAIL, by setting the PARAMERR bit (Bit 33) in its output task parameters.

PIC Statistics Monitoring

The PIC has configuration registers that count:

- Number of tasks received without a task error
- Number of tasks received with a task error
- Number of tasks sent without an error
- Number of tasks sent with an error

Apart from these counters, you can use the SMON counters and utilities to measure PIC-specific metrics.

Chapter 11: Network Compute Adapter (NCA)

This chapter defines the Network Compute Adapter (NCA) capabilities that transfer tasks between the CPUs and the accelerator engines.

Overview

The Network Compute Adapter (NCA) transfers tasks between the CPUs and the accelerator engines. The CPUs use the NCA to send and receive messages. The NCA enables the host CPU to configure the engine sequence.

The NCA engine has separate task receive queues and transmit queues that permit concurrent processing of different protocol levels. The CPUs can directly access these task queues from the user space.

Tasks, containing packet data, can have up to 32 bytes of parameters accessible by the software. All communication is through simple PCQs that are located in cached CPU memory. Therefore, the CPU overhead is minimized to a small number of cached memory accesses. The buffer pools support four allocation sizes (256 B, 2 KB, 16 KB, and 64 KB). The memory management block (MMB) controls these buffer pools so that the software can efficiently free or allocate buffer pools.

Regarding the memory-management model, the receive packet data is placed in buffers that the memory management block (MMB) allocates, whereas the transmit packets can be placed either in memory management-allocated buffers or in software-managed buffers. If the transmit packet data is in software-managed buffers, the NCA copies this data into the memory-manager-allocated buffers. The software can then reuse the original software-managed buffer after the NCA consumes the entry. However, when the transmit packet data is in memory-manager-allocated buffers, the NCA need not copy the data, and so the buffer is returned to the memory manager as a free buffer after the message is consumed.

For more information about the NCA see the *Network Compute Adapter (NCA)* chapter within the *Axxia AXX2500 Family of Communication Processors Multicore Reference Manual*.

Chapter 12: Ethernet Input Output Adapter (EIOA) Engines

The AXX2500 family has Ethernet Input/Output Adapter (EIOA) engines and a Preclassifier and Ethernet Switch (PCX). The EIOA-PCX performs Layer-2 Ethernet switching with Layer 3 and Layer 4 support. The PCX routes and switches packets, performs policing, modifies packets, gathers statistics, and enforces any access control lists (ACLs).

Overview

The AXX2500 family has two Ethernet interface adapters engines, EIOA0 and EIOA1, that each support two slots: a single 10G interface on one slot and up to four 1G interfaces on the other slot. Each EIOA engine connects to the Axxia Virtual Pipeline interface and memory.

Each slot contains an Ingress Packet Processor (IPP) and Egress Packet Processors (EPP). The IPP parses incoming packets, gathers the information needed for classification, and collects the classification results. The EPP shapes the traffic on the output ports, plus optionally modifies packet fields.

The bridging layer has a hardware based learning feature which can be used to bridge packets between Ethernet ports, offloading the process from other acceleration engines.

You can use the Preclassifier to configure the packet classification, based on the packet data. The bridging layer (BL) can determine which of the following packet actions to perform.

- Switch the packet
- Determine the packet bridging destination
- Filter packet according to an access control list (ACL)
- Policing
- Store VLAN statistics

EIOA Engines with Preclassifier and Ethernet Switch (PCX) Features

Each EIOA engine can serve as a start engine or an end engine for an engine sequence, creating a new task on packet arrival, and receiving a task for transmitting a processed packet.

Each EIOA engine receives packets from and transmits packets to external Ethernet interfaces. Each EIOA engine communicates with other engines by passing tasks that include parameters, flow IDs, packet information, or a pointer to packet information.

Additionally each EIOA engine supports these Ethernet switching capabilities.

- Hardware-based Address Learning
- VLAN Support
- Access Control List (ACL) support, using Layer 2, Layer 3, and Layer 4 information
- Ingress Policing
- Scheduling and Shaping
- Additional Ethernet Switching Functions

Hardware-based Address Learning

The Axxia EIOA supports the following Hardware-based Address Learning features.

- Supports up to 8000 addresses
- Supports programmable per port mask or drop packet behavior
- Provides statically programmable addresses
- Supports hardware-based forwarding
- Supports hardware-based address learning and aging

The EIOA provides lookup support for the following protocols.

- Independent VLAN Learning (IVL)
- Shared VLAN Learning (SVL)
- QinQ Tunneling (QinQ)
- Multiprotocol Label Switching (MPLS) address

VLAN Support

The EIOA/PCX supports the following VLAN features.

- Supports up to 4096 simultaneous VLANs
  - Port-protocol and MAC-based VLANs
  - Private VLANs
- Validates all VLAN IDs received with a packet
- Provides VLAN filtering on input ports
- Provides VLAN tag insertion, deletion, and replacement based on task parameters
- Performs optional static routing based on the VLAN ID
- Accumulates VLAN-based statistics
  - Monitoring done with 1024 available counters (counts both bytes and packets)
  - Counters can work on a per VLAN, Port (Rx or Tx), or Class of Service (Cos) Priority basis
- Supports up to 1024 packet byte counters and 1024 packet counters you can configure

Access Control List Support

The EIOA-PCX supports the following features for access control lists.

- The EIOA can support up to 32 ACLs (Access Control Lists), each with up to 16 rules (Access Control Entries) per ACL.
- Each ACL rule s is based on a table lookup, which spans the following Layer 2, Layer 3, and Layer 4 protocol fields.
  - MAC (Media Access Control) source address
  - MAC destination address
  - IP source address
  - IP destination address
  - TCP and UDP source port
  - TCP and UDP destination Port
  - IP protocol and Ethertype
  - VLAN Class of Service and IP priority
  - VLAN id

- A matching ACL determines which of the following actions to take.
  - Permit the packet (enqueue) or Deny the packet (do not enqueue)
  - Trust (use received CoS Priority value) or Do not Trust (use the default or programmed priority)
  - Log (task sent to the CPU to record the ACL match)
  - Port Mirror (task and packet sent to another Virtual Pipeline instance with EIOA egress)
  - Replace the VLAN ID based on a matching source MAC address (way to classify non-routed packets)

Ingress Policing

- 512 policers shared between ports
- Dual token-bucket algorithm on a per-port or per-priority basis
- Discard behavior programmed on a per-policer basis
- Packet color determination based on the Metro Ethernet Forum (MEF) 10.2 specification
- Programmable Discard and Demote based on priority, policer, or color.
- Congestion points selectable on per-policer basis

Scheduling and Shaping

The EIOA supports task scheduling on the egress ports and egress port shaping.

- Strict Priority (SP) scheduling
- Weighted Priority (WP) Scheduling
- Mixture of SP and WP scheduling on a port
- Port shaping using a single token bucket shaper

Additional Ethernet Switching Functions

- Internet Group Management Protocol (IGMP) snooping.
- Port mirroring support for both Tx and Rx ports.
- Logically combine an arbitrary number of links for higher bandwidth or fault redundancy. Up to eight ports can be aggregated.
- 802.1d and 802.1w Rapid Spanning Tree Protocol (STP) state management.
- Packet Discard (programmable maximum packet length).
- Port-protocol and MAC-based priority.

Axxia EIOA Architecture

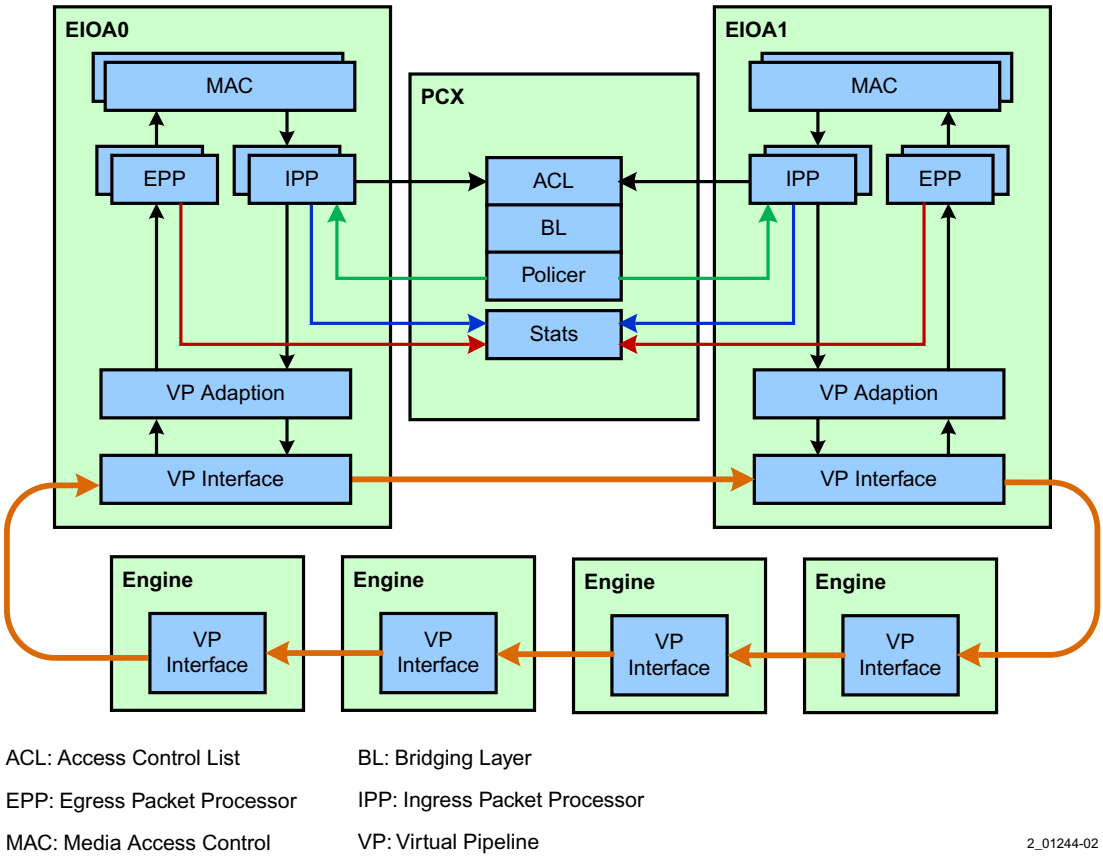

Figure 30 Axxia Dual EIOA Engines with Preclassifier and Ethernet Switching (PCX) Architecture

Each of the EIOAs exist as a separate task destination in an engine sequence, and communication with the two EIOAs is the same as with other accelerator engines. When a packet is bridged between two ports of the same EIOA engine, a task must be generated to transfer the packet from the ingress to the egress port.

EIOA Packet Classification

For ingress traffic, the two EIOAs share a Preclassifier that looks at specific packet fields to identify the packet destination and processing requirements. The Preclassifier examines the following packet information.

- Packet switching and routing – Information such as source address and destination information that includes MAC address, VLAN IDs, and MPLS labels.
- Compares the parsed packet information against a set of programmable Access Control List rules for one of 32 ACLs. You can configure the ACL to require an exact match or use designators. This check determines whether the packet is permitted or denied.

For the specific packet information examined, see the *Access Control List Support* section in this chapter.

Virtual Pipeline Destinations

Based on classification results, the EIOA drops the packet or sends it to one or more of 32 virtual pipeline destinations, with up to 24 switched destinations, including the ten EIOA ports, error tracking destinations, and other virtual pipeline entry points.

The task type determines the packet processing. The destination is one of the pieces of information in the task type. You can configure each packet destination to use one of three output task types. The following table describes the task types, parameter set size, task source, and packet destination.

Table 41 Packet Destinations Based on an Output Task Type

Output Task Type	Parameter Set Size	Task Source	Packet Destination
Legacy (ACP34xx format)	16B	EIOA Ingress	Other engines. You configure the receiving port destination; it is not determined through EIOA classification.
Classification	16B	EIOA Ingress	Other engines. A task that is bridged from the EIOA ingress port to destinations other than the EIOA egress ports
EIOA Egress	8B	EIOA, other engines	EIOA Egress. A task with a switched or bridged packet for an EIOA port

Additionally, the EIOA supports two packet output task types that are used by the following specially configured destinations.

Table 42 Special Destinations a Based on an Output Task Type

Output Task Type	Parameter Set Size	Task Source	Packet Destination
Multicast Replication	32B	EIOA Ingress	EIOA Multicast replicator Uses PCX classification and is sent to the EIOA multicast queue.
Learning	8B + 18B PDU	EIOA, MPP, or NCA	EIOA, MPP, or NCA A task with switch learning information, routed to an engine that is not an EIOA port.

Egress Processing

For egress traffic, each EIOA engine can perform packet field editing, based on task parameters, and can transmit packets based on configured scheduling and priority requirements.

Ingress Packet Data Flow Overview

When a packet arrives, the EIOA engine processes the packet using the following steps.

1. Initial EIOA processing checks packet integrity and determines if the packet requires legacy processing or PCX classification. See the *EIOA Core Logic Processing* section for common legacy packet types.
2. For non-legacy packets, the EIOA sends the packet header information to the PCX for additional classification to determine the following information.
   - The bridging destination, using the port and address
   - Which ACL to use and ACL filtering results
   - Policer to use and policing results
   - VLAN statistics
   - Virtual Pipeline Destination, based on classification results plus defined masks if appropriate, and associated output task parameters
3. The task is enqueued to one or more of the EIOA virtual pipeline destinations. Tasks with multiple destinations are enqueued in the multicast queue.

EIOA Core Logic Processing

When a packet arrives in the EIOA, the EIOA core logic performs the following functions.

- Checks packet integrity
- Performs IP checks, when necessary
- Stores packet data in memory
- Detects packets that require legacy processing – processing that skips the EIOA classification process. Legacy processing is automatically selected for the following packet types:
  - Packets less than 64 bytes in length
  - PAUSE packets
  - Ethernet Synchronization Messaging Channel (ESMC) packets
  - Virtual Pipeline Extension (VPE) packets – This is a packet type defined by a configurable Ethertype that lets you send packets from one Axxia device Virtual Pipeline instance directly to the Virtual Pipeline instance of another Axxia device. For example, a packet can be processed on one Axxia device, then sent to another Axxia device for scheduling on one of its output ports.

In addition, you can configure each EIOA input port to use one of the following modes.

- Legacy Mode – Supports processing in a similar way to earlier EIOA engine implementations, without performing classification actions.
- Classification Mode – Supports the full switching, ACL, and policing functions available in the EIOA. The packet types that require automatic legacy processing on a port in Classification Mode still receive legacy processing.

Core logic IP integrity checks generate status information that you can store and make available for task output parameters, regardless of packet classification mode.

Ingress Packet Processor (IPP) Functions

The Ingress Packet Processor performs the following tasks.

- Parsing the packet fields needed for classification. The parsing process detects the type of packet (Ethernet V2 or SNAP) and whether or not the packet has VLAN tags. This information is used to locate relevant packet fields.
- Examining Layer 2, Layer 3, and Layer 4 fields and performing checksum verification for output task parameters.
- Sending the packet field data needed by each classification block.
- Assembling the results of parsing and classification into information for the output task parameters.

Bridging Layer Classification and Switching

Bridging Layer classification performs the following switching functions that results in a 24-bit Virtual Pipeline destination.

- Forwarding (bridging)
- Address learning
- IP frame routing

Building the Bridging Lookup Keys

The EIOA supports a MAC address table of up to 8192 entries. This table typically can store approximately 4000 entries before eviction starts. The EIOA looks up source and destination addresses using the same engine table, with different keys. The bridging lookup key is a 62-bit key created to index the MAC address table.

Source Address Lookup

The EIOA is globally configured to use either the IVL mode or the SVL mode to determine the key for source address lookup.

- **Independent VLAN Learning (IVL) Mode** – This mode uses a VLAN ID extracted from a configurable VLAN ID table, using the port number or the port number plus Ethertype as an index. The EIOA creates the key using the packet 48-bit MAC destination address and the assigned VLAN ID.
- **Shared VLAN Learning (SVL) Mode** – This mode uses a valid packet VLAN ID. The packet's 48-bit MAC destination address is used to create the key.

Destination Address Lookup

The EIOA parses the packet to determine the key for the destination address lookup, based on the following switching modes.

- **One-to-one SVL Mode** – Single VLAN tagged packets. The EIOA creates the key with the single VLAN ID.
- **QinQ Mode** – Packets with valid inner and outer VLAN IDs. The inner and outer 12-bit VLAN IDs are used to create the key.
- **MPLS Mode** – MPLS packets. EIOA creates the key with the 20-bit MPLS label.

The search operation results in either a hit (address found) or a miss (address not found). In the case of a hit, the search returns an associated data structure which controls the forwarding of the packet. In the case of a miss, the packet's VLAN index is used to determine the spanning tree state by reading a software controlled table to determine if the packet can be flooded. The forwarding associated data structure consists of the following information.

- Address index
- Destination port map
- Source port number
- Layer 2 ACL deny or permit status
- Layer 2 ACL source/destination logging indication

This information is used to help determine the switching or routing decision for the packet.

For switched or bridged packets — packets sent from an EIOA input to an EIOA output port — a spanning tree algorithm can be applied, and the resulting state stored by the host processor.

MAC Address Learning

Every received packet's MAC address is queried in the MAC address table. The search operation results in either a hit (address found) or a miss (address not found). In the case of a hit, the received packet's source port is compared to the associated data in the address table. If these values match, then the packet has already been learned and the only action taken is to refresh the address age. A mismatching source port or a lookup miss results in the packet being submitted to hardware-based address learning. If possible, the address is stored in the MAC address table.

You can configure the hardware address learning to result in flooding, or sending a single learning task, for example, to the MPP.

MAC Address Aging

The age of a MAC address entry defines whether the entry is valid, invalid, or static.

- Valid entry – A value between zero and all ones (non-inclusive).
- Invalid entry – A value of all ones.
- Static entry – A value of all zeros.

A lookup hit or a learning event for an address entry causes the EIOA to refresh the entry to a value of one. Once the EIOA learns an address, it increments the address age by each configured period of time. You can define the aging period to range between approximately 100 µs to multiple years.

You configure the time-out period for an address on a per-port basis. When an address is timed out, its age is set to all ones, marking it a free entry.

ACL Classification

The EIOA supports up to 32 ACLs, with each list supporting a 9-tuple lookup on up to 16 rules. You can configure the ACL based on the port, the VLAN ID, or a combination of both.

The table lookup is based on classification of the following protocol fields (Layer 2, Layer 3, and Layer 4), using the associated classification algorithms.

Table 43 Type of Match for Each Protocol Field

Protocol Field	Classification Algorithm
MAC source address	Exact match
MAC destination address	Exact match
IP source address	Longest prefix match
IP destination address	Longest prefix match
TCP/UDP source port	Range match
TCP/UDP destination port	Range match
IP protocol and Ethertype	Exact match
VLAN Class of Service and IP Priority	Exact match

VLAN ID	Exact match

For each ACL rule, the above fields can be defined as requiring a match, match a wildcard specification, or don't care.

If a packet matches on any of the rules, you can select the Permit, Deny, Log, and Adjust Priority actions that result from an ACL classification. In addition, a policer flow is assigned, and port mirroring or VLAN replacement is indicated.

The result of the ACL classification can be used to modify the 24-bit virtual pipeline destination decision.

Ingress Policing

Each received packet is assigned to one of 512 policers based on an ACL-assigned policer ID. The policers limit the admission of packets to the EIOA by performing flow-based packet limiting.

You can configure policers to accept a maximum number of bytes per second. Policers classify packets as *out of profile* when the maximum number of bytes per second limit is exceeded.

Multicast storm control works similarly, and is performed on packets that are layer 2 multicast or broadcast packets on a per-port basis. You can configure the multicast storm control to accept a maximum number of bytes per second before traffic is considered to be out of profile.

You can configure the out of profile traffic processing with the following actions.

- Unconditionally dropped
- Dropped conditionally, based on the backpressure point associated with the policer
- Demoted priority
- Forwarded normally

The policing functions comply with the Metro Ethernet Forum (MEF) 10.2 specification. Policers have two token buckets, for committed information rate (CIR) and excess information rate (EIR), with configurable thresholds and fill rates.

Destination Result Map and Mask Operations

The EIOA supports bridging and routing. A packet is routed rather than bridged if it is addressed at Layer 2 to the router MAC address configured for EIOA or if it is associated with one of the standard IP multicast Layer 2 addresses. Bridged traffic is processed using the ingress bridging logic, using programmable virtual pipeline destinations.

When the EIOA completes ingress processing, it applies masks and maps to the destination map returned from the destination lookup process. Maps add destinations; masks remove destinations. The masks and maps operate as according to the following table.

Table 44 Sequential Destination Map and Mask Operations

Step	Map or Mask Type	Destination Operation
1	Destination Map	The **initial map** used. When the destination address was not found in the MAC address table for bridged packets, a flood pattern is returned, if the spanning tree state of the port/VLAN allows flooding. When the destination address was not found in the IP address table for routed packets, a default route is chosen.
2	Protected Port Mask	Limit destination port for those packets which are received from a protected port.
3	Learning Map	When the packet source address is unknown or if its source port address does not match the entry in the address table, the learning queue is added to the destination map.
4	IGMP Snooping Map	When the packet is an IGMP packet and IGMP snooping is enabled, the IGMP snooping queue is added to the destination map

Step	Map or Mask Type	Destination Operation
5	User Snooping Map	When the packet is a TCP packet, user snooping is enabled, and the TCP destination port number matches the user_port[15:0] value, the user snooping queue is added to the destination map.
6	Aggregation Mask	A mask is applied which is used to select a single port from each port aggregation group. This mask is retrieved from an 8-entry table which is indexed by an arithmetic reduction of the packet's MAC destination and source addresses. An additional aggregation ID mask is applied in order to mask source aggregate transmissions.
7	VLAN Mask	Disables all of the destinations which are not a member of the received packet VLAN. This mask is retrieved from a table indexed by the packet VLAN ID. This mask is not applied to routed packets.
8	Route Queue Map	When the input to the VLAN mask (from the output of the aggregation mask) is non-null and the output of the VLAN mask (that is, dest_map[23:0]) is null and the destination address was found in the address table, then the packet is marked as a candidate for routing. When the packet is marked as a candidate for routing and supervisor routing is enabled, then the destination map bit which corresponds to the supervisor's routing queue is asserted. This map is not applied to hardware routed packets.
9	Source Port Mask	The packet receive port is removed from the destination map through the use of a mask which is retrieved from a table indexed by the receive packet's port number. This mask is also used to enforce private VLANs and other forwarding restrictions. This mask is not applied to routed packets.
10	Spanning Tree Tx Mask	Ports that are disabled due to the spanning tree algorithm are masked off for transmission. This mask is not applied to routed packets.
11	IP Header Error Mask	When the IP checksum validation performed during ingress processing fails, a mask is applied to disable transmission of the packet. When the IPv4 options handling is enabled by the receive port, this mask is applied to all packets containing these types of IPv4 fields. A supervisor queue reserved for ICMP-related events is added to the destination map. The supervisor is expected to formulate the appropriate ICMP response back to the original source of the frame.
12	OAM Queue Map/Mask	When the packet is an OAM packet, and the port has OAM enabled, then the OAM port is added to the bit map, with the OAM mask applied to remove any unwanted ports.
13	Layer 2 Denial Mask	When the source address is flagged for entry denial, then all of the active transmit ports are masked off.
14	Time to Live Expired Mask	When the time to live (TTL) value in the IP header is equal to zero or one, and the frame is a routed frame, then the frame has expired and is not forwarded. Instead, a mask is applied to disable destinations while a supervisor queue is added as a destination. This supervisor queue is reserved for ICMP-related events such as TTL expiration. The supervisor is expected to formulate the appropriate ICMP response back to the original source of the frame.
15	Mirroring Map	When the packet receive port is on the mirroring list or if any of the bits asserted in the destination map at this stage of processing correspond to ports being mirrored, then the mirror port bit is asserted in the destination map.
16	Multicast Rate Discard Mask	When the policer indicates that this packet is part of a multicast storm, and multicast storm discard is set, then this mask is applied and some or all of the destinations are masked and eliminated from the destination map.
17	ACL Deny Mask	When this packet has been marked for Deny by ACL processing, then this mask is applied and some or all of the destinations are eliminated from the destination map.
18	CRC Error Mask	This mask is applied if there is any framer error.
19	Global Mask	This mask is a catch-all which is used to shut down traffic to a particular port.

EIOA Multicast Replication

The EIOA can send tasks to multiple destinations for the following reasons.

- Multicast or broadcast MAC addresses
- Mirroring of tasks to multiple ports
- Unlearned destination addresses

When a task needs to be multicast, it is first processed through the EIOA and then sent back to the same EIOA with a destination bitmap and a parameter that indicates that this task is intended for multicast replication. When it arrives at the EIOA, the task is then replicated multiple times, creating a task for each of the bitmap destinations, and then the task is deleted.

If the replicated task is part of an unlearned flow, as indicated in the task parameters, a counter keeps track of the number of outstanding transactions to the corresponding address. When an address is learned, the address must be routed through the multicast logic until all outstanding transactions are completed, at which time it can be routed directly through the output. This process maintains the order for a given Ethernet traffic flow.

VLAN Statistics

The EIOA keeps VLAN statistics using the following sets of counters.

- 1024 packet counters
- 1024 byte counters

The EIOA can count block bytes or packets for sets of packets, based on the packet descriptions it receives for each packet. You can configure what fields for which to count packets.

The EIOA receives packet descriptions from each IPP and EPP. The inputs include the following elements.

- Transmit or receive packet (For example, could use this to count Tx packets or bytes.)
- Physical port number
- Packet byte count
- VLAN ID
- IP priority or Class of Service

Each packet is counted once or ignored based on descriptions. When a packet is counted (packet counters) its byte count is added to the corresponding byte counter.

The counters enables are controlled by selecting a subset of the input parameters. For example, you can configure the EIOA to count the following items.

- Packets transmitted and received per port
- Packets transmitted and received per priority
- Packets received on 1024 selected VLAN IDs
- Packets transmitted and received on 512 selected VLAN IDs
- Packets transmitted and received on 510 selected VLAN IDs, plus other transmitted and received packets

EIOA Egress Processing and Scheduling

The EIOA egress processing supports:

- Queue scheduling
- Port shaping
- Packet field modification

Queue Scheduling

Each port has four queues. You can configure a port's queues to schedule packets using the following modes:

- Strict priority — Higher priority traffic is always serviced before lower priority traffic.
- Weighted Priority (byte based) — Weights are assigned to queues based on packet byte counts. All queues with non-zero weights are serviced proportionately according to their weights.

You can mix scheduling algorithms for a single port. For example, you could have a strict priority queue and three weighted priority queues. The behavior would be that the strict priority queue is serviced whenever it has traffic, and the weighted priority queues would share the remaining bandwidth according to weights.

Queue scheduling is work conserving. If only one of the queues is active, it will consume 100% of the available port bandwidth.

Port Shaping

The EIOA engine port shaping feature lets you limit port bandwidth using a single token bucket. This is an optional configuration setting.

Packet Field Modification

The Egress Packet Processor (EPP) can perform packet modifications. Based on the received tasks parameters, the EIOA can modify the following fields.

- MAC source and destination addresses
- Up to two VLAN IDs can be added, removed, or modified.
- Class of Service field
- TTL and Checksum (IPv4)
- Priority (IPv4/IPv6)

You can use the following EIOA capabilities.

- Each packet processed by the EIOA has a VLAN address, either natively or assigned by the PCX. For packets that should not be tagged with a VLAN address, that address can be removed.
- Based on the result of the ACL lookup, adjust the CoS or priority information received with the packet. Based on the task parameters, the priority can be changed.
- If a frame is routed, the source MAC address can be modified and the TTL information, plus for IPv4, the checksum.

Backpressure Support

The EIOA accepts backpressure from three sources:

- PAUSE frames from an external link (802.3x or 802.1Qbb)
- An accelerator engine
- Task queue backpressure

Handling PAUSE Frames

For 802.3x PAUSE frames, the MAC can be configured to perform these actions.

- Respond by not allowing transmissions after the current packet is sent
- Filter PAUSE frames, and not respond or forward them
- Forward the PAUSE frame to the parser for further processing

For 802.1Qbb priority-based PAUSE frames, the EIOA performs these actions.

- Filter PAUSE frames, and do not respond or forward them.
- Forward the PAUSE frame to the parser for further processing.
- Map the 8-bit priority to the four port queues, and stop serving task starts for that queue.

Engine Backpressure

Backpressure signals are 24-bit messages that indicates endpoints, if any, that need to be backpressured by the EIOA. The EIOA can respond to backpressure signals with the following actions.

- Creating Tx PAUSE packets, either 802.3x or 802.1Qbb
- Backpressuring policer queues.

Use Case Examples

The following examples show you can use the EIOA for specific applications.

Preclassifier MPP Assist

The EIOA supports up to 8192 MAC addresses, but many applications require tracking many more addresses. In this case, the MPP can store the complete database of MAC addresses for the application, with the EIOA supporting a cache of the most recently used addresses.

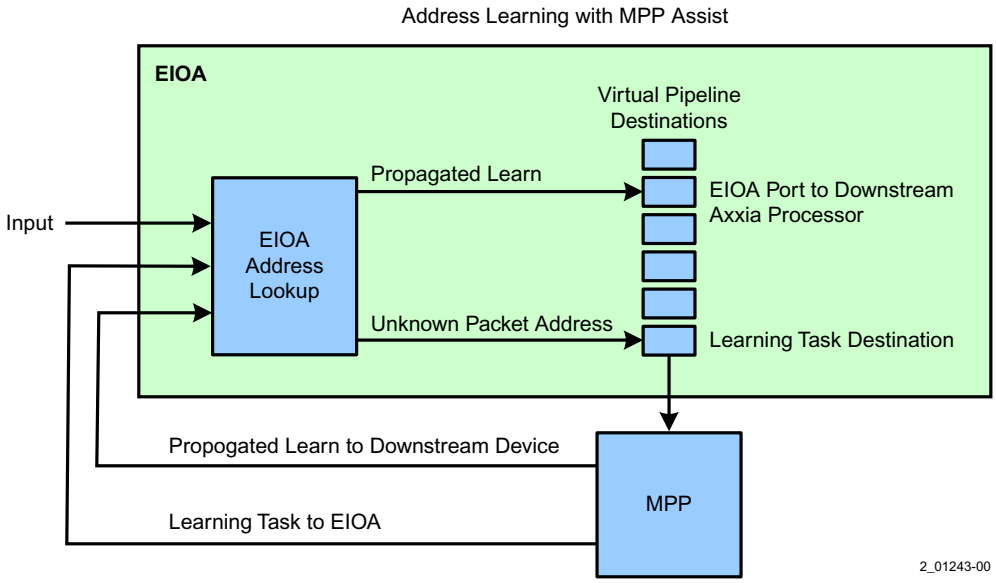

Figure 31 Address Learning with MPP Assist

With this configuration, when the EIOA receives a packet with a new MAC address, the EIOA sends a learning task to the MPP. The MPP determines which addresses to keep in the EIOA and sends learning and unlearning tasks to the EIOA to maintain that database and keep the two databases synchronized.

Similarly, this technique can be used to synchronize MAC address tables across multiple Axxia devices. The additional Axxia devices can be sent learning tasks, and can send learning tasks, to keep the MAC addresses in each device synchronized.

In addition to learning assist, the EIOA can assist the MPP with ACL lookups, policing, and statistics counters, offloading work the MPP would otherwise perform.

Daisy Chaining

For some designs, the network nodes may be connected in a daisy chain configuration. A message sent to one device is passed on to the next device in the chain, repeatedly until the device the message is intended for receives it.

With the bridging capabilities of the EIOA, the daisy-chained messages can be quickly recognized by the first EIOA classifier and switched to the next EIOA — without requiring any additional processing that could delay the message arriving at its intended device.

Virtual Pipeline Extension (VPE)

The Virtual Pipeline Extension permits the Axxia device that receives a packet to process that packet and encode the resulting output parameters in the packet for further processing by a downstream Axxia device. In effect, the packet is passed from the first Axxia device virtual pipeline task ring to another Axxia device virtual pipeline task ring.

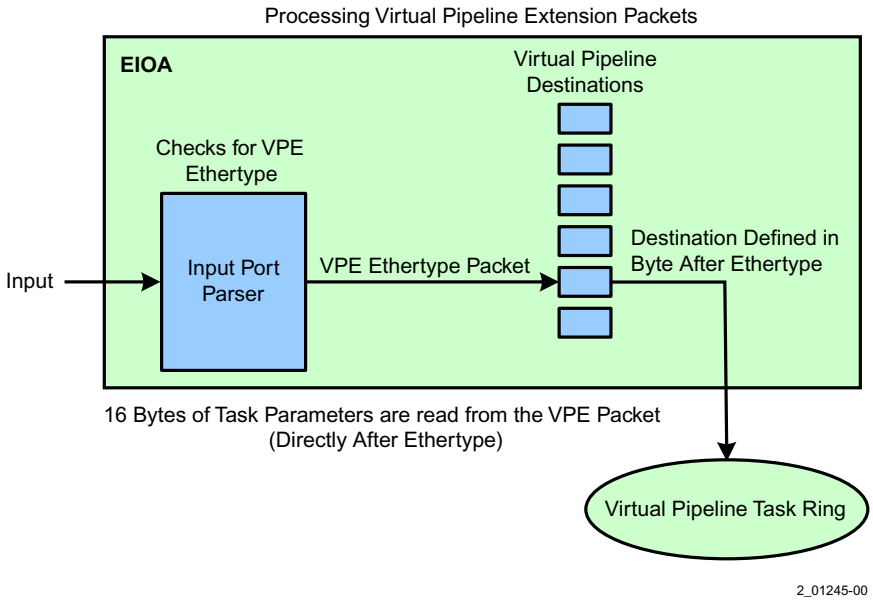

Figure 32 Processing Virtual Pipeline Extension Packets

This capability is achieved by defining an Ethertype to mark VPE packets. The sending Axxia device inserts the 16 bytes of task parameters, including the VPE Ethertype.

For example, in applications that require multiple Axxia devices, the device that receives a packet may not have the port that needs to schedule that packet. This device performs all necessary processing, except for scheduling to the destination, encodes the output parameters to create a VPE packet, and sends it to the Axxia device that services the port the packet requires. That Axxia device schedules the packet to the port and destination.

When the downstream Axxia device receives the packet, the EIOA parser recognizes it as a VPE packet, extracts the output parameters, and sends it out as a task with the parameters to a downstream engine. In this example, the packet is sent to the MTM for scheduling on the appropriate port.

Simple QoS without using the MTM

By using the port shaper and queue scheduling facilities of the EIOA, you can set up a simple QoS implementation that does not require the use of the MTM to schedule packets.

Use the port shaper capability to limit the bandwidth for a port. Then, configure the four port queues with the scheduling algorithms you want. For example, you might have one queue configured for strict priority to handle the highest priority traffic, with the remaining three queues scheduled using a weighted round robin algorithm.

Axxia Communication Processor Glossary

This glossary contains acronyms, terms, and definitions used with Axxia communication processors.

Numbers

3DES

Triple data encryption standard (DES)

3GPP

Third Generation Partnership Project

8B/10B

A line code used in telecommunications and Ethernet that maps 8-bit symbols to 10-bit symbols to achieve DC-balance and bounded disparity, and yet provide enough state changes to allow reasonable clock recovery.

A

Accelerator Engines

Accelerator engines are intelligent engines within an Axxia communication processor that perform specialized packet processing. They are sometimes called data path accelerator engines.

- DPI: Deep Packet Inspection
- EIOA: Ethernet Input/Output Adapters
- MMB: Memory Management Block
- MPP: Modular Packet Processor
- MTM: Modular Traffic Manager
- PAB: Packet Assembly Block
- PIC: Packet Integrity Check
- SED: Stream Editor
- SPP: Security Protocol Processor
- TMGR: Timer Manager

ACL

Access Control List

ADK

Axxia Development Kit

AEAD

Authenticated Encryption with Associated Data

AES

Advanced Encryption Standard

AH

Authentication header

ALU

Arithmetic Logic Unit, which is a digital circuit that performs arithmetic and logical operations.

AMC

Advanced Mezzanine Card is a printed circuit board (PCB) that follows a specification of the PCI Industrial Computer Manufacturers Group (PICMG).

anti-replay

An IPSec subprotocol designed to avoid hackers making changes in packets traveling from a source to a destination. The subprotocol uses a unidirectional security association to establish a secure connection between two network nodes. Once a secure connection is established, the subprotocol uses a sequence number or a counter to ensure the reception of sequenced transmitted packets.

API

Application programming interface

argument signature register

A 128-bit register within the MPP accelerator engine of an Axxia device. When an FPL program calls a function, the MPP loads all arguments passed to the function into the register and makes the contents of the register available to the FPL function as if the arguments were inserted into the packet immediately before the position indicated by the current pointer. This action allows an FPL function to treat its argument and packet data as one continuous stream for packet matching purposes.

ASE

Axxia Software Environment

An Eclipse-based graphical interface for building the Virtual Pipeline instances and packet data structures for use with the Axxia family of multicore communication processors. LSI provides the ASE as a stand-alone Eclipse package, or as a plug-in for an existing Eclipse installation.

Within the integrated development environment of the ASE, you can generate header files for FPL programs, C-NP scripts, and the C programs used by the CPU engine. You can also compile your FPL or C-NP code and run simulations to test and debug your configuration and application.

ASIC

Application-specific Integrated Circuit

ASR

argument signature register

Axxia

LSI Corporation family of multicore communications processors. Pronounced ak-see-uh.

Axxia Compressed Absolute Time

An Axxia absolute time format that uses a smaller number of bits, giving up precision and range of time (relative to current time). Only used internally within the Axxia device.

Axxia Delta Time

An Axxia time format that specifies time relative to current time. The devices uses this format for (a) all timer requests made from other accelerator engines, (b) FPL time requests to the MPP hash engine timer, and (c) as part of the MTM engine Traffic Shaper.

Axxia Time

A 64-bit accumulator that counts microseconds in a fixed-point representation, with the eight least-significant bits representing fractional microseconds.

B

backpressure signal

A control message sent from a downstream accelerator engine to an upstream engine used to regulate and optimize resources. The signals derive from task receive queue buffer thresholds or specific programmable conditions. All engines can send backpressure signals and the NCA, MPP, MTM, and EIOA engines can receive the signals.

base ID

The ID of the first entry in the namespace. The base ID applies to all engine tables in the namespace.

bit-level editing

Editing done on packet segment data using the BitFields parameter by the PAB engine. These actions include (a) inserting a bit stream into an assembly at an arbitrary starting point (relative to start of assembly), (b) transmitting a bit stream from an assembly, and (c) advancing an assembly head pointer and length to remove data from assembly front

block editing

Editing done on packet segment data in 128-byte blocks by the SED engine. For many applications, this is an efficient way to change, update, or add header information. You can edit packet header, payload, or trailer data.

Branching Virtual Pipeline instance

Two or more engine sequences (branches) that share the same start engine (EIOA, MPP, Expander, or CPU) and are grouped together. You use these pipelines when each branch (engine sequence) requires a similar parameter set from the start engine, such that the branch processing differences can be managed in a flow table.

buffer management

The regulation of task flow in the receive task queues by the Traffic Manager in the MTM engine using C-NP scripts. Regulation can be done by (a) enqueuing or discarding tasks, (b) generating backpressure messages at any specific scheduling hierarchy, and (c) removing backpressure.

Buffer Traffic Manager

A compute engine within the MTM that can run C-NP scripts. Scripts that run in the Buffer Traffic Manager typically perform buffer management actions to determine whether to discard a packet or queue it for processing.

C

C-NP

C for Network Processors is a high-level language designed to perform specific functions in scripts that run within the Axxia accelerator engines. You can also use C-NP to program the MPP state subengine.

cancellable timer

A timer provided by the Time Manager (TMGR) engine that can be changed (cancelled or time-out value reset) once they start.

CBC

Cipher Block Chaining, which is a DES and AES mode.

CBS

Committed burst size

CCL

Classification Completion List, which is an internal data structure maintained by the MPP.

CDB

Connection Database

CFU

Call Forwarding Unconditional

checksum

A fixed-size datum computed from an arbitrary block of data, for detecting accidental errors introduced during its transmission or storage.

CIR

Committed Information Rate

CMN

Configuration Master Node

compute engine

A subengine that exists in several accelerator engines. A compute engine runs C-NP scripts or executes predefined functions for FPL programs running in the MPP.

COW

Coherent one-way protocol

CPU coprocessing

A technique used by Virtual Pipeline instances that define simple engine paths to and from the CPUs, letting the CPUs off load computationally-intensive packet processing functions to the accelerator engines.

CRC

A cyclic redundancy check is an error-detecting code commonly used in protocols to detect accidental changes in transmitted data. Packets prior to transmission get a short check value attached, based on the remainder of a polynomial division of the packet contents. When data is retrieved the calculation is repeated, and corrective action can be taken against data corruption if the check values do not match.

CSE

Crypto Service Engine, which is the section within the SPP in which all the cryptography engines reside.

D

DCR

Device Control Register

DES

Data Encryption Standard

DMA

Direct Memory Access is a feature that allows hardware to access memory without involving the CPU. The NCA provides hardware support for DMA task queues.

DPI

Deep Packet Inspection

DSA

Digital Signature Algorithm

DTSL

Datagram Transport Layer Security protocol

DWRR

Deficit Weighted Round Robin, which is a scheduling algorithm.

E

EBS

Excess burst size

ECB

Electronic CodeBook, which is a DES and an AES mode.

ECC

Elliptical Curve Cryptography

EIOA

Ethernet Input/Output Adapter

End Engine

An end engine can only be last in an engine sequence. (EIOA, MPP, CPU, or Expander).

Engine Sequence

The sequence or list of accelerator engines that comprise a Virtual Pipeline instance as defined in the ASE. Tasks that are launched into the Virtual Pipeline instance traverse that sequence of accelerator engines to be processed.

Engine Table

An area of memory on an Axxia device that is reserved for use by a specific accelerator engine. Engine tables contain distinct data structures that you use for packet processing. They exist within namespaces and are indexed by a contiguous range of 24-bit ID values. The most memory-efficient entry sizes are powers of 2 values (16, 64, 256, and so forth). Engine tables include these available types:

- Scheduling queues, schedulers, and shared scheduling parameters
- Stream Editor (SED) parameters
- Deep Packet Inspection (DPI) contexts
- Packet assembly or reassembly queues
- Policing and statistics tables
- Security Association (SA) information

ESN

Extended Sequence Number

ESP

Encapsulating Security Payload

F

failure interrupt

An engine interrupt that notifies the host processor of an engine processing failure. For example, a failure interrupt could be triggered by a FIFO overflow or a packet processing error.

FBI

Functional Bus Interface

FBRS

File-based Register Simulation

flow

A set of packets that a configuration and an application treat the same way during packet processing, from start engine to end engine. A user-defined set of packets that requires uniform processing.

flow ID

An index to a set of parameter values organized as a flow table entry. The pipeline start engine loads the flow table values into the task that moves through the engine sequence.

flow table

A specific set of processing parameters needed for each individual flow in the pipeline.

FPL

Functional Programming Language is a proprietary LSI programming language for the MPP accelerator engine. FPL functions are used to direct the MPP classification by running compiled FPL program on the MPP pattern processing engines. State engine functions are also used with FPL programs for operations that span multiple packets.

frame check sequence

The extra bits and characters added to data packets (frames) for error detection and control. A set of characters is added to the end of the frames, which are checked at the destination. Matching frame check sequences indicate that the delivered data is correct

G

GMAC Ethernet

Gigabit Media Access Control, a block within each EIOA.

GMAC IPsec

Galois Message Authentication Code, which is a mechanism that provides data origin authentication, but not confidentiality, within the IPsec Encapsulating Security Payload (ESP) and Authentication Header (AH).

GSM

Global System for Mobile Communications

H

handle

A token that identifies and accesses the Axxia device when using the RTE APIs. Also, a handle register holds the result of an FPL function call in the MPP.

HFM

Hyper Frame Number

HMAC

Hash message authentication code

I

ICV

Integrity check value

IETF

Internet Engineering Task Force

IKE

Internet Key Exchange

IMIX

Internet Mix, which is typical Internet traffic passing some network equipment, such as routers, switches, or firewalls. When measuring equipment performance using an IMIX of packets, the performance is assumed to resemble real-world traffic patterns and packet distributions. IMIX traffic is defined as 56 percent 64-byte packets, 20 percent 576-byte packets, and 24 percent 1518-byte packets.

initial segment

The PAB applies the StartOffset data to every segment except the initial segment that begins the reassembly. For the initial segment, the PAB uses the FirstOffset instead of the StartOffset parameter.

inline task parameters

Output parameters from upstream engines or flow table entries.

intermediate engine

An engine that neither starts or terminates an engine sequence. It receives a task, optionally modifies it, and sends it to the next engine in the sequence. Intermediate accelerator engines are SPP, DPI, PIC, PAB, MTM, and SED.

IOA

Input/Output Adapter

IPsec

Internet Protocol Security protocol

IRAM

Instruction RAM, which is memory within a compute engine that stores VLIW instructions for execution.

ISA

Instruction Set Architecture

ITA

Input Task Accumulator

ITM

Input Task Manager

IV

Initialization Vector

J

JRE

Java Run-time Environment

L

LCM

Local CPU Memory

LPM

Longest prefix match is a table search algorithm that selects the most specific matching entry in a table where entries can be incompletely specified (wild carded). Longest prefix matching is required when looking up Internet Protocol routing table entries and is one of several types of matching that the MPP supports.

LTE

3GPP Long Term Evolution

M

MAC

Message Authentication Code (cryptography) or Media Access Control, depending on its context.

MKI

master key identifier

MMB

Memory Management Block

MPE

Multiprotocol Engine

MPIC

Multiprocessor Interrupt Controller

MPP

Modular Packet Processor

MPP PIC

MPP Packet Integrity Check engine, which is an engine within the MPP that performs a subset of PIC functions.

MTM

Modular Traffic Manager

N

Namespace

An area of memory on an Axxia device that holds distinct data structures that you use for packet processing, and is indexed by a contiguous range of 24-bit ID values. A namespace can contain one or more engine tables. Engine tables are sets of entries within a namespace that are used by a specific accelerator engine. Engine tables contain data structures for use by the specific engine. The available types of engine tables include:

- Scheduling queues, schedulers, and shared scheduling parameters
- Stream Editor (SED) parameters
- Deep Packet Inspection (DPI) contexts
- Packet assembly or reassembly queues
- Policing and statistics tables
- Security Association (SA) information

NCA

Nuevo CPU Adapter

NHA

Nuevo Host Adapter

NIC

Nuevo Interconnect Crossbar

non-cancellable timer

Unstoppable timer that runs until completion.

NSMI

Nuevo System Memory Interface

O

OTM

Output Task Manager

P

PAB

Packet Assembly Block

packet

A packet consists of two kinds of data: control information and user data (also known as payload). The control information provides data the network needs to deliver the user data, for example: source and destination addresses, error detection codes like checksums, and sequencing information. Typically, control information is found in packet headers and trailers, with payload data in between.

pattern processing engine

One of several highly multithreaded packet classification subengines that can access routing tables, access control lists, and other information.

PBU

Predicate and Branch Unit – a digital circuit within the compute engines that operates on predicates to increase instruction-level parallelism, streamline control flow, and reduce the number of branch instructions.

PCIe

Peripheral Component Interconnect Express

PCQ

Producer-Consumer Queue

PCX

Preclassifier and Ethernet Switch

PDCP

Packet Data Convergence Protocol

PDU

A Protocol Data Unit is (a) Information delivered as a unit among peer entities of a network, containing control information, such as address information, or user data. (b) In a layered system, a data unit of data specified in a protocol of a given layer that consists of protocol-control information and possibly user data of that layer.

PIC

Packet Integrity Check

PIM

PQM Instruction Memory

PIPE

PHY Interface for PCI Express, which is a standardized interface between a PCIe physical interface (PHY) block and a PCIe media access control (MAC) block, as defined in the *PHY Interface for the PCI Express Architecture* specification from Intel Corporation.

pipeline

Virtual Pipeline instance is the preferred terminology. One or more processing engine sequences through the Axxia device.

PIR

peak information rate

PMU

Performance Monitor Unit. Different Axxia devices include different internal CPUs, that in turn possess different numbers of PMUs. The PMUs count selected events.

PPE

Pattern Processing Engine

PQM

Prequeue Modifier. An entity in the MPP that can modify a packet.

Predicate Word

32 bits of memory available to a compute engine.

Processing Path

Packets travel along the processing path. The processing path is comprised of a sequence of Virtual Pipeline instances.

Project Configuration

A defined set of XML configuration files within one ASE project. The ASE uses the project configuration to build your project. A build action includes generating header files, compiling programs, scripts, and regular expression rule sets, validating your hardware definitions and software, and generating a binary configuration file. Every ASE project has at least one (default) project configuration.

R

RED

Random Early Detection is a task keep or discard policy that uses a single value (slope) to determine how aggressively to manage the task queue. The slope determines the aggressiveness of the accelerator engine Random Early Discard (RED) policy. Specify a decimal fraction from zero to one (very aggressive, yet even policy). A value less than one also defines a less evenly distributed policy, where a smaller percentage of incoming tasks are refused as the task queue uses more resources.

Register File

256 bytes of memory available to a compute engine.

RFC

Request for Comments, an IETF standard

RLC

Radio Link Control protocol

RN

Random number

RNC

Radio Network Controller

ROC

Rollover counter

RRC

Radio Resource Control

RSA

RSA is an algorithm for public-key cryptography. RSA represents Ron Rivest, Adi Shamir, and Leonard Adleman, who first described the algorithm.

RTCP

Real-time Transport Control Protocol

RTE

The Run-time Environment is a set of APIs and commands that can initialize, configure, and control the Axxia family of communication processors.

The APIs are implemented in ANSI C and are designed to easily port to a variety of different operating systems and microprocessor environments. To use the APIs, include them in a C-language control program, and load the program onto a host processor. The host processor can then access the Axxia device using the supported hardware connections. Use the APIs to perform the following tasks:

- Initialize components of the Axxia device.
- Read and update data structures, such as pattern trees or packets.
- Read and update status information and statistics.
- Read from and write to most control registers and memory.

The commands enable you to access the Axxia device from the command line.

RTP

Real-time Transport Protocol

Rx

Receive

S

SA

Security Association

SAL

The System Abstraction Layer isolates all system-level dependencies to a well-defined set of interfaces that enhance the portability of the RTE software.

SDWRR

Smoothed Deficit Weighted Round Robin, which is a scheduling algorithm that provides a good approximation to weighted fair queuing without incurring the burstiness and latency drawbacks traditionally experienced with deficit weighted round robin scheduling algorithms.

SED

The Stream Editor (SED) accelerator engine that edits 128-byte blocks of either packets or fragments. The SED contains the following functional components:

- The SED load logic merges multiple sets of input parameters into one set and can modify packet data before the SED script runs.
- The compute engine within the SED can run C-NP scripts to modify packet data and set up parameters to modify packet data after the script runs.
- The SED unload logic updates packet data in memory, makes any modified parameter values available to downstream engines, and can modify packet data to append trailer information.

service interrupt

An engine interrupt that signals the host processor that this engine requires service from the CPU. For example, when the NCA completes a DMA operation, it triggers an interrupt that notifies the CPU that the operation is complete.

Simple Virtual Pipeline Instance

A simple Virtual Pipeline instance defines only one engine sequence.

SMON

Statistics Monitor

SMP

Symmetric Multi-processor

SN

Sequence Number

SPI

Security Parameter Index

SPP

Security Protocol Processor

sRIO

Serial Rapid I/O is a point-to-point, packet-switched interconnect technology that uses 8B/10B encoding to stream clock and data signals.

SRTCP

Secure Real-time Transport Control Protocol

SRTP

Secure Real-time Transport Protocol

SSL

Secure Sockets Layer

SSSAR

Segmentation and Reassembly Service Specific Convergence sublayer of the ATM Adaptation Layer (AAL) type 2

Start Engine

A start engine starts or launches a task for an engine sequence and must be first in an engine sequence. (EIOA, MPP, CPU, or Expander).

State Engine

A component within the MPP accelerator engine. The State Engine includes both logic to perform arithmetic and logical calculations for an FPL program running in the MPP and a compute engine to run C-NP scripts. The State Engine is typically used to perform calculations for the FPL program or to maintain state information across more than one run of the FPL program.

T

task

A set of processing parameters that sequential engines use to exchange packet processing information. A start engine creates a task, the intermediate engines use the task, and an end engine consumes the task. When an engine completes its processing for a task, it sends the task to the next engine in the pipeline. A task often contains a reference to a packet and associated parameters to process the packet but can sometimes contain only commands and associated parameters.

task inline parameters

Parameters (32 bytes) used for script processing that arrive with the task. The SED engine loads these parameters when the first block is processed and stores them in the register file for a compute engine script to use for all blocks.

task load balancing

The choice of an individual subengine to perform packet processing based on (a) priority (least number of tasks enqueued for the given priority) or (b) subengine with the least number of tasks enqueued, without considering priority.

task receive queue

An address in memory used by an accelerator engine to maintain one or more task queues that store incoming tasks. Engines assign a 0 to 7 task priority to each queue and use a scheduling algorithm to select which queue is next for service.

TCP

Transmission Control Protocol

TCRI

Task Contents Read Interface

TDI

Task Deletion Interface

TLS

Transport Layer Security

TMGR

Timer Manager

TOID

Task Order ID

Traffic Shaper

A compute engine within the MTM that can run C-NP scripts. Typically, a configuration and an application run scripts in the Traffic Shaper either to choose between the children of a scheduler (arbitration) or to control the traffic of a scheduler or queue to conform to a particular rate (shaping).

Trafgen

A traffic generation utility provided with the ASE that enables you to generate or obtain input traffic for use as you simulate a configuration and application. You can direct Trafgen to either generate traffic internally, based on configuration parameters that you define, or to obtain the traffic externally, from an external file or an external application.

TRNG

True Random Number Generator is used for initial vector generation for key management protocols.

TSI

Task Start Interface

Tx

Transmit

U

UDIMM

Unbuffered Dual-Inline Memory Module

UDP

User Datagram Protocol

UE

User equipment

UTMS

Universal Mobile Telecommunications System

V

VPE

Virtual Pipeline Extension is an Axxia device feature that enables multiple Axxia devices to be efficiently networked or connected directly. An initial device receives a packet, processes it, encodes the resulting output parameters in a packet (using a custom Ethertype value) and sends the packet to another Axxia device. When the downstream Axxia device receives the packet, the EIOA uses the custom EtherType value to recognize a VPE packet, extract the output parameters, and send the packet to a destination (port or another Virtual Pipeline instance).

Virtual Pipeline instance

A set of one or more engine sequences that defines all or part of the packet processing in a configuration and application. Each engine sequence contains a start engine, an end engine, and optional intermediate engines. A configuration and an application can contain one or more Virtual Pipeline instances. Axxia devices support two types of Virtual Pipeline instances: simple and branching.

VLAN

Virtual Local Area Network

W

WCDMA

Wideband Code Division Multiple Access

WRR

Weighted Round Robin is a scheduling algorithm.

X

XGMAC

10-Gigabit Media Access Control, a block within each EIOA.

Storage. Networking. Accelerated.™