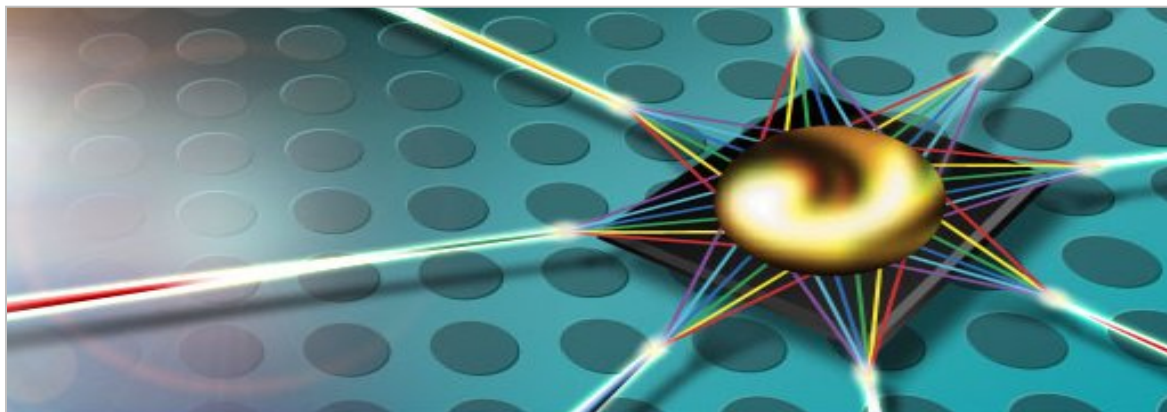




NPS-400 EZcp Reference Manual

Control Plane Library for NPS-400 Network Processors

Document Version 1.9



Document Number: 27-8217-04

The information contained is proprietary and confidential.

Preface

©2015 EZchip Semiconductor Ltd. EZchip is a registered trademark of EZchip Semiconductor Ltd. Brand and product names are trademarks or registered trademarks of their respective holders.

This document contains information proprietary to EZchip and may not be reproduced in any form without prior written consent from EZchip Semiconductor Ltd.

This document is provided on an “as is” basis. While the information contained herein is believed to be accurate, in no event will EZchip be liable for damages arising directly or indirectly from any use of the information contained in this document. All specifications are subject to change without notice.

EZchip Semiconductor Inc.
2700 Zanker Road, Suite 150,
San Jose, CA 95134, USA
Tel: (408) 520-3700, Fax: (408) 520-3701

EZchip Semiconductor Ltd.
1 Hatamar Street, PO Box 527,
Yokneam 20692, Israel
Tel: +972-4-959-6666, Fax: +972-4-959-4166

Email: info@ezchip.com, Web: www.ezchip.com

EZchip welcomes your comments on this publication. Please address them to: supportNP@ezchip.com.

About this Manual

This document describes the EZchip Control Plane Service library (EZcp) and its related APIs. The EZcp library provides an API for control-plane applications for the NPS network processors, abstracting the complexities of the underlying hardware.

This manual is intended for software developers who plan to develop control plane software for products using the EZchip NPS-400 Network Processor. To use this manual, you should be familiar with the network processor architecture.

Related Documents

For additional information refer to:

| DOCUMENT | CONTENT |
|---|---|
| <i>EZdev Reference Manual</i> | Describes the EZchip Device Access Layer library (EZdev) and its related APIs. The EZdev library defines and implements the services required for accessing NPS devices, such as detecting the devices on the PCI Express bus, mapping the devices to the CPU address space, performing memory accesses to the devices and handling interrupt event notifications from the devices. |
| <i>EZenv Reference Manual</i> | Describes the EZchip Environment library (EZenv) and its related APIs. The EZenv library provides a shared runtime infrastructure for all Control Plane Environment (CPE) libraries. |
| <i>CPE Developer's Guide</i> | Describes the various EZcp Control Plane Environment (CPE) components used to develop control-plane applications for NPS-400 based products and how they can be ported to various target platforms. |
| <i>NPS-400 Architectural Specifications</i> | Overview of the architecture, feature set and functionality of the NPS-400 network processor. |

This Document

The following is a brief description of the contents of each section:

| CHAPTER | NAME | DESCRIPTION |
|----------------|--|---|
| Section 1 | Introduction | Provides overview of the EZcp library and its APIs. |
| Section 2 | CP Group | Describes the APIs relating to the EZcp library. |
| Section 3 | Channel Group | Describes the APIs relating to the network processor in general, as well as its C-programmable Task Optimized Processors. |
| Section 4 | Network Interface Group | Describes the APIs relating to the device's network interfaces. |
| Section 5 | Input Classification Unit Group | Describes the APIs relating to configuration of the ICUs. |
| Section 6 | Flow Control Unit Group | Describes the APIs relating to configuration of the FCUs. |
| Section 7 | Traffic Manager Group | Describes the APIs relating to the device's embedded traffic managers. |
| Section 8 | Statistics Group | Describes the APIs relating to the statistics block. |
| Section 9 | TCAM Group | Describes the APIs relating to configuration of the TCAMs. |
| Section 10 | Primitive Group | Describes the APIs relating to reading and writing the registers and memories. |
| Section 11 | Search Structure Group | Describes the APIs relating to the search structures for lookups. |

Revision History

| REVISION | DATE | DESCRIPTION OF MODIFICATION |
|-----------------|---------------|---|
| 1.9 | Sept. 8, 2015 | Relates to EZdk version 1.9a. EZcp API routines, structures and enumerations are not provided in this document; they can be found in the on-line help file accessible from the EZide SDK. Major changes include: The Network Interface Group provides commands to configure SerDes parameters. Statistics Group: Advanced Token Bucket counters renamed Hierarchical Token Buckets. Optimized Hierarchical Token Buckets added. Configuration of Optimized Token Buckets changed. |
| 1.8 | Mar. 31, 2015 | Relates to EZdk version 1.8a. The APIs are preliminary and subject to change. Major changes include: External memory BIST moved from PRM group to Channel group. PCIe interface configuration added to Network Interface group. Statistics commands added to PRM group. |
| 1.7 | Nov. 6, 2014 | Relates to EZdk version 1.7a. The APIs are preliminary and subject to change. |
| 1.6 | July 24, 2014 | Relates to EZdk version 1.6a. The APIs are preliminary and subject to change. |
| 1.5 | Mar. 12, 2014 | Initial release of preliminary draft. Relates to EZdk version 1.5a. The APIs are preliminary and subject to change. |

Terminology and Conventions

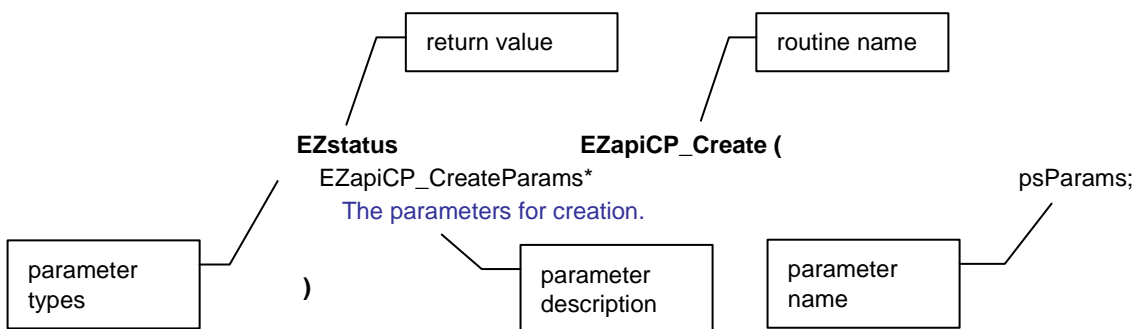
General

The following terminology is used throughout this document:

| TERM | DESCRIPTION |
|---------------------------|--|
| NPS | Refers to the EZchip NPS-400 network processor device and/or software simulator. |
| Channel | Refers to an NPS-400 device and/or simulator in the system |
| Control Plane Application | Refers to a customer-developed application responsible for configuration and management of the NPS device. |
| Control Plane CPU | Refers to the CPU on which the control plane application resides. This may be an external host CPU or the NPS CTOPs. |

Conventions Used for Routines

Routines, or functions/calls, are designated by the keyword EZstatus followed by the function name enclosed in parentheses. Parameters are listed in parentheses.



The following describes terminology for the description of fields and structures.

| TERM | DESCRIPTION |
|----------|---|
| (output) | In the structure field descriptions, the (output) notation may occasionally be used for clarification. It indicates that EZcp outputs data to this field. |

Typographical Conventions

The following typographical conventions are used in this manual. Routine (or function/call) names are written with a parenthesis, e.g. **EZapiCP_Create()**.

Refer to the section or document referenced here for additional information on the topic.

- Set: Indicates the API Set command for configuration.
- Get: Indicates the API Get command for reading.
- Struct: Indicates the relevant API structure for the Get/Set command.
For a Set command, the structure fields are used as input.
For a Get command, the structure fields are used as output.

- ▶ In the structure field descriptions, the (output) notation may occasionally be used for clarification. It indicates that EZcp outputs data to this field.
- ▶ Notes provide additional information that is not necessarily mandatory.

Important: Contains information that is mandatory for proper confirmation and/or operation that should not be overlooked.

Contents

| | |
|---|------------|
| Preface..... | 2 |
| About this Manual..... | 2 |
| Related Documents | 2 |
| This Document..... | 3 |
| Revision History | 3 |
| Terminology and Conventions..... | 4 |
| List of Tables | 8 |
| List of Figures | 8 |
| 1. Introduction..... | 1-1 |
| 1.1 Overview | 1-1 |
| 1.2 API Organization..... | 1-3 |
| 1.2.1 API Groups..... | 1-3 |
| 1.2.2 Routines..... | 1-4 |
| 1.2.3 States | 1-5 |
| 1.3 Version Control..... | 1-6 |
| 1.4 Infrastructures..... | 1-8 |
| 2. CP Group..... | 2-1 |
| 2.1 Overview | 2-1 |
| 2.1.1 CP Library States | 2-1 |
| 2.1.2 Software Event Notification..... | 2-2 |
| 2.1.3 OS Callbacks | 2-2 |
| 2.1.4 Host Memory Manager (HMM)..... | 2-2 |
| 2.2 Summary of Commands – CP Group..... | 2-3 |
| 3. Channel Group..... | 3-1 |
| 3.1 Overview | 3-1 |
| 3.1.1 Channel States | 3-1 |
| 3.1.2 General Parameters..... | 3-4 |
| 3.1.3 Interfaces | 3-4 |
| 3.1.3.1 External Memory Interfaces | 3-4 |
| 3.1.3.2 Network Interfaces | 3-5 |
| 3.1.3.3 Auxiliary Interfaces | 3-7 |
| 3.1.3.4 Management Interfaces..... | 3-7 |
| 3.1.3.5 PCI Express Interface | 3-7 |
| 3.1.4 Memory Partition | 3-8 |
| 3.1.5 Memory Space Parameters..... | 3-9 |
| 3.1.6 Applications | 3-9 |
| 3.1.7 Index Pools..... | 3-9 |
| 3.1.8 PMU | 3-10 |
| 3.1.9 System Info | 3-10 |
| 3.1.10 Protocol Decoder..... | 3-10 |
| 3.1.11 Real Time Clock..... | 3-11 |
| 3.1.12 Soft Memory Errors..... | 3-11 |
| 3.1.13 External Memory BIST | 3-12 |
| 3.1.14 Interrupt Handling..... | 3-12 |
| 3.2 Summary of Commands – Channel Group..... | 3-13 |
| 4. Network Interface Group..... | 4-1 |
| 4.1 Overview | 4-1 |
| 4.1.1 Network Interface States | 4-1 |
| 4.1.2 Configuring Ethernet Interfaces | 4-2 |
| 4.1.2.1 Ethernet Statistic Counters | 4-3 |
| 4.1.3 Configuring Interlaken Interfaces | 4-4 |

| | | |
|-----------|--|------------|
| 4.1.3.1 | Interlaken Interface MAC Configuration | 4-4 |
| 4.1.3.2 | Interlaken Statistic Counters..... | 4-5 |
| 4.1.4 | Configuring RX/TX FIFOs | 4-6 |
| 4.1.5 | Configuring General Parameters | 4-6 |
| 4.1.6 | Configuring the Priority Flow Control Parameters | 4-7 |
| 4.1.7 | Configuring the Out of Band Flow Control Interfaces..... | 4-7 |
| 4.1.8 | Configuring the LED Interface..... | 4-8 |
| 4.1.9 | Configuring SerDes Parameters | 4-8 |
| 4.1.10 | Configuring SMI Interfaces..... | 4-8 |
| 4.1.11 | Auto-negotiation..... | 4-9 |
| 4.2 | Summary of Commands – IF Group | 4-10 |
| 5. | Input Classification Unit Group | 5-1 |
| 5.1 | Overview | 5-1 |
| 5.1.1 | ICU States | 5-1 |
| 5.1.2 | ICU Packet Parsing and Classification..... | 5-1 |
| 5.1.3 | ICU Format Encapsulation..... | 5-2 |
| 5.1.4 | Layer 2 Classification..... | 5-2 |
| 5.1.5 | Layer 3 MPLS Classification | 5-2 |
| 5.1.6 | Layer 3 IP Classification | 5-3 |
| 5.1.7 | Layer 4 TCP and UDP Classification..... | 5-3 |
| 5.1.8 | General Purpose Rules Classification | 5-4 |
| 5.2 | Summary of Commands – ICU Group | 5-5 |
| 6. | Flow Control Unit Group | 6-1 |
| 6.1 | Overview | 6-1 |
| 6.1.1 | FCU States..... | 6-1 |
| 6.1.2 | Configuration of the FCU..... | 6-1 |
| 6.1.3 | Default Configuration of the FCU..... | 6-3 |
| 6.2 | Summary of Commands – FCU Group | 6-5 |
| 7. | Traffic Manager Group..... | 7-1 |
| 7.1 | Overview | 7-1 |
| 7.1.1 | TM States | 7-1 |
| 7.1.2 | TM API Organization..... | 7-2 |
| 7.1.3 | TM Chunk Addressing | 7-5 |
| 7.1.4 | TM Configuration Details | 7-7 |
| 7.1.4.1 | TM Identifier | 7-7 |
| 7.1.4.2 | Global Configuration..... | 7-7 |
| 7.1.4.3 | General Configuration | 7-7 |
| 7.1.4.4 | Scheduling Configuration..... | 7-7 |
| 7.1.4.5 | Shaping Configuration..... | 7-8 |
| 7.1.4.6 | WFQ Configuration..... | 7-10 |
| 7.1.4.7 | Priority Configuration..... | 7-10 |
| 7.1.4.8 | IPG Configuration | 7-10 |
| 7.1.4.9 | WRED Configuration..... | 7-11 |
| 7.1.4.10 | Service Profile Configuration..... | 7-12 |
| 7.1.4.11 | Entity Configuration..... | 7-12 |
| 7.1.4.12 | Topology Configuration | 7-15 |
| 7.1.4.13 | TM Statistics Configuration | 7-15 |
| 7.1.4.14 | TM Output Queue (OQ) Configuration..... | 7-16 |
| 7.2 | Summary of Commands – TM Group..... | 7-17 |
| 8. | Statistics Group | 8-1 |
| 8.1 | Overview | 8-1 |
| 8.1.1 | Statistics States | 8-1 |
| 8.1.2 | On-demand Counters..... | 8-2 |
| 8.1.2.1 | Partition Parameters..... | 8-2 |

| | | |
|------------|---|-------------|
| 8.1.2.2 | Group Types and Counter Types..... | 8-3 |
| 8.1.2.3 | Long Counters | 8-5 |
| 8.1.2.4 | Double Counters | 8-5 |
| 8.1.2.5 | Bitwise Counters | 8-6 |
| 8.1.2.6 | Token Bucket Profiles | 8-7 |
| 8.1.2.7 | Token Bucket Counters | 8-7 |
| 8.1.2.8 | Optimized Token Bucket Counters | 8-8 |
| 8.1.2.9 | Hierarchical Token Bucket Counters | 8-8 |
| 8.1.2.10 | Optimized Hierarchical Token Bucket Counters..... | 8-8 |
| 8.1.2.11 | Watchdog Profiles | 8-9 |
| 8.1.2.12 | Watchdog Counters | 8-9 |
| 8.1.2.13 | On-demand General Purpose Machines | 8-10 |
| 8.1.2.14 | On-demand General Counter Operations | 8-11 |
| 8.1.2.15 | On-demand Messaging Status | 8-11 |
| 8.1.3 | Posted Counters..... | 8-12 |
| 8.1.3.1 | Posted Partition Parameters | 8-12 |
| 8.1.3.2 | Posted Groups..... | 8-13 |
| 8.1.3.3 | Posted Shadow Group Parameters..... | 8-13 |
| 8.1.3.4 | Posted Overflow Profile | 8-13 |
| 8.1.3.5 | Posted GC Profile | 8-14 |
| 8.1.3.6 | Posted Counter Operations | 8-14 |
| 8.1.3.7 | Posted Messaging Status | 8-15 |
| 8.2 | Summary of Commands – Stat Group..... | 8-16 |
| 9. | TCAM Group | 9-1 |
| 9.1 | Overview | 9-1 |
| 9.1.1 | TCAM States..... | 9-1 |
| 9.1.2 | Configuration of the Internal TCAM | 9-1 |
| 9.1.3 | Configuration of the External TCAM | 9-3 |
| 9.2 | Summary of Commands – TCAM Group | 9-4 |
| 10. | Primitive Group | 10-1 |
| 10.1 | Overview | 10-1 |
| 10.1.1 | Register Access | 10-1 |
| 10.1.2 | Memory Access..... | 10-1 |
| 10.1.3 | Performance Module | 10-2 |
| 10.1.4 | Advanced Channel Features..... | 10-2 |
| 10.2 | Summary of Commands – PRM Group | 10-3 |
| 11. | Search Structures Group..... | 11-1 |
| 11.1 | Overview | 11-1 |
| 11.1.1 | Defining Search Structures..... | 11-2 |
| 11.1.2 | Search Memory Management | 11-2 |
| 11.1.3 | Search Structure Entries (Rules) | 11-3 |
| 11.1.4 | Search Structure States | 11-5 |
| 11.2 | Table Structures..... | 11-7 |
| 11.2.1 | Defining Table Structures | 11-7 |
| 11.2.2 | Memory Management | 11-7 |
| 11.2.3 | Operation and Entries | 11-8 |
| 11.3 | Hash Structures..... | 11-9 |
| 11.3.1 | Defining Hash Structures | 11-9 |
| 11.3.2 | Memory Management | 11-10 |
| 11.3.3 | Operation and Entries | 11-11 |
| 11.4 | UltraIP Structures | 11-12 |
| 11.4.1 | Defining UltraIP Structures..... | 11-13 |
| 11.4.2 | Memory Management | 11-13 |
| 11.4.3 | Operation and Entries..... | 11-14 |

| | | |
|------------|---|-------------|
| 11.5 | Algorithmic TCAM..... | 11-15 |
| 11.5.1 | Defining Algorithmic TCAM Structures..... | 11-15 |
| 11.5.2 | Memory Management | 11-16 |
| 11.5.3 | Operation and Entries | 11-16 |
| 11.6 | Summary of Commands – Struct Group | 11-17 |
| 12. | Reference..... | 12-1 |

List of Tables

| | | |
|-------------|---|-------|
| Table 1-1. | Summary of API functional groups | 1-4 |
| Table 1-2. | Basic API routines | 1-4 |
| Table 2-1. | CP Group routines..... | 2-3 |
| Table 2-2. | Summary of CP Group commands | 2-3 |
| Table 3-1. | Channel Group routines | 3-13 |
| Table 3-2. | Summary of Channel Group commands | 3-13 |
| Table 4-1. | Network Interface Group routines | 4-10 |
| Table 4-2. | Summary of Network Interface Group commands | 4-10 |
| Table 5-1. | ICU Group routines..... | 5-5 |
| Table 5-2. | Summary of ICU Group commands | 5-5 |
| Table 6-1. | FCU Group routines..... | 6-5 |
| Table 6-2. | Summary of FCU Group commands..... | 6-5 |
| Table 7-1. | TM Group routines..... | 7-17 |
| Table 7-2. | Summary of TM Group commands | 7-17 |
| Table 8-1. | Matrix of statistic counter groups and types supported | 8-3 |
| Table 8-2. | Statistics Group routines | 8-16 |
| Table 8-3. | Summary of Statistics Group commands..... | 8-16 |
| Table 9-1. | TCAM Group routines..... | 9-4 |
| Table 9-2. | Summary of TCAM Group commands..... | 9-4 |
| Table 10-1. | Primitive Group routines..... | 10-3 |
| Table 11-1. | Enforced vs. non-enforced operations when adding or updating an entry | 11-3 |
| Table 11-2. | Affect of Update Mode and Cache Mode on Table Operations..... | 11-8 |
| Table 11-3. | Search Structures Group routines..... | 11-17 |
| Table 11-4. | Summary of Search Structures Group commands | 11-17 |

List of Figures

| | | |
|--------------|--|-------|
| Figure 1-1. | Overview of the EZcp library | 1-1 |
| Figure 1-2. | API functional groups and basic routines | 1-3 |
| Figure 1-3. | Basic states of objects and the basic API routines..... | 1-5 |
| Figure 2-1. | CP library states..... | 2-1 |
| Figure 3-1. | Channel states and main stages in the configuration of an NPS channel | 3-2 |
| Figure 7-1. | TM API Organization..... | 7-2 |
| Figure 7-2. | Each of the four chunks contains one-quarter of the Subports, Classes and Flows..... | 7-5 |
| Figure 9-1. | Example configuration..... | 9-2 |
| Figure 11-1. | Search Structure states | 11-5 |
| Figure 11-2. | Table hierarchy based on segments of IP address | 11-12 |

1. Introduction

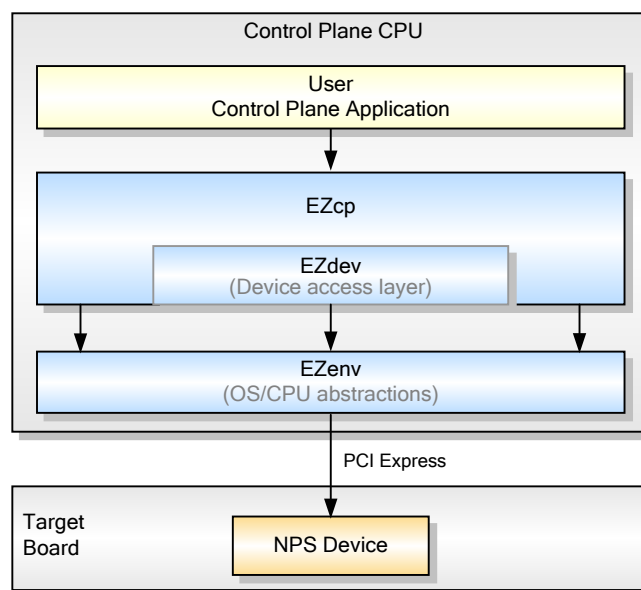
1.1 Overview

The EZchip Control Plane Service library (EZcp) provides an application programming interface (API) for control-plane applications for the NPS network processors, abstracting the complexities of the underlying hardware.

Using the API routines outlined in this document, programmers can write C/C++ language (ANSI) code for control-plane applications operating with the NPS-400 network processors. This includes configuration of the NPS-400 device's CTOPs; configuration of its network interfaces; configuration of its embedded traffic manager; creating and maintaining its lookup structures.

The EZcp API is comprised of a set of routines that are called directly by the application (i.e. user program). The syntax for calling each routine is described in the chapters that follow.

Figure 1-1. Overview of the EZcp library




Portability

The EZcp library is designed to be easily portable to any target platform (OS and/or CPU).

All operating system functionality used by the EZcp library is encapsulated in a separate runtime environment layer (EZenv), allowing customers to easily port OS requirements to any operating system. In addition, the EZcp supports compilation modes for both little-endian and big-endian CPUs, as well as CPUs requiring aligned memory accesses. Furthermore, the EZcp library supports registering user callbacks for several key OS services, such as memory allocation and mutex locking, allowing the control plane application to change or enhance these OS services at runtime.

The device access layer (EZdev) further extends the EZcp library portability by encapsulating the implementation of the platform specific services required for accessing NPS devices, such as reading from a device, writing to a device, and getting interrupt event notifications from a device. The EZcp library is connected to the device access layer (EZdev) using a set of device access callbacks. While the EZcp library is responsible for issuing NPS device access requests, the device access layer provides the platform specific services to execute the requests. The device access layer implementation may operate with the NPS-400 software simulator instead of the real NPS-400 device, using the virtual PCI interface. This is transparent to EZcp, which continues to receive the same services from the device access layer.

 Refer to the *EZenv Reference Manual* and *EZdev Reference Manual* for more information.

Multitasking

The EZcp library is linked with the user-supplied control-plane application software. The EZcp routines are performed in the context of the calling user-application tasks, and do not spawn any tasks of its own.

The EZcp library supports multitasked control-plane applications, with all required synchronization performed internally by the EZcp library. The synchronization of resources accessed by the control-plane application from separate tasks is handled by EZcp with mutexes. Mutexes are used to manage accesses to the network-processor hardware, as well as to manage access to the EZcp's internal software resources. For example, for any given NPS device, there is a mutex protecting the NPS Statistics block's hardware resources from incorrect usage. This mutex is locked by EZcp when required, preventing contention should any two tasks simultaneously attempt to access the Statistics block.

1.2 API Organization

The EZcp API is arranged in groups of routines based on their functionality.

1.2.1 API Groups

The three primary groups are the CP, Channel and Search Structure Groups.

Figure 1-2. API functional groups and basic routines

| | | | |
|---|--|------------------------------------|--|
| <div>High-level configuration used once per system</div> <div>CP Group</div> <div>EZapiCP_Config() EZapiCP_Create() EZapiCP_Delete()</div> <div>EZapiCP_GetVerInfo() EZapiCP_Go() EZapiCP_Status()</div> | | | |
| <div>Channel Group</div> <div>Network processor configuration for each channel in the system</div> <div>EZapiChannel_Config() EZapiChannel_Create() EZapiChannel_Delete() EZapiChannel_Finalize() EZapiChannel_Go() EZapiChannel_Initialize() EZapiChannel_PowerUp() EZapiChannel_Reset() EZapiChannel_Status() EZapiChannel_Stop()</div> | Configures the NPS interfaces in use | Network Interfaces Group | EZapiIF_Config() EZapiIF_Status() |
| | Define the internal input classification units in the system | Input Classification Unit Group | EZapiICU_Config() EZapiICU_Status() |
| | Define the internal flow control units in the system | Flow Control Unit Group | EZapiFCU_Config() EZapiFCU_Status() |
| | Configures the TM topology and QoS functionality | Traffic Manager Group | EZapiTM_Config() EZapiTM_Status() |
| | Configures the statistics counters | Statistics Group | EZapiStat_Config() EZapiStat_Status() |
| | Define the TCAMs in the system | TCAM Group | EZapiTCAM_Config() EZapiTCAM_Status() |
| | Read/write to NPS registers & memories | Primitive Group | EZapiPrm_WriteMem() EZapiPrm_ReadMem() ... |
| <div>Define each data structure (hash, table) in the system</div> <div>Search Structures Group</div> <div>EZapiStruct_Config() EZapiStruct_Status() EZapiStruct_PartitionConfig() EZapiStruct_PartitionStatus()</div> <div>EZapiStruct_AddEntry() EZapiStruct_DeleteEntry() EZapiStruct_GetEntry() EZapiStruct_UpdateEntry() ...</div> | | | |

The **CP Group** of API routines is used for configuration of the EZcp software library and related services, such as OS resources, OS callbacks and host memory management. It is the highest level and is configured once for the entire system.

The **Channel Group** of API routines is designed for the overall configuration of the NPS devices (channels). Due to the large amount of contents, the NPS configuration has been extended into several functional groups. The Channel Group controls the overall channel states, as well as provides configurations for the NPS C-programmable Task Optimized Processors (CTOPs); the **Network Interface Group** provides configuration of the external network interfaces; **Traffic Manager Group** provides configuration of the TM and QoS functionality; the **Statistics Group** enables operation with statistic counters; the **Input Classification Unit Group** provides configuration of the NPS Input Classification Units; the **Flow Control Unit Group** provides configuration of the NPS Flow Control Units; the **TCAM Group** provides configuration of the NPS internal and external lookup mechanisms; and the **Primitive Group** provides routines for low-level services to access the NPS memories and registers.

The **Search Structure Group** of API routines defines and maintains each of the search data structures, such as routing and policy tables, which are used for lookups by the network processors (channels). These routines configure the search structure, its properties, its data entries and memory management. Each search structure is configured independently and may be used for lookups by any or all of the NPS devices/channels in the system.

The EZcp API routines are divided into the following functional groups shown in the table below.

Table 1-1. Summary of API functional groups

| API GROUP | DESCRIPTION |
|--|---|
| CP Group | Routines for configuration of EZcp's overall functionality, such as its interaction with the operating system, memory management, and various modes of operation. |
| Channel Group | Routines for initialization, configuration and maintenance of each channel (NPS) in the system, such as configuration of its CTOPs. |
| Network Interface Group | Routines for configuration of the NPS device's external network interfaces in use. |
| Input Classification Unit Group | Routines for configuration of the NPS device's Input Classification Units (ICUs). |
| Flow Control Unit Group | Routines for configuration of the NPS device's Flow Control Units (FCUs). |
| Traffic Manager Group | Routines for configuration of the NPS device's traffic managers (TMs). |
| Statistics Group | Routines for operating with the NPS device's statistic counter accelerations. |
| TCAM Group | Routines for configuration of the NPS device's internal and external TCAM accelerations. |
| Primitive Group | Routines for low-level services to access the NPS device's memories and registers. |
| Search Structure Group | Routines for configuring and maintaining of search data structures, including their properties, data entries and memory management. |

1.2.2 Routines

For each API group, a basic set of routines – Create(), Go(), Delete(), Config(), Status() – is usually defined as shown in [Figure 1-2](#). In addition to these basic routines, each API group may have its own set of additional routines. As shown in the [Figure 1-2](#) above, all of the API groups have a similar set of routines with a convenient naming convention.

Table 1-2. Basic API routines

| ROUTINE NAME | DESCRIPTION |
|---------------------|---|
| Create() | Creates or defines the object and puts it in the created state. |
| Go() | Starts the objects run and puts it in the running state. |
| Config() | Configures the object. |
| Status() | Queries the objects configuration or runtime status. |
| Delete() | Deletes the object and returns it to the undefined state. |

Config() routines are used for configuration -- API Set operations-- and include a command parameter that has the format EZapiChannel_ConfigCmd_XXX.

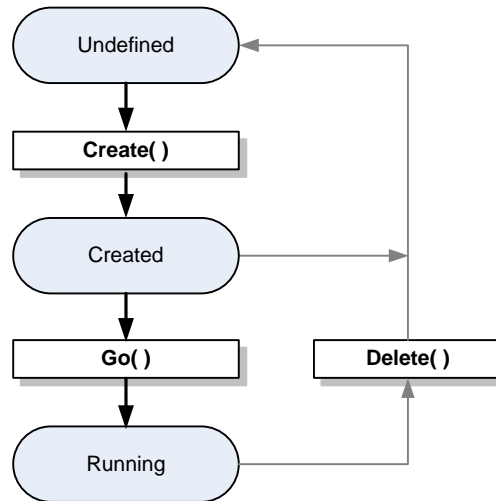
Status() routines are used for reading -- API Get operations -- and include a command parameter that has the format EZapiChannel_StatCmd_YYY.

Both the Config() and Status() routines include a parameter that is pointer to an API structure. For a Config() routines, the structure's fields are used as input, whereas for a Status() routines, the structure's fields are used as output.

1.2.3 States

The EZcp library maintains a software object representing the EZcp itself, as well as objects representing each of the channels and search structures. Objects share a set of basic states (Undefined, Created, and Running). In addition to these basic states, each API group's object may have its own set of additional states. The basic API routines move the object from one state into another as shown in [Figure 1-3](#) below.

Figure 1-3. Basic states of objects and the basic API routines



- ▶ The Create(), Go() and Delete() routines usually have no parameters.
- ▶ The Config() and Status() routines usually have two parameters: the command and the void* pointer parameter which points to an associated structure containing parameters specific to the command. Often the Config() and Status() routines have similar sets of commands (and associated structures) for performing a configuration and querying the existing configuration respectively.
- ▶ A parameter that is a pointer to a "c" structure may sometimes be replaced by NULL. In this case, default values are used for all the structure's fields.
- ▶ Each API group is defined in a separate "c" header file.

1.3 Version Control

The EZcp API includes a version control mechanism used for ensuring backward compatibility of control plane application code with subsequent EZcp version updates.

Each structure in the EZcp API routines begins with a 4-byte version field named *eVersion* of the following enumerated type:

```
typedef enum _EZapiCP_Version
{
    EZapiCP_Version_1_5 = 0x00000105,
    /* Use version 1.5. */
    EZapiCP_Version_1_6 = 0x00000106,
    /* Use version 1.6. */
    ...
} EZapiCP_Version;
```

When initially developing the control plane application code, the application uses the version field to specify the version of EZcp for which the code was written and verified (according to the version of EZcp used at the time of development/verification).

When upgrading the control plane application to use a newer EZcp version, the control plane application code is left unchanged. When an API call is invoked for a structure where new functionality was added, EZcp checks the version field of the structure, and accordingly executes code to ensure that the behavior of the API call is handled in a manner which is backward compatible to that which existed for the previous version (see example below).

When the control plane application needs to be enhanced to use new functionality (supported from a later EZcp version), the control plane application is updated to explicitly initialize the new API fields, and the *eVersion* field of the structure is updated to explicitly specify the new EZcp version for which the new code was developed/verified.

- ▶ *The control plane application should explicitly initialize the *eVersion* field of each API call to the EZcp version at the time when the code is written/verified.*
- ▶ *If the *eVersion* field of a structure does not match a valid version number (according to the version enum), the version is considered to be 1.5.*
- ▶ *The version enum includes a value of “EZapiCP_Version_Latest” which can be used to ensure that latest functionality is used at all times. It is not recommended for control plane applications to use this value, as it does not ensure backward compatibility handling when moving to subsequent EZcp versions.*
- ▶ *In addition to the version control mechanism, it is recommended (but not mandatory) to perform a memset operation on API software structures before initializing field values and passing them to the EZcp API calls.*

Example:

In the initial version (1.5), the structure XXX was defined as follows:

```
typedef struct __XXX
{
    EZapiCP_Version      eVersion;
    /* The version of the structure. */
    EZbool               bEnable1;
    /* Enables/disables option 1.
     * Default value is FALSE. */
    EZbool               bEnable2;
    /* Enables/disables option 2.
     * Default value is FALSE. */
} EZapiTM_WREDParams;
```

Thus, the following control plane application code was developed to call routine `YYY` and pass it the structure `XXX`:

```
XXX sStructParams;  
  
sStructParams.eVersion = EZapiCP_Version_1_5;  
sStructParams.bEnable1 = FALSE;  
sStructParams.bEnable2 = TRUE;  
  
retVal = YYY(&sStructParams);
```

In version 1.6, if a new feature was introduced (option 3): In order to support the new feature, the `XXX` structure was enhanced to include an additional field, `bEnable3`:

```
typedef struct _XXX  
{  
    EZapiCP_Version      eVersion;  
    /* The version of the structure. */  
    EZbool               bEnable1;  
    /* Enables/disables option 1.  
     * Default value is FALSE. */  
    EZbool               bEnable2;  
    /* Enables/disables option 2.  
     * Default value is FALSE. */  
    EZbool               bEnable3;  
    /* Enables/disables option 3.  
     * Relevant from version 1.5  
     * Default value is FALSE. */  
} XXX;
```

The control plane application is then upgraded to use the new EZcp (version 1.6). Initially, the control plane application code is left unmodified (as above).

As part of the implementation of the new feature in EZcp, the 1.6 code for the `YYY` routine checks the `eVersion` field on the `XXX` structure. If the `eVersion` field is earlier than 1.6, then EZcp identifies that the code was originally written without taking into account the new field. As a result, EZcp will internally initialize the field to a backward compatible value (in this case, the `bEnable3` field will be set to `FALSE` internally by EZcp).


When the control plane application wants to use the enhanced functionality, the control plane application code should be modified to explicitly initialize the new API fields and update the `eVersion` field of the structure as follows:

```
XXX sStructParams;  
  
sStructParams.eVersion = EZapiCP_Version_1_6;  
sStructParams.bEnable1 = FALSE;  
sStructParams.bEnable2 = TRUE;  
sStructParams.bEnable3 = TRUE;  
  
retVal = YYY(&sStructParams);
```

Now, when the EZcp code is called, it identifies that the code was updated to take into account the new field. As a result, the EZcp will use the value of the new field specified by the control plane application (no backward compatibility handling will be invoked).

1.4 Infrastructures

The EZcp library uses the shared runtime infrastructure supplied by the EZenv library. This includes the return code infrastructure, the logging/tracing infrastructure, and the development level infrastructure.

 Refer to the *EZenv Reference Manual* for more information on the common infrastructures.

2. CP Group

The CP Group contains routines for configuration and management of the Control Plane library, such as its interaction with the operating system and various modes of operation.

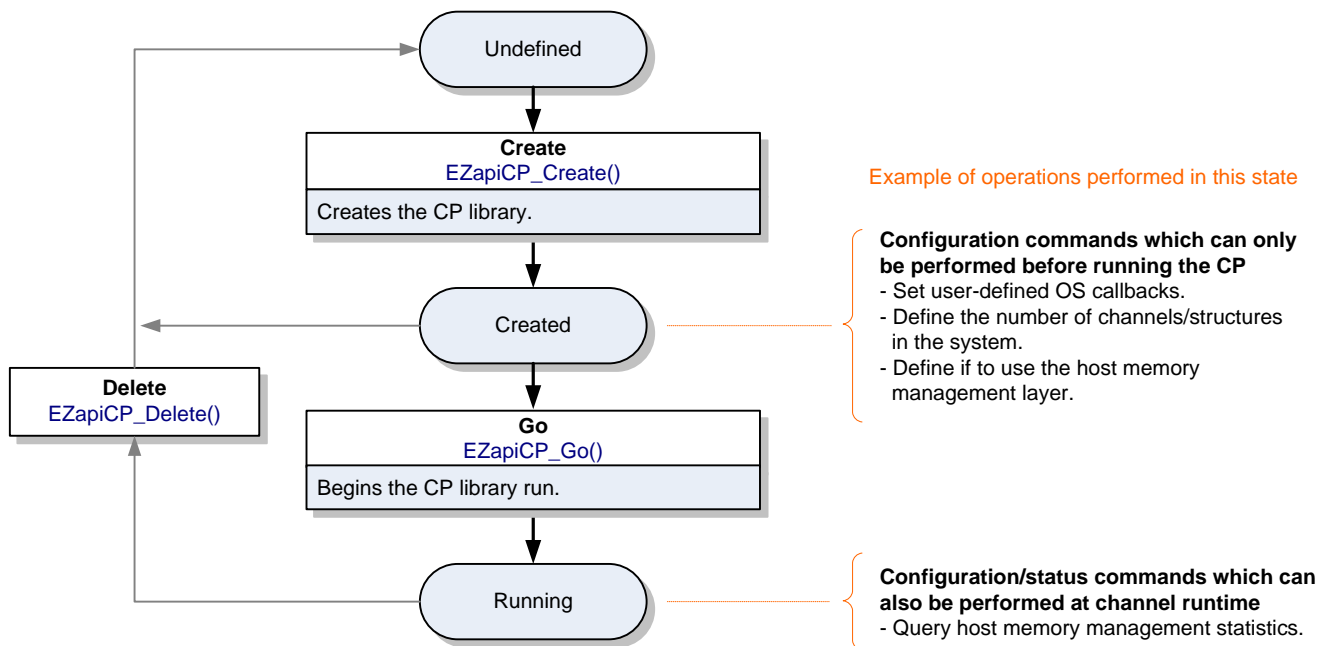
2.1 Overview

Following are more detailed explanations regarding several CP Group API related topics.

2.1.1 CP Library States

The following diagram illustrates the CP library's states:

Figure 2-1. CP library states



The CP library is initially in the undefined state.


The **EZapiCP_Create()** routine creates the CP software library. This routine does not require any OS resources. All the OS resource allocations are postponed until the **EZapiCP_Go()** routine, allowing user-defined OS callbacks before any OS resources are used (see the “[OS Callbacks](#)” section). After this routine, the CP library is in the created state.

The **EZapiCP_Go()** routine begins the CP library run. After this routine, the CP library is in the running state.

The **EZapiCP_Delete()** routine stops the CP library, freeing all allocated resources. This call also internally deletes all defined channels and/or search structures and their related resources. After this routine, the CP library is in the undefined state.

- ▶ The CP Group API routines control the CP library. They only affect the CP library's software state, and do not perform accesses to the NPS hardware devices.
- ▶ The CP library must be in running state before other API groups' routines can be performed.


📖 For a summary of which CP Group commands can be called during which states, see “[Summary of Commands – CP Group](#)”.

 The existing CP library state may be queried in the following manner:


Get: EZapiCP_StatCmd_GetStateInfo
Struct: EZapiCP_StateInfo

2.1.2 Software Event Notification

The CP library provides services for customer-developed software event handlers. Software event handlers notify user applications at key points in time, such as the creation of each channel, deletion of each channel, etc. This allows users to develop hooks to handle these events, without needing to alter the control plane application generating these events.


 Customer-developed software event handler configurations are done in the following manner:

Set: EZapiCP_ConfigCmd_SetEventCallbacks
Get: EZapiCP_StatCmd_GetEventCallbacks
Struct: EZapiCP_EventCallbacks

 For more information, refer to the *CPE Developer's Guide*.

2.1.3 OS Callbacks

The CP library supports registering user callbacks for several key OS services, such as memory allocation and mutex locking. This allows the control plane application to change/enhance these OS services during runtime.

 OS callback service configurations are done in the following manner:

Set: EZapiCP_ConfigCmd_SetOSCallbacks
Get: EZapiCP_StatCmd_GetOSCallbacks
Struct: EZapiCP_OSCallbacks


2.1.4 Host Memory Manager (HMM)

Some operating systems may exhibit large performance penalties when large amounts of memory allocations are performed. To handle such cases, the CP library contains a host memory management (HMM) layer which handles host memory allocations requests, while maintaining high performance.

The HMM layer provides host memory allocation and recycling services, while significantly reducing performance penalties. These services are provided in an OS-independent manner. In addition, the HMM layer prevents fragmentation of host memory which can significantly affect performance in real-time environments.

The HMM layer may be enabled in the CP Group's initialization phase. Once enabled, the CP library directs memory allocation requests to the HMM layer, instead of to the standard OS callbacks. The HMM layer itself continues to use the OS callbacks when needed.

The HMM layer allocates several large memory pools from the operating system, divides them into buffers of pre-determined sizes, and handles the buffer allocations and releases. The amount of each of the buffers is dynamic, and can grow during the system run-time, according to the growth in the memory consumption.

 The HMM layer also supports an API to query the current memory usage status (allocated/used), as well as to query statistics of memory usage over time (peak usage, total requests, etc.).

Get: EZapiCP_StatCmd_GetHMMStatus
Struct: EZapiCP_HMMStatus

2.2 Summary of Commands – CP Group

Table 2-1. CP Group routines

| API Routines | |
|-------------------|---------------------------|
| EZapiCP_Config() | EZapiCP_GetVersionInfo() |
| EZapiCP_Create() | EZapiCP_Go() |
| EZapiCP_Delete() | EZapiCP_Status() |

Table 2-2. Summary of CP Group commands

This table provides a summary of the commands in the CP Group, as well as the EZcp library state in which each command may be performed.

| | Undefined | Created | Running |
|---|-----------|----------|----------|
| <i>TOPIC & COMMAND</i> | <i>U</i> | <i>C</i> | <i>R</i> |
| EZcp Library Configuration | | | |
| General configurations | | | |
| Set: EZapiCP_ConfigCmd_SetGeneralParams | - | ✓ | - |
| Get: EZapiCP_StatCmd_GetGeneralParams | - | ✓ | ✓ |
| Software event notification configuration | | | |
| Set: EZapiCP_ConfigCmd_SetEventCallbacks | - | ✓ | ✓ |
| Get: EZapiCP_StatCmd_GetEventCallbacks | - | ✓ | ✓ |
| OS callback configuration | | | |
| Set: EZapiCP_ConfigCmd_SetOSCallbacks | - | ✓ | ✓ |
| Get: EZapiCP_StatCmd_GetOSCallbacks | - | ✓ | ✓ |
| Memory status | | | |
| Get: EZapiCP_StatCmd_GetHMMStatus | - | ✓ | ✓ |
| EZcp library state | | | |
| Get: EZapiCP_StatCmd_GetStateInfo | ✓ | ✓ | ✓ |

3. Channel Group

The Channel Group contains routines for creating, configuring and managing each NPS in the system.

3.1 Overview

Each EZchip NPS network processor in the system is considered a channel. The Channel Group routines enable initializing the channel, configuring the channel's internal memory, external memory and network interfaces, memory management, configuring and managing the CTOPs on the channel, and other channel-related operations.

The API also supports dynamic deletion of existing channels, addition of new channels and replacement of existing channels for maintenance purposes.

Default values are configured for most channel configuration parameters. Thus, there is no need to explicitly configure these items, unless defaults do not match required behavior. When changing configurations, it is recommended to first perform a get operation to retrieve the previous values, modify only the fields whose value needs to be modified to attain the required behavior, and then perform a set operation with the updated software structure.

Following are more detailed explanations regarding several Channel Group API related topics.

3.1.1 Channel States

There is a channel object representing each of the NPS devices in the system. [Figure 3-1](#) below illustrates the channel object's states.

The lifetime of an NPS channel is comprised of the following main stages:

A channel object is initially in the **Undefined** state.

The **EZapiChannel_Create()** routine defines a new channel (NPS device). This routine defines the channel software object, locates the device in the PCIe bus configuration space and maps the device to the CPU memory space. After this routine, the channel is in the created state.

While in the **Created** state, the application performs the system configuration commands, defining parameters for the NPS's network interfaces and external memory interfaces. The application may also perform configuration commands which affect the power up sequence. In addition, the application may perform configuration commands which affect the NPS default initialization.

The **EZapiChannel_PowerUp()** routine performs the power up sequence on the NPS device, taking the NPS device out of reset state, as well as configuring its network interfaces and external memory interfaces. After this routine, the channel is in the powered-up state.

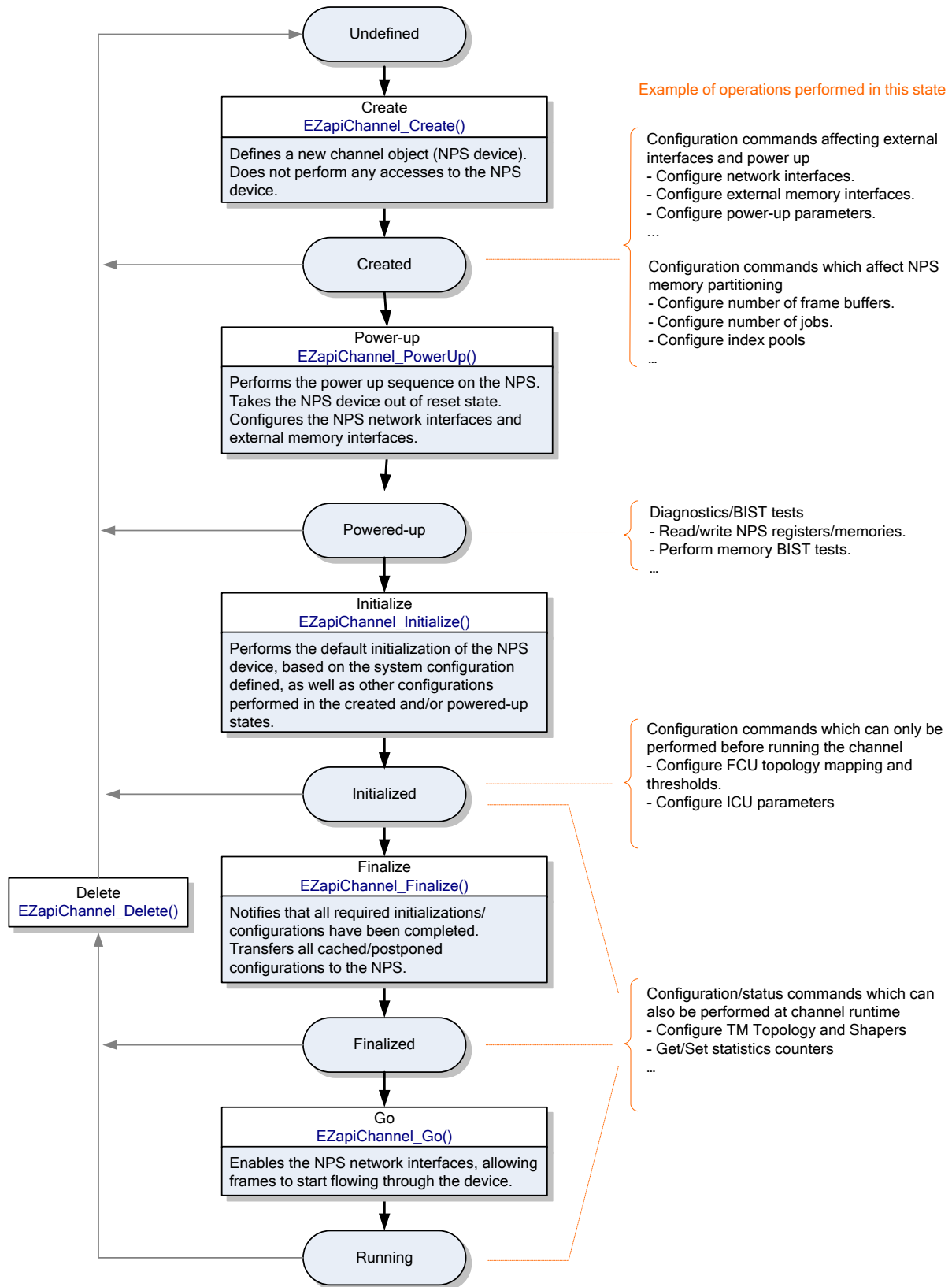
While in the **Powered-Up** state, the application may access NPS registers and memories using the Primitive API Group routines, allowing it to perform diagnostic tests before the initialization of the NPS memories.

The **EZapiChannel_Initialize()** routine performs the default initialization of the NPS device, based on the system configuration defined, as well as other configurations performed in the created state. After this routine, the channel is in the initialized state.

While in the **Initialized** state, the application may perform configuration commands which cannot be performed at channel runtime (configurations that must be performed in the initialized state).

(continued)

Figure 3-1. Channel states and main stages in the configuration of an NPS channel



The **EZpiChannel_Finalize()** routine notifies the EZcp library that all required initializations/configurations have been completed. When called, any postponed validations are performed and any cached configurations are transferred to the NPS. After this routine, the channel is in the finalized state.

The **Finalized** state allows performing further configuration changes after all deferred configurations have been transferred to the NPS, but before the channel is transferred into the running state.


The **EZapiChannel_Go()** routine enables various hardware machines within the NPS as well as the NPS's network interfaces, allowing frames to start flowing through the NPS device. After this routine, the channel is in the running state.


While in the **Running** state, the application may perform additional configuration commands which may also be performed at channel runtime. These configurations can usually also be performed in the initialized and finalized states.

The **EZapiChannel_Delete()** routine deletes the channel, freeing all resources used by the channel object. This call does not perform any accesses to the NPS device (the NPS device is left unchanged). In addition, this call does not affect any other channel objects. Any search structure objects mapped to this channel are detached from the channel but not deleted (see "[Search Structure Group](#)" for more information). After this routine, the channel is in the undefined state.

The **EZapiChannel_Stop()** routine disables the NPS's network interfaces and stops the various hardware machines within the NPS thus stopping all activity within the NPS. After this routine, the channel is in the finalized state.

- ▶ *The EZapiChannel_Create() routine can only be used when the EZcp library is in the running state (e.g. after calling the EZapiCP_Go() API routine).*
- ▶ *The channel object's state also affects the Network Interface, Traffic Manager, Statistics, ICU, FCU TCAM and Primitive API Groups. These groups are an extension of the Channel Group, and thus do not have states of their own, but rather share the channel object's states. For more information on the affects of the channel state on these API groups, see the "States" section in each of the relevant group sections of this document.*
- ▶ *The channel object must be initialized before loading memory partition to it.*
- ▶ *Calling the EZapiChannel_Initialize() routine will internally invoke the power-up sequence if not already invoked.*

 For a summary of which Channel Group commands can be called during which states, see "[Summary of Commands – Channel Group](#)".

 The existing channel state may be queried in the following manner:

Get: EZapiChannel_StatCmd_GetStateInfo
Struct: EZapiChannel_StateInfo

3.1.2 General Parameters

General parameters configure additional modes/parameters which are required for the NPS power up sequence, or have a major affect on system behavior and/or configurations, such as:

- System clock frequency
- Number of internal and external frame buffers
- Number of job buffers
- General SerDes lane related configuration: normal or reduced lanes mode, lane routing

 General parameters are configured in the following manner:

Set: EZapiChannel_ConfigCmd_SetGeneralParams
 Get: EZapiChannel_StatCmd_GetGeneralParams
 Struct: EZapiChannel_GeneralParams

In addition, the channel group includes command to get/check NPS device information, such as its phase.


 Device information is retrieved in the following manner:


Get: EZapiChannel_StatCmd_GetDeviceInfo
 Struct: EZapiChannel_DeviceInfo

3.1.3 Interfaces

The channel group provides routines for configuring the NPS's various external interfaces.

Network Interface configuration parameters must be configured when the channel is in the created state (before performing the power up sequence to the NPS device).

 Refer to *NPS-400 Hardware Reference Manual* for a complete definition of supported combinations of external interfaces.

 Additional configuration of the interfaces is provided in the Network Interface group.

3.1.3.1 External Memory Interfaces

The channel group provides routines for configuring the NPS device's external memory interfaces.

 External memory interface parameters are configured in the following manner:

Set: EZapiChannel_ConfigCmd_SetExtMemParams
 Get: EZapiChannel_StatCmd_GetExtMemParams
 Struct: EZapiChannel_ExtMemParams

Set: EZapiChannel_ConfigCmd_SetGCIIFParams
 Get: EZapiChannel_StatCmd_GetGCIIFParams
 Struct: EZapiChannel_GCIIFParams

Set: EZapiChannel_ConfigCmd_SetInterlakenLAIFParams
 Get: EZapiChannel_StatCmd_GetInterlakenLAIFParams
 Struct: EZapiChannel_InterlakenLAIFParams

3.1.3.2 Network Interfaces

The channel group provides routines for configuring the NPS's Ethernet and Interlaken network interfaces:

- 2x 48x 10GbE XFI/SFI/10GBASE-KR
- 2x 12x 40GbE XLAUI/40GBASE-KR4
- 2x 4x 100GbE CAUI
- 2x Interlaken

 Configuration of the [PCIe](#), [GCI](#) and [Interlaken-LookAside](#) interfaces is provided in separate sections.


The network interface parameters configuration command configures basic information for each interface, such as:

- Enable/disable the interface for RX/TX.
 - ▶ *By default, all external network interfaces are disabled. The [control plane](#) application must explicitly define the external network interfaces used for each channel.*
 - ▶ *For Ethernet interfaces, RXEnable and TXEnable should be symmetric. For Interlaken, RXEnable and TXEnable can be asymmetric.*
 - ▶ *Enabled network interfaces mapped to the same pair of lanes must be of the same interface type.*
- The number of physical lanes used for RX/TX.
 - ▶ *For Ethernet interfaces, the number of RX channels is determined by the interface's flow control mode. When priority-based flow control mode is enabled, there is one RX channel. When class-based flow control mode is enabled, there are up to 8 RX channels. Up to 8 TX channels are defined.*
 - ▶ *For Interlaken interfaces, the number of lanes for RX and TX may differ and can vary between 1 and 48 lanes.*
- The number of channels used for RX/TX.
 - ▶ *For Ethernet interfaces, one RX channel is defined and up to 8 TX channels are defined.*
 - ▶ *For Interlaken interfaces, up to 128 channels are supported.*
- The interface's flow control (FC) method
 - Expected RX FC method: link level (LL), priority based (PFC) or class based (CBFC).
 - TX FC method: link level (LL), priority based (PFC) or class based (CBFC).
- The interface's SerDes data rate.

When the channel is initialized, all NPS configurations are set to default values. The network interface parameters affect several of these default configurations, such as:

- The allocation of RX/TX data FIFOs for the interfaces.
- The mapping of ports to the FCU budget ID counters.

These default configurations may later be changed using the specific API commands in the various API groups.

 Network interface parameters are configured in the following manner:

```
Set:      EZapiChannel_ConfigCmd_SetEthIFParams
Get:      EZapiChannel_StatCmd_GetEthIFParams
Struct:   EZapiChannel_EthIFParams

Set:      EZapiChannel_ConfigCmd_SetEthRXChannelParam
```


Get: EZapiChannel_StatCmd_GetEthRXChannelParams
Struct: EZapiChannel_EthRXChannelParams

Set: EZapiChannel_ConfigCmd_SetEthTXChannelParam
Get: EZapiChannel_StatCmd_GetEthTXChannelParams
Struct: EZapiChannel_EthTXChannelParams

Set: EZapiChannel_ConfigCmd_SetInterlakenIFParams
Get: EZapiChannel_StatCmd_GetInterlakenIFParams
Struct: EZapiChannel_InterlakenIFParams

Set: EZapiChannel_ConfigCmd_SetInterlakenRXChannelParams
Get: EZapiChannel_StatCmd_GetInterlakenRXChannelParams
Struct: EZapiChannel_InterlakenRXChannelParams


Set: EZapiChannel_ConfigCmd_SetInterlakenTXChannelParams
Get: EZapiChannel_StatCmd_GetInterlakenTXChannelParams
Struct: EZapiChannel_InterlakenTXChannelParams

Set: EZapiChannel_ConfigCmd_SetRXAllocationProfile
Get: EZapiChannel_StatCmd_GetRXAllocationProfile
Struct: EZapiChannel_RXAllocationProfile

3.1.3.3 Auxiliary Interfaces

The channel group provides routines for configuring the following auxiliary interfaces:

- 2x 8 loopback ports
- 2x 1 timestamp confirmation port

 Auxiliary interface parameters are configured in the following manner:

| | |
|---------|--|
| Set: | EZapiChannel_ConfigCmd_SetLoopbackIFParams |
| Get: | EZapiChannel_StatCmd_GetLoopbackIFParams |
| Struct: | EZapiChannel_LoopbackIFParams |
| Set: | EZapiChannel_ConfigCmd_SetTSConfirmationIFParams |
| Get: | EZapiChannel_StatCmd_GetTSConfirmationIFParams |
| Struct: | EZapiChannel_TSConfirmationIFParams |

3.1.3.4 Management Interfaces


The channel group provides routines for configuring the management network interfaces. Management network interfaces are used to supply network connectivity for the Linux OS running on the CTOPs.

 Management network interface parameters are configured in the following manner:


| | |
|---------|--|
| Set: | EZapiChannel_ConfigCmd_SetMngNetIFParams |
| Get: | EZapiChannel_StatCmd_GetMngNetIFParams |
| Struct: | EZapiChannel_MngNetIFParams |

3.1.3.5 PCI Express Interface


The Channel Group includes commands to configure the PCI Express interfaces and retrieve the PCI configuration space information.

 PCI Express interface configuration is done in the following manner:


| | |
|---------|--|
| Set: | EZapiChannel_ConfigCmd_SetPCIeIFParams |
| Get: | EZapiChannel_StatCmd_GetPCIeIFParams |
| Struct: | EZapiChannel_PCIeIFParams |

 PCIe memory space parameter configuration is done in the following manner:

| | |
|---------|--|
| Set: | EZapiChannel_ConfigCmd_SetPCIeMemSpaceParams |
| Get: | EZapiChannel_StatCmd_GetPCIeMemSpaceParams |
| Struct: | EZapiChannel_PCIeMemSpaceParams |

 PCIe physical function parameter configuration is done in the following manner:

| | |
|---------|--|
| Set: | EZapiChannel_ConfigCmd_SetPCIePhysicalFuncParams |
| Get: | EZapiChannel_StatCmd_GetPCIePhysicalFuncParams |
| Struct: | EZapiChannel_PCIePhysicalFuncParams |


 PCIe message queue parameter configuration is done in the following manner:

| | |
|---------|--|
| Set: | EZapiChannel_ConfigCmd_SetPCIeMsgQueueParams |
| Get: | EZapiChannel_StatCmd_GetPCIeMsgQueueParams |
| Struct: | EZapiChannel_PCIeMsgQueueParams |

 PCIe message queue event parameter configuration is done in the following manner:

| | |
|------|---|
| Set: | EZapiChannel_ConfigCmd_SetPCIeMsgQueueEventParams |
| Get: | EZapiChannel_StatCmd_GetPCIeMsgQueueEventParams |

Struct: EZapiChannel_PCIMsgQueueEventParams

 PCI configuration space information is retrieved in the following manner:

Get: EZapiChannel_StatCmd_GetPCIInfo

Struct: EZapiChannel_PCInfo

3.1.4 Memory Partition

The NPS has an integrated internal memory of 16 megabytes.

The internal memory (IMEM) is shared between the following clients:

- Internal frame data buffers and multicast counters
- Internal job descriptors
- Internal TM resources (QD and PD)
- Internal Search structures
- Data-plane application code and data and stack
- System Information

In addition, the NPS utilizes external DDR3/DDR4 and GCI memory devices.

The external memory (EMEM) is shared between the following clients:


- External frame data buffers, multicast counters and buffer pools
- External TM resources (QD and PD)
- External Search structures
- External Statistics counters (on-demand and posted)
- Data-plane application data and stack
- Linux OS code and data
- Index pools

Users control the memory division/allocation using various APIs:


- General Parameters (EZapiChannel_GeneralParams) – number of internal memory buffers, number of external memory buffers, and number of job buffers.
- Internal Memory Space Parameters (EZapiChannel_IntMemSpaceParams)
- External Memory Space Parameters (EZapiChannel_ExtMemSpaceParams)
- Application Parameters (EZapiChannel_ApplicationParams) – internal/external stack size, private data size
- Index Pool Parameters (EZapiChannel_IndexPoolParams)
- Statistic Partition and Group definitions (EZapiStat_PartitionParams, EZapiStat_GroupParams, EZapiStat_PostedPartitionParams and EZapiStat_PostedGroupParams)
- TM partition definitions (EZapiTM_GlobalParams and EZapiTM_GeneralParams).

3.1.5 Memory Space Parameters

The channel group provides routines for configuring internal and external memory spaces used by the data-plane applications.

 Internal memory space parameters are configured in the following manner:

Set: EZapiChannel_ConfigCmd_SetIntMemSpaceParams
Get: EZapiChannel_StatCmd_GetIntMemSpaceParams
Struct: EZapiChannel_IntMemSpaceParams


 External memory space parameters are configured in the following manner:

Set: EZapiChannel_ConfigCmd_SetExtMemSpaceParams
Get: EZapiChannel_StatCmd_GetExtMemSpaceParams
Struct: EZapiChannel_ExtMemSpaceParams

3.1.6 Applications

The Channel Group includes commands to configure the NPS data-plane applications.

An application defines a group of NPC clusters which share the same memory definitions and service the same types of frames via the PMU schedulers.


 The application parameters are configured in the following manner:

Get: EZapiChannel_ConfigCmd_SetApplicationParams
Set: EZapiChannel_StatCmd_GetApplicationParams
Struct: EZapiChannel_ApplicationParams


3.1.7 Index Pools

The Channel Group includes commands to configure the NPS index pools. Index pools are used by data-plane applications to implement high-performance dynamic resource pools, as well as for the hash search structures' dynamic pointer queues (for hash structures updated by the data-plane application).

The NPS supports 64 index pools (32 on each side). Alternatively, each two matching pools on each side may be combined to operate in shared mode to achieve higher performance.


 The index pool parameters are configured in the following manner:

Get: EZapiChannel_ConfigCmd_SetIndexPoolParams
Set: EZapiChannel_StatCmd_GetIndexPoolParams
Struct: EZapiChannel_IndexPoolParams


 See "[Memory Management](#)" the *Search Structures Group* chapter for more information on hash structures.

3.1.8 PMU

The Channel Group includes commands to configure and operate with the NPS Processor Management Unit (PMU). The PMU serves as the job scheduler. Jobs are created by different events: packet arrival, PMU timer expiration, multicast and more. Each job is inserted into one of the PMU queues and scheduled by the PMU application scheduler for processing by the cluster's CTOPs.

 PMU configurations are done in the following manner:

| | |
|---------|--|
| Set: | EZapiChannel_ConfigCmd_SetPMUQueueParams |
| Get: | EZapiChannel_StatCmd_GetPMUQueueParams |
| Struct: | EZapiChannel_PMUQueueParams |
| Set: | EZapiChannel_ConfigCmd_SetPMUSchedulerParams |
| Get: | EZapiChannel_StatCmd_GetPMUSchedulerParams |
| Struct: | EZapiChannel_PMUSchedulerParams |
| Set: | EZapiChannel_ConfigCmd_SetPMUTimerParams |
| Get: | EZapiChannel_StatCmd_GetPMUTimerParams |
| Struct: | EZapiChannel_PMUTimerParams |
| Set: | EZapiChannel_ConfigCmd_SetPMUTimerControl |
| Get: | EZapiChannel_StatCmd_GetPMUTimerControl |
| Struct: | EZapiChannel_PMUTimerControl |
| Set: | EZapiChannel_ConfigCmd_SetPMUJobLimitProfileParams |
| Get: | EZapiChannel_StatCmd_GetPMUJobLimitProfileParams |
| Struct: | EZapiChannel_PMUJobLimitProfileParams |

 PMU statuses can be retrieved by the following manner:

| | |
|---------|--|
| Get: | EZapiChannel_StatCmd_GetPMUStatus |
| Struct: | EZapiChannel_PMUStatus |
| Get: | EZapiChannel_StatCmd_GetPMUQueueStatus |
| Struct: | EZapiChannel_PMUQueueStatus |

3.1.9 System Info

The Channel Group includes a command to configure the NPS system information. The system info feature is used to transfer PMU and BMU information to the data plane application. It also includes the PMU queues status and related flow control statuses.

| | |
|---------|--|
| Get: | EZapiChannel_StatCmd_GetSystemInfoParams |
| Set: | EZapiChannel_ConfigCmd_SetSystemInfoParams |
| Struct: | EZapiChannel_SystemInfoParams |

3.1.10 Protocol Decoder


The Channel Group includes commands to configure the frame protocol decoder hardware accelerators.

 Protocol decoder configurations are done in the following manner:

| | |
|---------|---|
| Set: | EZapiChannel_ConfigCmd_SetProtocolDecoderParams |
| Get: | EZapiChannel_StatCmd_GetProtocolDecoderParams |
| Struct: | EZapiChannel_ProtocolDecoderParams |

3.1.11 Real Time Clock

The Channel Group includes commands to configure and operate with the NPS accurate real-time clock (RTC).

 RTC configuration and execution are done in the following manner:

| | |
|---------|--|
| Set: | EZapiChannel_ConfigCmd_SetRTCParams |
| Get: | EZapiChannel_StatCmd_GetRTCParams |
| Struct: | EZapiChannel_RTCParams |
| Set: | EZapiChannel_ConfigCmd_ExecuteRTCOperation |
| Struct: | EZapiChannel_RTCOperationParams |

3.1.12 Soft Memory Errors


The soft memory errors (SER) are categorized into three types:

- **Transient** – Errors that may cause a transient effect and will be recovered automatically by the HW without the need for SW intervention.
Example: An external memory Frame Data error will cause a specific frame to be dropped, however it will not cause to any other errors in subsequent frames.
- **Persistent Fixable** – Errors that cannot be automatically fixed by the HW, but are fixable by EZcp SW. These errors are double bit ECC errors or parity errors in which HW cannot indicate the defective bits and, therefore, the EZcp writes the whole memory line from its cache.
Example: Error in configuration SRAMs (e.g. TM profiles, weights, SDTs)
- **Persistent Non-recoverable** – Errors that may be persistent or non-persistent but once occurred have caused the device to go into an unknown state. Based on the block ID and type of error, fast reset or full reset may be required.
Example: Double bit errors on control, pointers, etc.

The Channel Group API provides routines for detecting and handling soft memory errors (SER).

EZapiChannel_StatCmd_GetSERInfo returns information on a soft memory error (ECC or parity) such as the NPS block ID, the memory ID and the error line. In addition, the routine returns the recommended handling — fix the error (call **EZapiChannel_ConfigCmd_FixSER**), ignore the error (transient error), or reset the NPS device. The routine can be invoked to locate the first error (e.g. any error), or to check if an error exists in a specific NPS block and memory ID. To detect several errors, the control plane application can call this routine repeatedly until no more errors are found (e.g. `eErrorType` is `EZapiChannel_SERType_NO_ERROR`). This routine can be invoked either in polling mode or as a result of a single/double ECC error interrupt.

EZapiChannel_ConfigCmd_FixSER is used to fix ECC/parity errors detected by the command **EZapiChannel_StatCmd_GetSERInfo**. This routine should be passed the error information structure returned by **EZapiChannel_StatCmd_GetSERInfo**.

 Get information on or fix a soft memory error (ECC or parity) is done in the following manner:

| | |
|---------|---------------------------------|
| Get: | EZapiChannel_StatCmd_GetSERInfo |
| Set: | EZapiChannel_ConfigCmd_FixSER |
| Struct: | EZapiChannel_SERInfo |

3.1.13 External Memory BIST

The Channel Group API provides routines for running diagnostic tests on the NPS's external memories. The following types of tests are supported:

- Data Bus – Walking zero and walking one pattern on the data bus. This test is performed by the host SW.
- Address Bus – Checks stuck low, stuck high and shorted bits of the address bus. This test is performed by the host SW.
- Device – Perform a write, read and compare test on a user-defined portion of the memory with a user-defined pattern. In addition, this test can be run in one of three repetition modes: run once; run until the first error is detected (or until manually stopped); or run endlessly (or until manually stopped). This test is performed by the on-chip HW BIST machine (device).

 Configuration and running of external memory BIST tests are done in the following manner:

| | |
|---------|--|
| Set: | EZapiChannel_ConfigCmd_SetExtMemBISTPatternParams |
| Struct: | EZapiChannel_ExtMemBISTPatternParams |
| Set: | EZapiChannel_ConfigCmd_StartExtMemBIST |
| Struct: | EZapiChannel_ExtMemBISTParams |
| Set: | EZapiChannel_ConfigCmd_StartExtMemDataBusSWTest |
| Struct: | EZapiChannel_ExtMemDataBusSWTestParams |
| Set: | EZapiChannel_ConfigCmd_StartExtMemAddressBusSWTest |
| Struct: | EZapiChannel_ExtMemAddressBusSWTestParams |

 The result of the external memory BIST test is retrieved in the following manner:

| | |
|---------|--|
| Get: | EZapiChannel_StatCmd_GetExtMemBISTResult |
| Struct: | EZapiChannel_ExtMemBISTResult |


3.1.14 Interrupt Handling

The Channel Group includes commands to configure and operate with interrupts:


- Destination – PCIe0 MSI
- Interrupts service routine (ISR) – a callback is provided; once an interrupt is triggered the registered callback is called.

 Configuration of interrupts parameters is done in the following manner:

| | |
|---------|--|
| Get: | EZapiChannel_StatCmd_GetInterruptsParams |
| Set: | EZapiChannel_ConfigCmd_SetInterruptsParams |
| Struct: | EZapiChannel_InterruptsParams |

 Control of interrupts is done in the following manner:

| | |
|---------|--|
| Set: | EZapiChannel_ConfigCmd_SetInterruptsControl |
| Struct: | EZapiChannel_InterruptsControl |
| Set: | EZapiChannel_ConfigCmd_Set InterruptsAcknowledge |
| Struct: | EZapiChannel_InterruptsAcknowledge |

 Retrieving interrupt status is done in the following manner:

| | |
|---------|--|
| Get: | EZapiChannel_StatCmd_GetInterruptsStatus |
| Struct: | EZapiChannel_InterruptsStatus |

3.2 Summary of Commands – Channel Group

Table 3-1. Channel Group routines

| API Routines | |
|--------------------------|----------------------------|
| EZapiChannel_Create() | EZapiChannel_Initialize() |
| EZapiChannel_Config() | EZapiChannel_PowerUp() |
| EZapiChannel_Delete() | EZapiChannel_Reset() |
| EZapiChannel_Finalize() | EZapiChannel_Status() |
| EZapiChannel_Go() | EZapiChannel_Stop() |

Table 3-2. Summary of Channel Group commands

This table provides a summary of the commands in the Channel Group, as well as the channel state in which each command may be performed.

| | Undefined | Created | Powered up | Initialized | Finalized | Running |
|--|-----------|----------|------------|-------------|-----------|----------|
| <i>TOPIC & COMMAND</i> | <i>U</i> | <i>C</i> | <i>P</i> | <i>I</i> | <i>F</i> | <i>R</i> |
| General channel parameter configuration | U | C | P | I | F | R |
| Set: EZapiChannel_ConfigCmd_SetGeneralParams | - | ✓ | - | - | - | - |
| Get: EZapiChannel_StatCmd_GetGeneralParams | - | ✓ | ✓ | ✓ | ✓ | ✓ |
| External memory configuration | U | C | P | I | F | R |
| Set: EZapiChannel_ConfigCmd_SetExtMemParams | - | ✓ | - | - | - | - |
| Set: EZapiChannel_ConfigCmd_SetExtMemSpaceParams | - | ✓ | ✓ | - | - | - |
| Get: EZapiChannel_StatCmd_GetExtMemParams | - | ✓ | ✓ | ✓ | ✓ | ✓ |
| Get: EZapiChannel_StatCmd_GetExtMemSpaceParams | - | ✓ | ✓ | ✓ | ✓ | ✓ |
| Internal memory configuration | U | C | P | I | F | R |
| Set: EZapiChannel_ConfigCmd_SetIntMemSpaceParams | - | ✓ | ✓ | - | - | - |
| Get: EZapiChannel_StatCmd_GetIntMemSpaceParams | - | ✓ | ✓ | ✓ | ✓ | ✓ |
| Application commands | U | C | P | I | F | R |
| Set: EZapiChannel_ConfigCmd_SetApplicationParams | - | ✓ | - | - | - | - |
| Get: EZapiChannel_StatCmd_GetApplicationParams | - | ✓ | ✓ | ✓ | ✓ | ✓ |
| Initial network interface configuration | U | C | P | I | F | R |
| Set: EZapiChannel_ConfigCmd_SetEthIFParams | - | ✓ | - | - | - | - |
| Set: EZapiChannel_ConfigCmd_SetEthRXChannelParam | - | ✓ | - | - | - | - |
| Set: EZapiChannel_ConfigCmd_SetEthTXChannelParam | - | ✓ | - | - | - | - |
| Set: EZapiChannel_ConfigCmd_SetInterlakenIFParams | - | ✓ | - | - | - | - |
| Set: EZapiChannel_ConfigCmd_SetInterlakenRXChannelParams | - | ✓ | - | - | - | - |
| Set: EZapiChannel_ConfigCmd_SetInterlakenTXChannelParams | - | ✓ | - | - | - | - |
| Set: EZapiChannel_ConfigCmd_SetLoopbackIFParams | - | ✓ | - | - | - | - |
| Set: EZapiChannel_ConfigCmd_SetTSConfirmationIFParams | - | ✓ | - | - | - | - |
| Set: EZapiChannel_ConfigCmd_SetGCIIFParams | - | ✓ | - | - | - | - |
| Set: EZapiChannel_ConfigCmd_SetInterlakenLAIFParams | - | ✓ | - | - | - | - |
| Set: EZapiChannel_ConfigCmd_SetPCleIFParams | - | ✓ | - | - | - | - |
| Set: EZapiChannel_ConfigCmd_SetMngNetIFParams | - | ✓ | - | - | - | - |
| Set: EZapiChannel_ConfigCmd_SetRXAllocationProfile | - | ✓ | - | - | - | - |

| TOPIC & COMMAND | | U | C | P | I | F | R |
|---------------------------------------|--|----------|----------|----------|----------|----------|----------|
| Get: | EZapiChannel_StatCmd_GetEthIFParams | - | ✓ | ✓ | ✓ | ✓ | ✓ |
| Get: | EZapiChannel_StatCmd_GetEthRXChannelParams | - | ✓ | ✓ | ✓ | ✓ | ✓ |
| Get: | EZapiChannel_StatCmd_GetEthTXChannelParams | - | ✓ | ✓ | ✓ | ✓ | ✓ |
| Get: | EZapiChannel_StatCmd_GetInterlakenIFParams | - | ✓ | ✓ | ✓ | ✓ | ✓ |
| Get: | EZapiChannel_StatCmd_GetInterlakenRXChannelParams | - | ✓ | ✓ | ✓ | ✓ | ✓ |
| Get: | EZapiChannel_StatCmd_GetInterlakenTXChannelParams | - | ✓ | ✓ | ✓ | ✓ | ✓ |
| Get: | EZapiChannel_StatCmd_GetLoopbackIFParams | - | ✓ | ✓ | ✓ | ✓ | ✓ |
| Get: | EZapiChannel_StatCmd_GetTSConfirmationIFParams | - | ✓ | ✓ | ✓ | ✓ | ✓ |
| Get: | EZapiChannel_StatCmd_GetGCIIIFParams | - | ✓ | ✓ | ✓ | ✓ | ✓ |
| Get: | EZapiChannel_StatCmd_GetInterlakenLAIFParams | - | ✓ | ✓ | ✓ | ✓ | ✓ |
| Get: | EZapiChannel_StatCmd_GetPCIEIFParams | - | ✓ | ✓ | ✓ | ✓ | ✓ |
| Get: | EZapiChannel_StatCmd_GetMngNetIFParams | - | ✓ | ✓ | ✓ | ✓ | ✓ |
| Get: | EZapiChannel_StatCmd_GetRXAllocationProfile | - | ✓ | ✓ | ✓ | ✓ | ✓ |
| PCIe configuration | | U | C | P | I | F | R |
| Set: | EZapiChannel_ConfigCmd_SetPCIEMemSpaceParams | - | ✓ | - | - | - | - |
| Set: | EZapiChannel_ConfigCmd_SetPCIEPhysicalFuncParams | - | ✓ | - | - | - | - |
| Set: | EZapiChannel_ConfigCmd_SetPCIEMsgQueueParams | - | ✓ | - | - | - | - |
| Set: | EZapiChannel_ConfigCmd_SetPCIEMsgQueueEventParams | - | ✓ | - | - | - | - |
| Get: | EZapiChannel_StatCmd_GetPCIEMemSpaceParams | - | ✓ | ✓ | ✓ | ✓ | ✓ |
| Get: | EZapiChannel_StatCmd_GetPCIEPhysicalFuncParams | - | ✓ | ✓ | ✓ | ✓ | ✓ |
| Get: | EZapiChannel_StatCmd_GetPCIEMsgQueueParams | - | ✓ | ✓ | ✓ | ✓ | ✓ |
| Get: | EZapiChannel_StatCmd_GetPCIEMsgQueueEventParams | - | ✓ | ✓ | ✓ | ✓ | ✓ |
| PMU configuration | | U | C | P | I | F | R |
| Set: | EZapiChannel_ConfigCmd_SetPMUTimerParams | - | ✓ | - | - | - | - |
| Set: | EZapiChannel_ConfigCmd_SetPMUTimerControl | - | - | - | ✓ | ✓ | ✓ |
| Set: | EZapiChannel_ConfigCmd_SetPMUJobLimitProfileParams | - | ✓ | - | - | - | - |
| Set: | EZapiChannel_ConfigCmd_SetPMUQueueParams | - | ✓ | - | - | - | - |
| Set: | EZapiChannel_ConfigCmd_SetPMUSchedulerParams | - | ✓ | - | - | - | - |
| Get: | EZapiChannel_StatCmd_GetPMUTimerParams | - | ✓ | ✓ | ✓ | ✓ | ✓ |
| Get: | EZapiChannel_StatCmd_GetPMUTimerControl | - | - | - | ✓ | ✓ | ✓ |
| Get: | EZapiChannel_StatCmd_GetPMUStatus | - | - | - | ✓ | ✓ | ✓ |
| Get: | EZapiChannel_StatCmd_GetPMUJobLimitProfileParams | - | ✓ | ✓ | ✓ | ✓ | ✓ |
| Get: | EZapiChannel_StatCmd_GetPMUQueueParams | - | ✓ | ✓ | ✓ | ✓ | ✓ |
| Get: | EZapiChannel_StatCmd_GetPMUQueueStatus | - | - | - | ✓ | ✓ | ✓ |
| Get: | EZapiChannel_StatCmd_GetPMUSchedulerParams | - | ✓ | ✓ | ✓ | ✓ | ✓ |
| Protocol Decoder configuration | | U | C | P | I | F | R |
| Set: | EZapiChannel_ConfigCmd_SetProtocolDecoderParams | - | ✓ | - | - | - | - |
| Get: | EZapiChannel_StatCmd_GetProtocolDecoderParams | - | ✓ | ✓ | ✓ | ✓ | ✓ |
| Index Pool configuration | | U | C | P | I | F | R |
| Set: | EZapiChannel_ConfigCmd_SetIndexPoolParams | - | ✓ | - | - | - | - |
| Get: | EZapiChannel_StatCmd_GetIndexPoolParams | - | ✓ | ✓ | ✓ | ✓ | ✓ |
| Accurate RTC commands | | U | C | P | I | F | R |
| Set: | EZapiChannel_ConfigCmd_SetRTCParams | - | - | - | ✓ | ✓ | ✓ |
| Get: | EZapiChannel_StatCmd_GetRTCParams | - | - | - | ✓ | ✓ | ✓ |
| Set: | EZapiChannel_ConfigCmd_ExecuteRTCOperation | - | - | - | ✓ | ✓ | ✓ |

| TOPIC & COMMAND | U | C | P | I | F | R |
|---|----------|----------|----------|----------|----------|----------|
| SER commands | U | C | P | I | F | R |
| Set: EZapiChannel_ConfigCmd_FixSER | - | - | ✓ | ✓ | ✓ | ✓ |
| Get: EZapiChannel_StatCmd_GetSERInfo | - | - | ✓ | ✓ | ✓ | ✓ |
| External memory BIST commands | U | C | P | I | F | R |
| Set: EZapiChannel_ConfigCmd_SetExtMemBISTPatternParams | - | - | ✓ | ✓ | - | - |
| Set: EZapiChannel_ConfigCmd_StartExtMemBIST | - | - | ✓ | ✓ | - | - |
| Set: EZapiChannel_ConfigCmd_StartExtMemDataBusSWTest | - | - | ✓ | ✓ | ✓ | ✓ |
| Set: EZapiChannel_ConfigCmd_StartExtMemAddressBusSWTest | - | - | ✓ | ✓ | ✓ | ✓ |
| Get: EZapiChannel_StatCmd_GetExtMemBISTResult | - | - | ✓ | ✓ | ✓ | ✓ |
| Interrupt commands | U | C | P | I | F | R |
| Set: EZapiChannel_ConfigCmd_SetInterruptsParams | - | ✓ | - | - | - | - |
| Set: EZapiChannel_ConfigCmd_SetInterruptsControl | - | - | - | ✓ | ✓ | ✓ |
| Set: EZapiChannel_ConfigCmd_SetInterruptsAcknowledge | - | - | - | ✓ | ✓ | ✓ |
| Get: EZapiChannel_StatCmd_GetInterruptsParams | - | ✓ | ✓ | ✓ | ✓ | ✓ |
| Get: EZapiChannel_StatCmd_GetInterruptsStatus | - | - | - | ✓ | ✓ | ✓ |
| Miscellaneous Commands | U | C | P | I | F | R |
| Get: EZapiChannel_StatCmd_GetStateInfo | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Get: EZapiChannel_StatCmd_GetPCInfo | - | ✓ | ✓ | ✓ | ✓ | ✓ |
| Get: EZapiChannel_StatCmd_GetDeviceInfo | - | ✓ | ✓ | ✓ | ✓ | ✓ |
| Set: EZapiChannel_ConfigCmd_SetSystemInfoParams | - | ✓ | - | - | - | - |
| Get: EZapiChannel_StatCmd_GetSystemInfoParams | - | ✓ | ✓ | ✓ | ✓ | ✓ |

4. Network Interface Group

The Network Interface Group contains routines for configuration and management of the NPS device's external network interfaces in use.

4.1 Overview

The Network Interface Group contains routines for configuration, management and getting status for the NPS-400 external networking interfaces.

The Network Interface Group handles several types of interfaces:

- 2x 48x 10GbE XFI/SFI/10GBASE-KR
- 2x 12x 40GbE XLAUI/40GBASE-KR4
- 2x 4x 100GbE CAUI
- 2x 48-lane Interlaken
- 2x 24-lane Interlaken-LA for external TCAM
- 2x SMI (serial management interface)
- 1x LED interface
- 2x OOB (out-of-band) interfaces

 Refer to the *NPS-400 Hardware Reference Manual* for a complete definition of supported combinations of external networking interfaces.

Default values are configured for all interfaces based on system configuration parameters. Thus, there is no need to explicitly configure these items, unless defaults do not match required behavior. When changing configurations, it is recommended to first perform a get operation to retrieve the previous values, modify only the fields whose value needs to be modified to attain the required behavior, and then perform a set operation with the updated software structure.

4.1.1 Network Interface States

The Network Interface Group is an extension of the Channel Group, and thus does not have states of its own, but rather shares the Channel Group states (see "[Channel States](#)" section).


Ethernet and Interlaken MAC and FIFO parameters must be defined when the channel is in the created state. When the channel is powered-up (by calling **EZapiChannel_PowerUp()**), these configuration are transferred to the NPS-400 device, and these interfaces are partially initialized.

SMI and LED interface parameters must be defined when the channel is in the created state. When the channel is powered-up (by calling **EZapiChannel_PowerUp()**), these configuration are transferred to the NPS-400 device, and these interfaces are initialized.

The OOB interface general parameters must be defined when the channel is in the created state. When the channel is powered-up (by calling **EZapiChannel_PowerUp()**), these configuration are transferred to the NPS-400 device, and the OOB interface general configurations are programmed.

When the channel is in the initialized state, the Network Interface Group routines may be called to change network interface configurations. The enabling of the main network interfaces (10GE/40GE/100GE/Interlaken) is cached in the CP library and performed only during the **EZapiChannel_Go()** routine.

Several of the Network Interface Group routines may also be called to change network interface configurations at runtime. Most interface configuration parameters should not be changed while frames are flowing through the interface.

 For a summary of which Network Interface Group commands can be called during which states, see [“Summary of Commands – IF Group”](#).

4.1.2 Configuring Ethernet Interfaces


The Network Interface Group contains several commands to configure and manage the various Ethernet interfaces. This section describes all of the Ethernet interface types – 10GE, 40GE and 100GE

External interfaces are grouped into Ethernet interface engines with each interface engine serving 12 SerDes lanes. The Ethernet interface engine configuration is performed by calling the API command **EZapiIF_ConfigCmd_SetEthIFEPParams()**. In addition to the basic interface configuration, the NPS-400's Ethernet interfaces support embedded statistic counters for counting interface events.

The initial definition of all the various Ethernet interfaces/ports, is performed from the API command **EZapiIF_ConfigCmd_SetEthMACParams()** based on its Interface Type and Interface Number parameters. Some parameters may apply only to specific interface types.

When changing interface configuration by calling **EZapiIF_ConfigCmd_SetEthMACParams()** in run time, you should make sure that the interface is disabled for receiving/transmitting traffic.


Ethernet interface engines parameters configure modes/parameters which affect all interfaces mapped to a specific engine, such as IEEE1588 timestamp processing and timeout values.

 The Ethernet interface engine parameter configuration is done in the following manner:

```
Set:      EZapiIF_ConfigCmd_SetEthIFEPParams
Get:      EZapiIF_StatCmd_GetEthIFEPParams
Struct:   EZapiIF_EthIFEPParams
```

The interface configuration parameters include:

- Interface Type – Determines the type of interface to be configured – 10GE, 40GE and 100GE.
- Interface Sub-Type – Determines the sub-type of interface to be configured; 10GE_XFI, 10GE_SFI, 10GE_KR, 40GE_XLAUI, 40GE_XLPPI or 40GE_KR. For a 100GE interface there is only one option, 100GE_CAUI, so this field is irrelevant for 100GE interfaces.
- Interface Number – Determines the number of interface to be configured.
- The interface's MAC address.
- Enabling/disabling of receiving/transmitting of frames.
- CRC support – Configurations affecting CRC checking, stripping and adding.
- Frame length support – Checking of maximum length for incoming frames, checking of frame length against the frame length field in the Ethernet header and padding of outgoing frames.
- TX MAC flushing.
- Unidirectional mode configuration affecting behavior on fault reception.
- IEEE1588 timestamp processing.
- Flow control support – Configurations affecting link-level flow control, class-based flow control, priority-based flow control, and out-of-band flow control.
- Inter-frame gap (IFG) support – Minimum IFG between incoming/outgoing frames, IFG rate control and thresholds for out-of-band IFG status indications.
- Rx and Tx Software preamble – Configure the interface to work in Software Preamble mode.
- Inter-frame gap rate control – MAC operates in self pace mode (minimum IFG length depends on the previous frame length). Valid for 10GE interfaces.

 Ethernet MAC configuration is done in the following manner:

```
Set:      EZapiIF_ConfigCmd_SetEthMACParams
Get:      EZapiIF_StatCmd_GetEthMACParams
Struct:   EZapiIF_EthMACParams

Set:      EZapiIF_ConfigCmd_SetEthMACControl
Get:      EZapiIF_StatCmd_GetEthMACControl
Struct:   EZapiIF_EthMACControl

Get:      EZapiIF_StatCmd_GetEthMACStatus
Struct:   EZapiIF_EZapiIF_EthMACStatus
```

 The interrupt configuration/status for Ethernet ports is done in the following manner:


```
Set:      EZapiIF_ConfigCmd_SetEthInterruptControl
Struct:   EZapiIF_EthInterruptControl

Get:      EZapiIF_StatCmd_GetEthInterruptStatus
Struct:   EZapiIF_EthInterruptStatus
```


► *Fields may be affected by “reduced mode” which is configured in EZapiChannel_GeneralParams.*

4.1.2.1 Ethernet Statistic Counters


The NPS-400 Ethernet interfaces include integrated statistic counters for tracking interface events. The Network Interface Group supplies commands for reading (and optionally resetting) the value of one or more counters (including the counter’s overflow status), as well as for resetting the counters.

 An interface’s statistic counters are read (and optionally reset) in the following manner:

```
Get:      EZapiIF_StatCmd_GetEthStatCounter
Struct:   EZapiIF_EthStatCounter
```

 An interface’s statistic counters are reset in the following manner:

```
Set:      EZapiIF_ConfigCmd_ResetEthStatCounter
Struct:   EZapiIF_EthStatCounter
```

 Configure interface statistics maximum frame length profile parameters for given profile ID in given IFE in the following manner:

```
Set:      EZapiIF_ConfigCmd_SetStatOversizedFrameLengthProfile
Get:      EZapiIF_StatCmd_GetStatOversizedFrameLengthProfile
Struct:   EZapiIF_StatOversizedFrameLengthProfile
```

► *The interface’s statistic counters are automatically reset during channel creation.*

4.1.3 Configuring Interlaken Interfaces

The NPS-400 has two types of Interlaken interfaces; one type is used for normal network traffic, while Interlaken-LA connects to an external TCAM knowledge based processor.

The Network Interface Group contains several commands to configure and manage the Interlaken interfaces. In addition to the basic interface configuration, the NPS-400's Interlaken interfaces support embedded statistic counters for counting interface events.

4.1.3.1 Interlaken Interface MAC Configuration

The Interlaken interface configuration supports configuring the MAC for each of the Interlaken interfaces, such as:

- Enabling/disabling of receiving/transmitting of frames.
- Calendar length
- Flow control support – Configurations affecting link-level flow control, class-based flow control, per-channel flow control, per-context flow control and out-of-band flow control.
- Rate limiting support – Configurations for rate limiting token bucket (RateLimit and BurstLimit).
- IEEE1588 timestamp processing.
- CRC support – Configurations affecting CRC checking, stripping and adding.
- Retransmission support
- Interlaken statistics accumulation mode



The Interlaken interface configuration is done in the following manner:

```
Set:      EZapiIF_ConfigCmd_SetInterlakenParams
Get:      EZapiIF_StatCmd_GetInterlakenParams
Struct:   EZapiIF_InterlakenParams

Set:      EZapiIF_ConfigCmd_SetInterlakenControl
Get:      EZapiIF_StatCmd_GetInterlakenControl
Struct:   EZapiIF_InterlakenControl

Get:      EZapiIF_StatCmd_GetInterlakenStatus
Struct:   EZapiIF_InterlakenStatus
```



The interrupt configuration/status for Interlaken and Interlaken-LA interfaces is done in the following manner:

```
Set:      EZapiIF_ConfigCmd_SetInterlakenInterruptControl
Struct:   EZapiIF_InterlakenInterruptControl

Get:      EZapiIF_StatCmd_GetInterlakenInterruptStatus
Struct:   EZapiIF_InterlakenInterruptStatus
```



The Interlaken-LA interface configuration is done in the following manner:


```
Set:      EZapiIF_ConfigCmd_SetInterlakenLAParams
Get:      EZapiIF_StatCmd_GetInterlakenLAParams
Struct:   EZapiIF_InterlakenLAParams

Get:      EZapiIF_StatCmd_GetInterlakenLAStatus
Struct:   EZapiIF_InterlakenLAStatus
```

4.1.3.2 Interlaken Statistic Counters


The Interlaken interfaces include integrated statistic counters for tracking interface events. The Network Interface Group supplies commands for reading and resetting the value of one or more counters.

The counting mode and counter ID are configured in the Interlaken Params structure.

 The Interlaken/Interlaken-LA interfaces statistic counters are read (and optionally reset) in the following manner:


```
Get:      EZapiIF_StatCmd_GetInterlakenStatCounter
Struct:   EZapiIF_InterlakenStatCounter

Get:      EZapiIF_StatCmd_GetInterlakenLStatCounter
Struct:   EZapiIF_InterlakenLStatCounter
```

 The Interlaken/Interlaken-LA interfaces statistic counters are reset in the following manner:

```
Set:      EZapiIF_ConfigCmd_ResetInterlakenStatCounter
Struct:   EZapiIF_InterlakenStatCounter

Set:      EZapiIF_ConfigCmd_ResetInterlakenLStatCounter
Struct:   EZapiIF_InterlakenLStatCounter
```

 Configure interface statistics maximum frame length profile parameters for given profile ID in given IFE in the following manner:

```
Set:      EZapiIF_ConfigCmd_SetStatOversizedFrameLengthProfile
Get:      EZapiIF_StatCmd_GetStatOversizedFrameLengthProfile
Struct:   EZapiIF_StatOversizedFrameLengthProfile
```

► *The Interlaken interface's statistic counters are automatically reset during channel creation.*

4.1.4 Configuring RX/TX FIFOs

Traffic received by the NPS-400 flows from the RX MACs to the RX FIFOs. Traffic transmitted out from the NPS-400 flows from the TX FIFOs to the TX MACs. The Network Interface Group supplies commands for configuring parameters related to the size of the FIFOs allocated for the interfaces and setting of thresholds that generate flow control, priority drop and other indications. The RX/TX FIFOs are allocated in dedicated SRAMs.

The space dedicated to the FIFOs is configured through FIFO slices. There are 12 slices allocated for these FIFOs. Each slice configures the RX FIFO slice size and the TX FIFO slice size.

Each interface is mapped to a specific slice:

- 10GE interface occupies one-quarter of a slice.
- 40GE interface occupies 1 slice.
- 100GE interface occupies 2 slices.
- An Interlaken interface occupies a maximum of 8 slices based on a slice per 6 lanes.

The total FIFO size of an interface cannot exceed the size of the slice mapped to the interface.



The FIFO slice configuration is done in the following manner:

```
Set:      EZapiIF_ConfigCmd_SetFIFOSliceParams
Get:      EZapiIF_StatCmd_GetFIFOSliceParams
Struct:   EZapiIF_FIFOSliceParams
```



The RX FIFOs parameter configuration is done in the following manner:

```
Set:      EZapiIF_ConfigCmd_SetEthRXFIFOParams
Get:      EZapiIF_StatCmd_GetEthRXFIFOParams
Struct:   EZapiIF_EthRXFIFOParams

Set:      EZapiIF_ConfigCmd_SetInterlakenRXFIFOParams
Get:      EZapiIF_StatCmd_GetInterlakenRXFIFOParams
Struct:   EZapiIF_InterlakenRXFIFOParams
```



The TX FIFOs parameter configuration is done in the following manner:

```
Set:      EZapiIF_ConfigCmd_SetEthTXFIFOParams
Get:      EZapiIF_StatCmd_GetEthTXFIFOParams
Struct:   EZapiIF_EthTXFIFOParams

Set:      EZapiIF_ConfigCmd_SetInterlakenTXFIFOParams
Get:      EZapiIF_StatCmd_GetInterlakenTXFIFOParams
Struct:   EZapiIF_InterlakenTXFIFOParams
```

4.1.5 Configuring General Parameters

General parameters configure additional modes/parameters which affect all interfaces.



The general parameter configuration is done in the following manner:

```
Set:      EZapiIF_ConfigCmd_SetGeneralParams
Get:      EZapiIF_StatCmd_GetGeneralParams
Struct:   EZapiIF_GeneralParams
```


4.1.6 Configuring the Priority Flow Control Parameters

Priority based flow control parameters define how the network interfaces operate with priority based flow control (PFC) according to the number of TX channels on the interface.

The following types of priority-based flow control parameters exist:

- RX – Used to translate received priority-based FC status (8 bits) to the NPS-400's four COS levels (4 bits). Each profile defines a separate bitmap of four NPS-400 COS levels affected by each of the eight class-based FC status bits, allowing full flexibility in the mapping. When a priority-based FC status is received, an NPS-400 COS level is affected if any one of the flow-controlled classes is configured to affect it.



The priority flow control parameter configurations are done in the following manner:

```
Set:      EZapiIF_ConfigCmd_SetEthRXPriorityBasedFCParams
Get:      EZapiIF_StatCmd_GetEthRXPriorityBasedFCParams
Struct:   EZapiIF_EthRXPriorityBasedFCParams
```

4.1.7 Configuring the Out of Band Flow Control Interfaces

The Network Interface Group contains several commands to configure and manage the out of band flow control interfaces (two incoming interfaces and one outgoing interface). The OOB FC interfaces can use the SPI bus standard interface or the Interlaken interface to import/export flow control indications. The transmitted signals can operate at core clock multiples of 1/2, 1/4 and 1/8.

Each out of band interface is controlled by a configurable calendar. Each calendar command is interpreted as referring to an associated client and can represent 1, 4 or 16 calendar slots. The calendar configuration memory contains 512 lines, each containing a configuration of six commands. The configuration memory is partitioned to control slots for each out of band interface. The size and content of each control slot should match the calendar length configured for the respective interface.



The OOB FC interface configuration is done in the following manner:

```
Set:      EZapiIF_ConfigCmd_SetOOBFCIFParams
Get:      EZapiIF_StatCmd_GetOOBFCIFParams
Struct:   EZapiIF_OOBFCIFParams
```



The calendar sequence configurations are done in the following manner:

```
Set:      EZapiIF_ConfigCmd_SetOOBFCInCalEntryParams
Get:      EZapiIF_StatCmd_GetOOBFCInCalEntryParams
Struct:   EZapiIF_OOBFCInCalEntryParams


Set:      EZapiIF_ConfigCmd_SetOOBFCOutCalEntryParams
Get:      EZapiIF_StatCmd_GetOOBFCOutCalEntryParams
Struct:   EZapiIF_OOBFCOutCalEntryParams
```



The OOB FC interface is explicitly enabled or disabled the following manner:

```
Set:      EZapiIF_ConfigCmd_SetOOBFCIFEnableParams
Get:      EZapiIF_StatCmd_GetOOBFCIFEnableParams
Struct:   EZapiIF_OOBFCIFEnableParams
```


4.1.8 Configuring the LED Interface

 The LED interface configuration is done in the following manner:


Set: EZapiIF_ConfigCmd_SetLEDParams
 Get: EZapiIF_StatCmd_GetLEDParams
 Struct: EZapiIF_LEDParams

4.1.9 Configuring SerDes Parameters

The Network Interface Group provides commands to configure SerDes parameters.

 The SerDes low-level access is done in the following manner:


Set: EZapiIF_ConfigCmd_SetSerDesTXParams
 Set: EZapiIF_ConfigCmd_SetSerDesRXParams
 Set: EZapiIF_ConfigCmd_SetSerDesPRBSTestControl
 Set: EZapiIF_ConfigCmd_SetSerDesEyePattern
 Get: EZapiIF_StatCmd_GetSerDesTXParams
 Get: EZapiIF_StatCmd_GetSerDesRXParams
 Get: EZapiIF_StatCmd_GetSerDesPRBSTestStatus
 Struct: EZapiIF_SerDesTXParams
 Struct: EZapiIF_SerDesRXParams
 Struct: EZapiIF_SerDesPRBSTestControl
 Struct: EZapiIF_SerDesEyePattern

 Retrieval of information on the internal SerDes is done in the following manner:


Get: EZapiIF_StatCmd_GetSerDesStatus
 Struct: EZapiIF_SerDesStatus

4.1.10 Configuring SMI Interfaces

The Network Interface Group provides commands for reading and writing of PHY registers external to the NPS via two serial management interfaces. Note that the CP library does not initialize or maintain external PHY devices.

 Configuration of the SMI interfaces is done in the following manner:

Set: EZapiIF_ConfigCmd_SetSMIParams
 Get: EZapiIF_StatCmd_GetSMIParams
 Struct: EZapiIF_SMIParams

 Writing/reading an external PHY register via the SMI interfaces is done in the following manner:

Set: EZapiIF_ConfigCmd_WritePhyReg
 Get: EZapiIF_StatCmd_ReadPhyReg
 Struct: EZapiIF_PHYRegister

 The interrupt configuration/status for SMI interfaces is done in the following manner:

Set: EZapiIF_ConfigCmd_SetSMIIInterruptControl
 Struct: EZapiIF_SMIIInterruptControl
 Get: EZapiIF_StatCmd_GetSMIIInterruptStatus
 Struct: EZapiIF_SMIIInterruptStatus

4.1.11 Auto-negotiation

The Network Interface Group includes several commands relating to auto-negotiation with a peer device. These commands are relevant only for 10GE_KR and 40GE_KR interfaces.

Before starting the auto-negotiation process, the developer should define the advertisement parameters of the interface (the SerDes that controls the auto-negotiation and the flow control abilities) by calling EZapiIF_ConfigCmd_SetANParams.

After the advertisement parameters are defined, the developer may start the auto-negotiation process.

By calling EZapiIF_ConfigCmd_SetANControl, the developer can go through all of the auto-negotiation phases. Advance to the next phase only after the previous phase is finished.


The phases are: EZapiIF_ANOperation_START_AN_PROCESS,
EZapiIF_ANOperation_SEND_NEXT_PAGE,
EZapiIF_ANOperation_START_TX_TRAINING,
EZapiIF_ANOperation_COMPLETE_TX_TRAINING,
EZapiIF_ANOperation_COMPLETE_AN_PROCESS,
EZapiIF_ANOperation_STOP_AN_PROCESS.

Auto-negotiation commands include:

- Set AN Parameters – Configures the auto-negotiation parameters, such as the mode of operations and the capabilities to advertise.
- Get AN Parameters – Returns the currently configured auto-negotiation parameters.
- Get AN Status – This command checks that the auto-negotiation process has completed, reads the negotiated capabilities, and configures the NPS's PCS and MAC according to the negotiated results. The command also returns status information on the negotiated values.
- Send Next Page – Sends the next page (used for passing the AN process).
- Start TX Training – Starts the TX training.

When using the out-of-band mode, the above APIs are not used. Instead, the following APIs are used to operate directly with the external PHY:

- WritePHYReg – Write a single PHY register via the SMI interface.
- ReadPHYReg – Read a single PHY register via the SMI interface.

 The auto-negotiation parameter configuration is done in the following manner::

| | |
|---------|--|
| Set: | EZapiIF_ConfigCmd_SetANParams (also starts the auto-negotiation process) |
| Get: | EZapiIF_StatCmd_GetANParams |
| Struct: | EZapiIF_ANParams |
| Set: | EZapiIF_ConfigCmd_SetANControl |
| Struct: | EZapiIF_ANControl |
| Get: | EZapiIF_StatCmd_GetANStatus |
| Struct: | EZapiIF_ANStatus |

4.2 Summary of Commands – IF Group

Table 4-1. Network Interface Group routines

| API Routines |
|------------------|
| EZapiIF_Config() |
| EZapiIF_Status() |

Table 4-2. Summary of Network Interface Group commands

This table provides a summary of the commands in the Network Interface Group, as well as the channel state in which each command may be performed.

| | Undefined | Created | Powered up | Initialized | Finalized | Running |
|---|-----------|----------|------------|-------------|-----------|----------|
| TOPIC & COMMAND | U | C | P | I | F | R |
| Interface configuration | | | | | | |
| Ethernet port configuration | | | | | | |
| Set: EZapiIF_ConfigCmd_SetEthIFEPParams | - | ✓ | - | - | - | - |
| Set: EZapiIF_ConfigCmd_SetEthMACParams | - | ✓ | ✓ | ✓ | ✓ | ✓ |
| Set: EZapiIF_ConfigCmd_SetEthMACControl (*When NPS is in running state this can be called only when EnableReceiveand EnableTransmit in EthMACControl are FALSE.) | - | - | - | ✓ | ✓ | ✓ * |
| Set: EZapiIF_ConfigCmd_SetEthInterruptControl | - | - | - | ✓ | ✓ | ✓ |
| Get: EZapiIF_StatCmd_GetEthIFEPParams | - | ✓ | ✓ | ✓ | ✓ | ✓ |
| Get: EZapiIF_StatCmd_GetEthMACParams | - | ✓ | ✓ | ✓ | ✓ | ✓ |
| Get: EZapiIF_StatCmd_GetEthMACControl | - | - | - | ✓ | ✓ | ✓ |
| Get: EZapiIF_StatCmd_GetEthInterruptStatus | - | - | - | ✓ | ✓ | ✓ |
| Get: EZapiIF_StatCmd_GetEthMACStatus | - | - | - | ✓ | ✓ | ✓ |
| Interlaken interface configuration | | | | | | |
| Set: EZapiIF_ConfigCmd_SetInterlakenParams | - | ✓ | - | - | - | - |
| Set: EZapiIF_ConfigCmd_SetInterlakenControl | - | - | - | ✓ | ✓ | ✓ |
| Set: EZapiIF_ConfigCmd_SetInterlakenLAParams | - | ✓ | - | - | - | - |
| Set: EZapiIF_ConfigCmd_SetInterlakenInterruptControl | - | - | - | ✓ | ✓ | ✓ |
| Get: EZapiIF_StatCmd_GetInterlakenParams | - | ✓ | ✓ | ✓ | ✓ | ✓ |
| Get: EZapiIF_StatCmd_GetInterlakenControl | - | - | - | ✓ | ✓ | ✓ |
| Get: EZapiIF_StatCmd_GetInterlakenLAParams | - | ✓ | ✓ | ✓ | ✓ | ✓ |
| Get: EZapiIF_StatCmd_GetInterlakenInterruptStatus | - | - | - | ✓ | ✓ | ✓ |
| Get: EZapiIF_StatCmd_GetInterlakenStatus | - | - | - | ✓ | ✓ | ✓ |
| Get: EZapiIF_StatCmd_GetInterlakenLAStatus | - | - | - | ✓ | ✓ | ✓ |
| Miscellaneous commands | U | C | P | I | F | R |
| General interface configuration | | | | | | |
| Set: EZapiIF_ConfigCmd_SetGeneralParams | - | ✓ | - | - | - | - |
| Get: EZapiIF_StatCmd_GetGeneralParams | - | ✓ | ✓ | ✓ | ✓ | ✓ |
| LED interface configuration | | | | | | |
| Set: EZapiIF_ConfigCmd_SetLEDParams | - | ✓ | - | - | - | - |
| Get: EZapiIF_StatCmd_GetLEDParams | - | ✓ | ✓ | ✓ | ✓ | ✓ |

| TOPIC & COMMAND | U | C | P | I | F | R |
|--|----------|----------|----------|----------|----------|----------|
| Serial management interface configuration | | | | | | |
| Set: EZapiIF_ConfigCmd_SetSMIPParams | - | ✓ | - | - | - | - |
| Set: EZapiIF_ConfigCmd_SetSMIIInterruptControl | - | - | - | ✓ | ✓ | ✓ |
| Get: EZapiIF_StatCmd_GetSMIPParams | - | ✓ | ✓ | ✓ | ✓ | ✓ |
| Get: EZapiIF_StatCmd_GetSMIIInterruptStatus | - | - | - | ✓ | ✓ | ✓ |
| Access PHY registers using SMI interfaces | | | | | | |
| Set: EZapiIF_ConfigCmd_WritePhyReg | - | ✓ | ✓ | ✓ | ✓ | ✓ |
| Get: EZapiIF_StatCmd_ReadPhyReg | - | ✓ | ✓ | ✓ | ✓ | ✓ |
| SerDes commands | U | C | P | I | F | R |
| Set: EZapiIF_ConfigCmd_SetSerDesTXParams | - | ✓ | - | - | - | - |
| Set: EZapiIF_ConfigCmd_SetSerDesRXParams | - | ✓ | - | - | - | - |
| Set: EZapiIF_ConfigCmd_SetSerDesPRBSTestControl | - | - | ✓ | ✓ | ✓ | ✓ |
| Set: EZapiIF_ConfigCmd_SetSerDesEyePattern | - | - | ✓ | ✓ | ✓ | ✓ |
| Get: EZapiIF_StatCmd_GetSerDesTXParams | - | - | ✓ | ✓ | ✓ | ✓ |
| Get: EZapiIF_StatCmd_GetSerDesRXParams | - | ✓ | ✓ | ✓ | ✓ | ✓ |
| Get: EZapiIF_StatCmd_GetSerDesPRBSTestStatus | - | - | ✓ | ✓ | ✓ | ✓ |
| Get: EZapiIF_StatCmd_GetSerDesStatus | - | - | ✓ | ✓ | ✓ | ✓ |
| Statistic counters read/reset | U | C | P | I | F | R |
| Set: EZapiIF_ConfigCmd_ResetEthStatCounter | - | - | - | ✓ | ✓ | ✓ |
| Set: EZapiIF_ConfigCmd_ResetInterlakenStatCounter | - | - | - | ✓ | ✓ | ✓ |
| Set: EZapiIF_ConfigCmd_ResetInterlakenLAStatCounter | - | - | - | ✓ | ✓ | ✓ |
| Get: EZapiIF_StatCmd_GetEthStatCounter | - | - | - | ✓ | ✓ | ✓ |
| Get: EZapiIF_StatCmd_GetInterlakenStatCounter | - | - | - | ✓ | ✓ | ✓ |
| Get: EZapiIF_StatCmd_GetInterlakenLAStatCounter | - | - | - | ✓ | ✓ | ✓ |
| Interface FIFO parameter configuration | U | C | P | I | F | R |
| Set: EZapiIF_ConfigCmd_SetFIFOSliceParams | - | ✓ | - | - | - | - |
| Set: EZapiIF_ConfigCmd_SetEthRXFIFOParams | - | - | ✓ | - | - | - |
| Set: EZapiIF_ConfigCmd_SetEthTXFIFOParams | - | - | ✓ | - | - | - |
| Set: EZapiIF_ConfigCmd_SetInterlakenRXFIFOParams | - | - | ✓ | - | - | - |
| Set: EZapiIF_ConfigCmd_SetInterlakenTXFIFOParams | - | - | ✓ | - | - | - |
| Get: EZapiIF_StatCmd_GetFIFOSliceParams | - | ✓ | ✓ | ✓ | ✓ | ✓ |
| Get: EZapiIF_StatCmd_GetEthRXFIFOParams | - | - | ✓ | ✓ | ✓ | ✓ |
| Get: EZapiIF_StatCmd_GetEthTXFIFOParams | - | - | ✓ | ✓ | ✓ | ✓ |
| Get: EZapiIF_StatCmd_GetInterlakenRXFIFOParams | - | - | ✓ | ✓ | ✓ | ✓ |
| Get: EZapiIF_StatCmd_GetInterlakenTXFIFOParams | - | - | ✓ | ✓ | ✓ | ✓ |
| Priority flow control configuration | U | C | P | I | F | R |
| Set: EZapiIF_ConfigCmd_SetEthRXPriorityBasedFCParams | - | ✓ | ✓ | - | - | - |
| Get: EZapiIF_StatCmd_GetEthRXPriorityBasedFCParams | - | ✓ | ✓ | - | - | - |
| Out of band flow control configuration | U | C | P | I | F | R |
| Set: EZapiIF_ConfigCmd_SetOOBFCIFParams | - | ✓ | - | - | - | - |
| Set: EZapiIF_ConfigCmd_SetOOBFCIFEnableParams | - | - | - | - | ✓ | ✓ |
| Set: EZapiIF_ConfigCmd_SetOOBFCInCalEntryParams | - | - | - | ✓ | - | - |
| Set: EZapiIF_ConfigCmd_SetOOBFCOutCalEntryParams | - | - | - | ✓ | - | - |
| Get: EZapiIF_StatCmd_GetOOBFCIFParams | - | ✓ | ✓ | ✓ | ✓ | ✓ |
| Get: EZapiIF_StatCmd_GetOOBFCIFEnableParams | - | ✓ | ✓ | ✓ | ✓ | ✓ |

| TOPIC & COMMAND | U | C | P | I | F | R |
|--|----------|----------|----------|----------|----------|----------|
| Get: EZapiIF_StatCmd_GetOOBFCInCalEntryParams | - | ✓ | ✓ | ✓ | ✓ | ✓ |
| Get: EZapiIF_StatCmd_GetOOBFCOutCalEntryParams | - | ✓ | ✓ | ✓ | ✓ | ✓ |
| Interface statistics maximum frame length profile configuration | U | C | P | I | F | R |
| Set: EZapiIF_ConfigCmd_SetStatOversizedFrameLengthProfile | - | - | - | ✓ | ✓ | ✓ |
| Get: EZapiIF_StatCmd_GetStatOversizedFrameLengthProfile | - | - | - | ✓ | ✓ | ✓ |
| Auto negotiation commands | U | C | P | I | F | R |
| Set: EZapiIF_ConfigCmd_SetANParams | - | - | ✓ | ✓ | ✓ | ✓ |
| Set: EZapiIF_ConfigCmd_SetANControl | - | - | ✓ | ✓ | ✓ | ✓ |
| Get: EZapiIF_StatCmd_GetANParams | - | - | ✓ | ✓ | ✓ | ✓ |
| Get: EZapiIF_StatCmd_GetANStatus | - | - | ✓ | ✓ | ✓ | ✓ |

5. Input Classification Unit Group


The Input Classification Unit (ICU) Group contains API routines for configuration and management of NPS device's embedded Input Classification Units.

5.1 Overview

The ICU Group includes commands to configure the Input Classification Units (ICU) and related functionality, such as the hardware decoding performed by the ICU, the classification configurations, and the generation of COS, layer 2 class and hash indications toward the RND unit.


5.1.1 ICU States

The ICU Group is an extension of the Channel Group, and thus does not have states of its own, but rather shares the Channel Group states (see "[Channel States](#)" section).

 For a summary of which ICU commands can be called during which states, see "[Summary of Commands – ICU Group](#)".

5.1.2 ICU Packet Parsing and Classification

The ICU performs the classification of incoming packets based on the following parameters: Ethernet packet's physical source port or Interlaken channel. Parts of the configured options are global and others are per Ethernet source port or Interlaken channel.


 Mapping of the configured options is performed via the following commands:

| | |
|---------|---|
| Set: | EZapiICU_ConfigCmd_SetEthParams |
| Get: | EZapiICU_StatCmd_GetEthParams |
| Struct: | EZapiICU_EthParams |
| Set: | EZapiICU_ConfigCmd_SetInterlakenParams |
| Get: | EZapiICU_StatCmd_GetInterlakenParams |
| Struct: | EZapiICU_InterlakenParams |
| Set: | EZapiICU_ConfigCmd_SetInterlakenProfile |
| Get: | EZapiICU_StatCmd_GetInterlakenProfile |
| Struct: | EZapiICU_InterlakenProfile |

The ICU support two modes of classification: (1) internal classification mode where the ICU internally generates the COS, layer 2 class and hash indications, or (2) external classification mode where the ICU extracts the COS, layer 2 class and hash indications from the header based on the following profile:

| | |
|---------|---|
| Set: | EZapiICU_ConfigCmd_SetExternalClassificationProfile |
| Get: | EZapiICU_StatCmd_GetExternalClassificationProfile |
| Struct: | EZapiICU_ExternalClassificationProfile |

When the ICU classification is internal mode, the ICU generates the COS, layer 2 Class and hash indications. The COS and layer 2 class can be remapped to different values. The hash value is used for load balancing proposes; in this case the hash value defines the PMU queue offset within a range of queues.

 COS remapping and load balancing are performed via the following commands:


| | |
|------|--|
| Set: | EZapiICU_ConfigCmd_SetCOSToOUTPrioMappingProfile |
|------|--|

Get: EZapiICU_StatCmd_GetCOSToOUTPrioMappingProfile
 Struct: EZapiICU_COSToOUTPrioMappingProfile

Set: EZapiICU_ConfigCmd_SetLoadBalanceProfile
 Get: EZapiICU_StatCmd_GetLoadBalanceProfile
 Struct: EZapiICU_LoadBalanceProfile


5.1.3 ICU Format Encapsulation

The ICU classification is performed by parsing the packet encapsulation. Once the encapsulation format is identified, the additional configurations (profiles) are used to generate the ICU outputs.

 The ICU encapsulation options are based on the following profile:

Set: EZapiICU_ConfigCmd_SetEncapsulationFormatProfile
 Get: EZapiICU_StatCmd_GetEncapsulationFormatProfile
 Struct: EZapiICU_EncapsulationFormatProfile

5.1.4 Layer 2 Classification

 The following configurations are used to control the Layer 2 classification:

Set: EZapiICU_ConfigCmd_SetL2ControlProfile
 Get: EZapiICU_StatCmd_GetL2ControlProfile
 Struct: EZapiICU_L2ControlProfile


Set: EZapiICU_ConfigCmd_SetMiscParams
 Get: EZapiICU_StatCmd_GetMiscParams
 Struct: EZapiICU_MiscParams

Set: EZapiICU_ConfigCmd_SetVLANEncapsulationCOSMappingProfile
 Get: EZapiICU_StatCmd_GetVLANEncapsulationCOSMappingProfile
 Struct: EZapiICU_VLANEncapsulationCOSMappingProfile

Set: EZapiICU_ConfigCmd_SetControlFramesCOSParams
 Get: EZapiICU_StatCmd_GetControlFramesCOSParams
 Struct: EZapiICU_ControlFramesCOSParams

Set: EZapiICU_ConfigCmd_SetL2UserDefinedDestAddrAndEthertype
 Get: EZapiICU_StatCmd_GetL2UserDefinedDestAddrAndEthertype
 Struct: EZapiICU_L2UserDefinedDestAddrAndEthertype

5.1.5 Layer 3 MPLS Classification

 The following configurations are used to control the MPLS classification:

Set: EZapiICU_ConfigCmd_SetMPLSProfile
 Get: EZapiICU_StatCmd_GetMPLSProfile
 Struct: EZapiICU_MPLSProfile


Set: EZapiICU_ConfigCmd_SetClassificationTables
 Get: EZapiICU_StatCmd_GetClassificationTables
 Struct: EZapiICU_ClassificationTables

Set: EZapiICU_ConfigCmd_SetMPLSUniEncapsulationCOSMappingProfile

Get: EZapiICU_StatCmd_GetMPLSUniEncapsulationCOSMappingProfile
 Struct: EZapiICU_MPLSUniEncapsulationCOSMappingProfile

 Set: EZapiICU_ConfigCmd_SetMPLSMulEncapsulationCOSMappingProfile
 Get: EZapiICU_StatCmd_GetMPLSMulEncapsulationCOSMappingProfile
 Struct: EZapiICU_MPLSMulEncapsulationCOSMappingProfile

5.1.6 Layer 3 IP Classification

 The following configurations are used to control the IPv4 and IPv6 classification:

Set: EZapiICU_ConfigCmd_SetIPv4Profile
 Get: EZapiICU_StatCmd_GetIPv4Profile
 Struct: EZapiICU_IPv4Profile

 Set: EZapiICU_ConfigCmd_SetIPv6Profile
 Get: EZapiICU_StatCmd_GetIPv6Profile
 Struct: EZapiICU_IPv6Profile

 Set: EZapiICU_ConfigCmd_SetClassificationTables
 Get: EZapiICU_StatCmd_GetClassificationTables
 Struct: EZapiICU_ClassificationTables

 Set: EZapiICU_ConfigCmd_SetL3IPv4ControlProfile
 Get: EZapiICU_StatCmd_GetL3IPv4ControlProfile
 Struct: EZapiICU_L3IPv4ControlProfile


 Set: EZapiICU_ConfigCmd_SetL3IPv6ControlProfile
 Get: EZapiICU_StatCmd_GetL3IPv6ControlProfile
 Struct: EZapiICU_L3IPv6ControlProfile

 Set: EZapiICU_ConfigCmd_SetL3ControlCOSParams
 Get: EZapiICU_StatCmd_GetL3ControlCOSParams
 Struct: EZapiICU_L3ControlCOSParams

 Set: EZapiICU_ConfigCmd_SetIPv4EncapsulationCOSMappingProfile
 Get: EZapiICU_StatCmd_GetIPv4EncapsulationCOSMappingProfile
 Struct: EZapiICU_IPv4EncapsulationCOSMappingProfile

 Set: EZapiICU_ConfigCmd_SetIPv6EncapsulationCOSMappingProfile
 Get: EZapiICU_StatCmd_GetIPv6EncapsulationCOSMappingProfile
 Struct: EZapiICU_IPv6EncapsulationCOSMappingProfile

5.1.7 Layer 4 TCP and UDP Classification


 The following configurations are used to control the TCP and UDP classification:

Set: EZapiICU_ConfigCmd_SetL4ControlProfile
 Get: EZapiICU_StatCmd_GetL4ControlProfile
 Struct: EZapiICU_L4ControlProfile

 Set: EZapiICU_ConfigCmd_SetL4ControlCOSParams
 Get: EZapiICU_StatCmd_GetL4ControlCOSParams
 Struct: EZapiICU_L4ControlCOSParams

5.1.8 General Purpose Rules Classification

The general purpose classification is used for unknown protocol types.

 The following configurations are used to control general purpose rules classification:

| | |
|---------|--|
| Set: | EZapiICU_ConfigCmd_SetGeneralPurposeProfile |
| Get: | EZapiICU_StatCmd_GetGeneralPurposeProfile |
| Struct: | EZapiICU_GeneralPurposeProfile |
| Set: | EZapiICU_ConfigCmd_SetGeneralPurposeRules |
| Get: | EZapiICU_StatCmd_GetGeneralPurposeRules |
| Struct: | EZapiICU_GeneralPurposeRules |
| Set: | EZapiICU_ConfigCmd_SetGeneralPurposeTCAMRule |
| Get: | EZapiICU_StatCmd_GetGeneralPurposeTCAMRule |
| Struct: | EZapiICU_GeneralPurposeTCAMRule |

5.2 Summary of Commands – ICU Group

Table 5-1. ICU Group routines

| API Routines |
|--------------------|
| EZapiICU_Config() |
| EZapiICU_Status() |

Table 5-2. Summary of ICU Group commands

This table provides a summary of the commands in the ICU Group, as well as the channel state in which each command may be performed.

| | Undefined | Created | Powered up | Initialized | Finalized | Running |
|--|-----------|----------|------------|-------------|-----------|----------|
| TOPIC & COMMAND | U | C | P | I | F | R |
| ICU Packet Parsing and Classification | | | | | | |
| Set: EZapiICU_ConfigCmd_SetEthParams | - | - | - | ✓ | - | - |
| Set: EZapiICU_ConfigCmd_SetInterlakenParams | - | - | - | ✓ | - | - |
| Set: EZapiICU_ConfigCmd_SetInterlakenProfile | - | - | - | ✓ | - | - |
| Set: EZapiICU_ConfigCmd_SetExternalClassificationProfile | - | - | - | ✓ | - | - |
| Set: EZapiICU_ConfigCmd_SetCOSToOUTPrioMappingProfile | - | - | - | ✓ | - | - |
| Set: EZapiICU_ConfigCmd_SetLoadBalanceProfile | - | - | - | ✓ | - | - |
| Get: EZapiICU_StatCmd_GetEthParams | - | - | - | ✓ | ✓ | ✓ |
| Get: EZapiICU_StatCmd_GetInterlakenParams | - | - | - | ✓ | ✓ | ✓ |
| Get: EZapiICU_StatCmd_GetInterlakenProfile | - | - | - | ✓ | ✓ | ✓ |
| Get: EZapiICU_StatCmd_GetExternalClassificationProfile | - | - | - | ✓ | ✓ | ✓ |
| Get: EZapiICU_StatCmd_GetCOSToOUTPrioMappingProfile | - | - | - | ✓ | ✓ | ✓ |
| Get: EZapiICU_StatCmd_GetLoadBalanceProfile | - | - | - | ✓ | ✓ | ✓ |
| ICU Format Encapsulation | U | C | P | I | F | R |
| Set: EZapiICU_ConfigCmd_SetEncapsulationFormatProfile | - | - | - | ✓ | - | - |
| Get: EZapiICU_StatCmd_GetEncapsulationFormatProfile | - | - | - | ✓ | ✓ | ✓ |
| Layer 2 Classification | U | C | P | I | F | R |
| Set: EZapiICU_ConfigCmd_SetL2ControlProfile | - | - | - | ✓ | - | - |
| Set: EZapiICU_ConfigCmd_SetMiscParams | - | - | - | ✓ | - | - |
| Set: EZapiICU_ConfigCmd_SetVLANEncapsulationCOSMappingProfile | - | - | - | ✓ | - | - |
| Set: EZapiICU_ConfigCmd_SetControlFramesCOSParams | - | - | - | ✓ | - | - |
| Set: EZapiICU_ConfigCmd_SetL2UserDefinedDestAddrAndEthertype | - | - | - | ✓ | - | - |
| Get: EZapiICU_StatCmd_GetL2ControlProfile | - | - | - | ✓ | ✓ | ✓ |
| Get: EZapiICU_StatCmd_GetMiscParams | - | - | - | ✓ | ✓ | ✓ |
| Get: EZapiICU_StatCmd_GetVLANEncapsulationCOSMappingProfile | - | - | - | ✓ | ✓ | ✓ |
| Get: EZapiICU_StatCmd_GetControlFramesCOSParams | - | - | - | ✓ | ✓ | ✓ |
| Get: EZapiICU_StatCmd_GetL2UserDefinedDestAddrAndEthertype | - | - | - | ✓ | ✓ | ✓ |
| Layer 3 MPLS Classification | U | C | P | I | F | R |
| Set: EZapiICU_ConfigCmd_SetMPLSProfile | - | - | - | ✓ | - | - |
| Set: EZapiICU_ConfigCmd_SetClassificationTables | - | - | - | ✓ | - | - |
| Set: EZapiICU_ConfigCmd_SetMPLSUniEncapsulationCOSMappingProfile | - | - | - | ✓ | - | - |

| TOPIC & COMMAND | | U | C | P | I | F | R |
|---|---|----------|----------|----------|----------|----------|----------|
| Set: | EZapilCU_ConfigCmd_SetMPLSMulEncapsulationCOSMappingProfile | - | - | - | ✓ | - | - |
| Get: | EZapilCU_StatCmd_GetMPLSProfile | - | - | - | ✓ | ✓ | ✓ |
| Get: | EZapilCU_StatCmd_GetClassificationTables | - | - | - | ✓ | ✓ | ✓ |
| Get: | EZapilCU_StatCmd_GetMPLSUniEncapsulationCOSMappingProfile | - | - | - | ✓ | ✓ | ✓ |
| Get: | EZapilCU_StatCmd_GetMPLSMulEncapsulationCOSMappingProfile | - | - | - | ✓ | ✓ | ✓ |
| Layer 3 IP Classification | | U | C | P | I | F | R |
| Set: | EZapilCU_ConfigCmd_SetIPv4Profile | - | - | - | ✓ | - | - |
| Set: | EZapilCU_ConfigCmd_SetIPv6Profile | - | - | - | ✓ | - | - |
| Set: | EZapilCU_ConfigCmd_SetL3IPv4ControlProfile | - | - | - | ✓ | - | - |
| Set: | EZapilCU_ConfigCmd_SetL3IPv6ControlProfile | - | - | - | ✓ | - | - |
| Set: | EZapilCU_ConfigCmd_SetL3ControlCOSParams | - | - | - | ✓ | - | - |
| Set: | EZapilCU_ConfigCmd_SetIPv4EncapsulationCOSMappingProfile | - | - | - | ✓ | - | - |
| Set: | EZapilCU_ConfigCmd_SetIPv6EncapsulationCOSMappingProfile | - | - | - | ✓ | - | - |
| Get: | EZapilCU_StatCmd_GetIPv4Profile | - | - | - | ✓ | ✓ | ✓ |
| Get: | EZapilCU_StatCmd_GetIPv6Profile | - | - | - | ✓ | ✓ | ✓ |
| Get: | EZapilCU_StatCmd_GetL3IPv4ControlProfile | - | - | - | ✓ | ✓ | ✓ |
| Get: | EZapilCU_StatCmd_GetL3IPv6ControlProfile | - | - | - | ✓ | ✓ | ✓ |
| Get: | EZapilCU_StatCmd_GetL3ControlCOSParams | - | - | - | ✓ | ✓ | ✓ |
| Get: | EZapilCU_StatCmd_GetIPv4EncapsulationCOSMappingProfile | - | - | - | ✓ | ✓ | ✓ |
| Get: | EZapilCU_StatCmd_GetIPv6EncapsulationCOSMappingProfile | - | - | - | ✓ | ✓ | ✓ |
| Layer 4 TCP and UDP Classification | | U | C | P | I | F | R |
| Set: | EZapilCU_ConfigCmd_SetL4ControlProfile | - | - | - | ✓ | - | - |
| Set: | EZapilCU_ConfigCmd_SetL4ControlCOSParams | - | - | - | ✓ | - | - |
| Get: | EZapilCU_StatCmd_GetL4ControlProfile | - | - | - | ✓ | ✓ | ✓ |
| Get: | EZapilCU_StatCmd_GetL4ControlCOSParams | - | - | - | ✓ | ✓ | ✓ |
| General Purpose Rules Classification | | U | C | P | I | F | R |
| Set: | EZapilCU_ConfigCmd_SetGeneralPurposeProfile | - | - | - | ✓ | - | - |
| Set: | EZapilCU_ConfigCmd_SetGeneralPurposeRules | - | - | - | ✓ | - | - |
| Set: | EZapilCU_ConfigCmd_SetGeneralPurposeTCAMRule | - | - | - | ✓ | - | - |
| Get: | EZapilCU_StatCmd_GetGeneralPurposeProfile | - | - | - | ✓ | ✓ | ✓ |
| Get: | EZapilCU_StatCmd_GetGeneralPurposeRules | - | - | - | ✓ | ✓ | ✓ |
| Get: | EZapilCU_StatCmd_GetGeneralPurposeTCAMRule | - | - | - | ✓ | ✓ | ✓ |

6. Flow Control Unit Group

The Flow Control Unit (FCU) Group contains API routines for configuration and management of the NPS device's embedded Flow Control Units.

6.1 Overview


The FCU Group performs memory resource management on the following resources:

- Data frame buffers allocated from the NPS internal memory
- Job descriptor buffers allocated from the NPS internal memory
- Data frame buffers allocated from the NPS external memory

The FCU Group performs accounting on these resources at several levels (described below) and reflects statuses for various units: RXIF, PMU, TXIF, and OOB.


6.1.1 FCU States

The FCU Group is an extension of the Channel Group, and thus does not have states of its own, but rather shares the Channel Group states (see “[Channel States](#)” section).

 For a summary of which FCU commands can be called during which states, see “[Summary of Commands – FCU Group](#)”.

6.1.2 Configuration of the FCU

The FCU Group provides commands to configure the FCU for memory resource management.

 The FCU resource accounting method and other general parameters are configured in the following manner:

| | |
|---------|--------------------------------------|
| Set: | EZapiFCU_ConfigCmd_SetGeneralParams |
| Get: | EZapiFCU_StatCmd_GetGeneralParams |
| Struct: | EZapiFCU_GeneralParams |
| Set: | EZapiFCU_ConfigCmd_SetResourceParams |
| Get: | EZapiFCU_StatCmd_GetResourceParams |
| Struct: | EZapiFCU_ResourceParams |


The FCU performs memory resource accounting at several levels as described below.

BudgetID:

Each packet is assigned with a unique budgetID (0 – 127). The FCU maintains a counter for each of the 128 budgetIDs. Each budgetID is mapped to a lower level entity called an *interface*.

 The FCU budgetID parameters and thresholds are configured in the following manner:


| | |
|---------|---------------------------------------|
| Set: | EZapiFCU_ConfigCmd_SetBudgetIDParams |
| Get: | EZapiFCU_StatCmd_GetBudgetIDParams |
| Struct: | EZapiFCU_BudgetIDParams |
| Set: | EZapiFCU_ConfigCmd_SetBudgetIDProfile |
| Get: | EZapiFCU_StatCmd_GetBudgetIDProfile |
| Struct: | EZapiFCU_BudgetIDProfile |

 The FCU budgetID counters are read in the following manner:

Get: EZapiFCU_StatCmd_GetBudgetIDCounter
 Struct: EZapiFCU_BudgetIDCounter


Interface:

Each budgetID (0 – 127) is mapped to one of the 64 interfaces (0 – 63). The first 48 interfaces are used for the physical network interfaces, the last 16 are used for other auxiliary interfaces. The FCU maintains a counter for each of the 64 interfaces. Each interface is mapped to a lower level entity called a *group*.

 The FCU interface parameters and thresholds are configured in the following manner:

Set: EZapiFCU_ConfigCmd_SetIFParams
 Get: EZapiFCU_StatCmd_GetIFParams
 Struct: EZapiFCU_IFParams


Set: EZapiFCU_ConfigCmd_SetIFProfile
 Get: EZapiFCU_StatCmd_GetIFProfile
 Struct: EZapiFCU_IFProfile

 The FCU interface counters are read in the following manner:

Get: EZapiFCU_StatCmd_GetIFCounter
 Struct: EZapiFCU_IFCounter

Group:

Each interface (0 – 63) is mapped to one of the 8 groups (0 – 7). The FCU maintains a counter for each of the 8 groups. Each group is mapped to a lower level entity called *global*.

 The FCU group parameters and thresholds are configured in the following manner:

Set: EZapiFCU_ConfigCmd_SetGroupParams
 Get: EZapiFCU_StatCmd_GetGroupParams
 Struct: EZapiFCU_GroupParams


Set: EZapiFCU_ConfigCmd_SetGroupThresholds
 Get: EZapiFCU_StatCmd_GetGroupThresholds
 Struct: EZapiFCU_GroupThresholds

 The FCU group counters are read in the following manner:

Get: EZapiFCU_StatCmd_GetGroupCounter
 Struct: EZapiFCU_GroupCounter

Global:

The global entity is the lowest entity. The FCU maintains a global counter and a total counter.

 The FCU global parameters and thresholds are configured in the following manner:

Set: EZapiFCU_ConfigCmd_SetGlobalParams
 Get: EZapiFCU_StatCmd_GetGlobalParams
 Struct: EZapiFCU_GlobalParams

Set: EZapiFCU_ConfigCmd_SetGlobalThresholds
 Get: EZapiFCU_StatCmd_GetGlobalThresholds
 Struct: EZapiFCU_GlobalThresholds

 The FCU global and total counters are read in the following manner:

```

Get:      EZapiFCU_StatCmd_GetGlobalCounter
Struct:   EZapiFCU_GlobalCounter

Get:      EZapiFCU_StatCmd_GetTotalCounter
Struct:   EZapiFCU_TotalCounter

```


Flow Control:

 The FCU priority flow control is configured and read in the following manner:

```

Set:      EZapiFCU_ConfigCmd_SetPriorityFCProfile
Get:      EZapiFCU_StatCmd_GetPriorityFCProfile
Struct:   EZapiFCU_PriorityFCProfile

```

 The FCU class-based flow control (CBFC) is configured and read in the following manner:

```

Set:      EZapiFCU_ConfigCmd_SetCBFCGlobalCounterThresholds
Get:      EZapiFCU_StatCmd_GetCBFCGlobalCounterThresholds
Struct:   EZapiFCU_CBFCGlobalCounterThresholds

Set:      EZapiFCU_ConfigCmd_SetCBFCMappingProfile
Get:      EZapiFCU_StatCmd_GetCBFCMappingProfile
Struct:   EZapiFCU_CBFCMappingProfile

```

6.1.3 Default Configuration of the FCU

The FCU is configured internally by EZcp for memory resources protection. A user can overwrite these configurations later. These configurations protect the memory resources from overflowing and prevent deadlocks but do not guarantee fairness between the interfaces. A user can disable this internal initialization by toggling `EZapiFCU_GeneralParams.bDefaultInit` field.

FCU default configurations consist of:

- Accounting is enabled on all resources: IMEM, EMEM and JD.
- Network interfaces mapping: budget IDs -> relevant interfaces -> group 0.
- Special interfaces mapping: loopback budget IDs -> interface 60 -> group 7, timestamp confirmation budget ID -> interface 61 -> group 7, PMU timers budget IDs -> interface 62 -> group 7.
- Network interfaces budget IDs: all budget IDs are mapped to budget ID profile 0, all thresholds of profile 0 are UNLIMITED except the Guarantee threshold which is set to 0.
- Network interfaces: all interfaces are mapped to interface profile 0, all thresholds of profile 0 are UNLIMITED except the Guarantee threshold which is set to 0.
- Group 0: all thresholds are UNLIMITED except the tail drop threshold which is set to 75% of the allocated resources.
- Global: all thresholds of group 0 are UNLIMITED.
- Special interfaces budget IDs: all special interfaces budget IDs are mapped to profile 0, all thresholds of profile 0 are UNLIMITED except the Guarantee threshold which is set to 0.
- Special interfaces:
 - loopback interfaces are mapped to interface profile 28, timestamp interfaces are mapped to interface profile 29, PMU timer interfaces are mapped to interface profile 30, all thresholds are UNLIMITED except the tail drop which is set to 15% of the allocated resources for loopback interfaces, 5% of the allocated resources for timestamp confirmation interfaces and 5% of the allocated resources for PMU timer interfaces.

- Group 7: all thresholds are UNLIMITED.
- Global: all thresholds of group 7 are UNLIMITED.

6.2 Summary of Commands – FCU Group

Table 6-1. FCU Group routines

| API Routines |
|--------------------|
| EZapiFCU_Config() |
| EZapiFCU_Status() |

Table 6-2. Summary of FCU Group commands

This table provides a summary of the commands in the FCU Group, as well as the channel state in which each command may be performed.

| | Undefined | Created | Powered up | Initialized | Finalized | Running |
|--|-----------|---------|------------|-------------|-----------|---------|
| TOPIC & COMMAND | U | C | P | I | F | R |
| FCU Configuration | | | | | | |
| Set: EZapiFCU_ConfigCmd_SetGeneralParams | - | ✓ | - | - | - | - |
| Set: EZapiFCU_ConfigCmd_SetResourceParams | - | - | - | ✓ | - | - |
| Set: EZapiFCU_ConfigCmd_SetBudgetIDParams | - | - | - | ✓ | - | - |
| Set: EZapiFCU_ConfigCmd_SetIFParams | - | - | - | ✓ | - | - |
| Set: EZapiFCU_ConfigCmd_SetGroupParams | - | - | - | ✓ | - | - |
| Set: EZapiFCU_ConfigCmd_SetGlobalParams | - | - | - | ✓ | - | - |
| Set: EZapiFCU_ConfigCmd_SetBudgetIDProfile | - | - | - | ✓ | - | - |
| Set: EZapiFCU_ConfigCmd_SetIFProfile | - | - | - | ✓ | - | - |
| Set: EZapiFCU_ConfigCmd_SetGroupThresholds | - | - | - | ✓ | - | - |
| Set: EZapiFCU_ConfigCmd_SetGlobalThresholds | - | - | - | ✓ | - | - |
| Set: EZapiFCU_ConfigCmd_SetPriorityFCProfile | - | - | - | ✓ | - | - |
| Set: EZapiFCU_ConfigCmd_SetCBFCGlobalCounterThresholds | - | - | - | ✓ | - | - |
| Set: EZapiFCU_ConfigCmd_SetCBFCMappingProfile | - | - | - | ✓ | - | - |
| Get: EZapiFCU_StatCmd_GetGeneralParams | - | ✓ | ✓ | ✓ | ✓ | ✓ |
| Get: EZapiFCU_StatCmd_GetResourceParams | - | - | - | ✓ | ✓ | ✓ |
| Get: EZapiFCU_StatCmd_GetBudgetIDParams | - | - | - | ✓ | ✓ | ✓ |
| Get: EZapiFCU_StatCmd_GetIFParams | - | - | - | ✓ | ✓ | ✓ |
| Get: EZapiFCU_StatCmd_GetGroupParams | - | - | - | ✓ | ✓ | ✓ |
| Get: EZapiFCU_StatCmd_GetGlobalParams | - | - | - | ✓ | ✓ | ✓ |
| Get: EZapiFCU_StatCmd_GetBudgetIDProfile | - | - | - | ✓ | ✓ | ✓ |
| Get: EZapiFCU_StatCmd_GetIFProfile | - | - | - | ✓ | ✓ | ✓ |
| Get: EZapiFCU_StatCmd_GetGroupThresholds | - | - | - | ✓ | ✓ | ✓ |
| Get: EZapiFCU_StatCmd_GetGlobalThresholds | - | - | - | ✓ | ✓ | ✓ |
| Get: EZapiFCU_StatCmd_GetPriorityFCProfile | - | - | - | ✓ | ✓ | ✓ |
| Get: EZapiFCU_StatCmd_GetCBFCGlobalCounterThresholds | - | - | - | ✓ | ✓ | ✓ |
| Get: EZapiFCU_StatCmd_GetCBFCMappingProfile | - | - | - | ✓ | ✓ | ✓ |
| Get: EZapiFCU_StatCmd_GetBudgetIDCounter | - | - | - | ✓ | ✓ | ✓ |
| Get: EZapiFCU_StatCmd_GetIFCounter | - | - | - | ✓ | ✓ | ✓ |
| Get: EZapiFCU_StatCmd_GetGroupCounter | - | - | - | ✓ | ✓ | ✓ |
| Get: EZapiFCU_StatCmd_GetGlobalCounter | - | - | - | ✓ | ✓ | ✓ |
| Get: EZapiFCU_StatCmd_GetTotalCounter | - | - | - | ✓ | ✓ | ✓ |

7. Traffic Manager Group

The TM Group contains API routines for configuring and managing the NPS device's internal traffic managers including their hierarchical scheduling and QoS parameters per level for each TM. The two TM engines are identical, yet each one may be configured individually.


7.1 Overview

The TM Group contains routines for configuration and management of the NP's traffic managers.

The TM Group supports commands to configure the various TM parameters and profiles for QOS management, such as Shaping, WFQ, priority, IPG, WRED and statistics. In addition, commands are provided to configure the 5-level hierarchical scheduling topology. Furthermore, parameters specific to each entity in the topology may be configured.

Parameters for each internal TM engine may be configured individually.

Default values are configured for most TM configuration parameters. Thus, there is no need to explicitly configure these items, unless defaults do not match required behavior. When changing configurations, it is recommended to first perform a get operation to retrieve the previous values, modify only the fields whose value needs to be modified to attain the required behavior, and then perform a set operation with the updated software structure.


 This overview does not detail the embedded TM features and capabilities. For more information, refer to the *NPS-400 Architectural Specifications*.

7.1.1 TM States

The TM Group is an extension of the Channel Group, and thus does not have states of its own, but rather shares the Channel Group states (see "[Channel States](#)" section).

The general TM parameters must be defined when the channel is in the created state – before initializing the channel.

While the channel is in the initialized/running state, the TM Group routines may be called to configure the various TM parameters, profiles, entity topology and entity parameters. In most cases, global and per-level QOS parameters can only be changed in the initialized state, while QOS profiles, entity topology and entity parameters may also be changed in the running state.

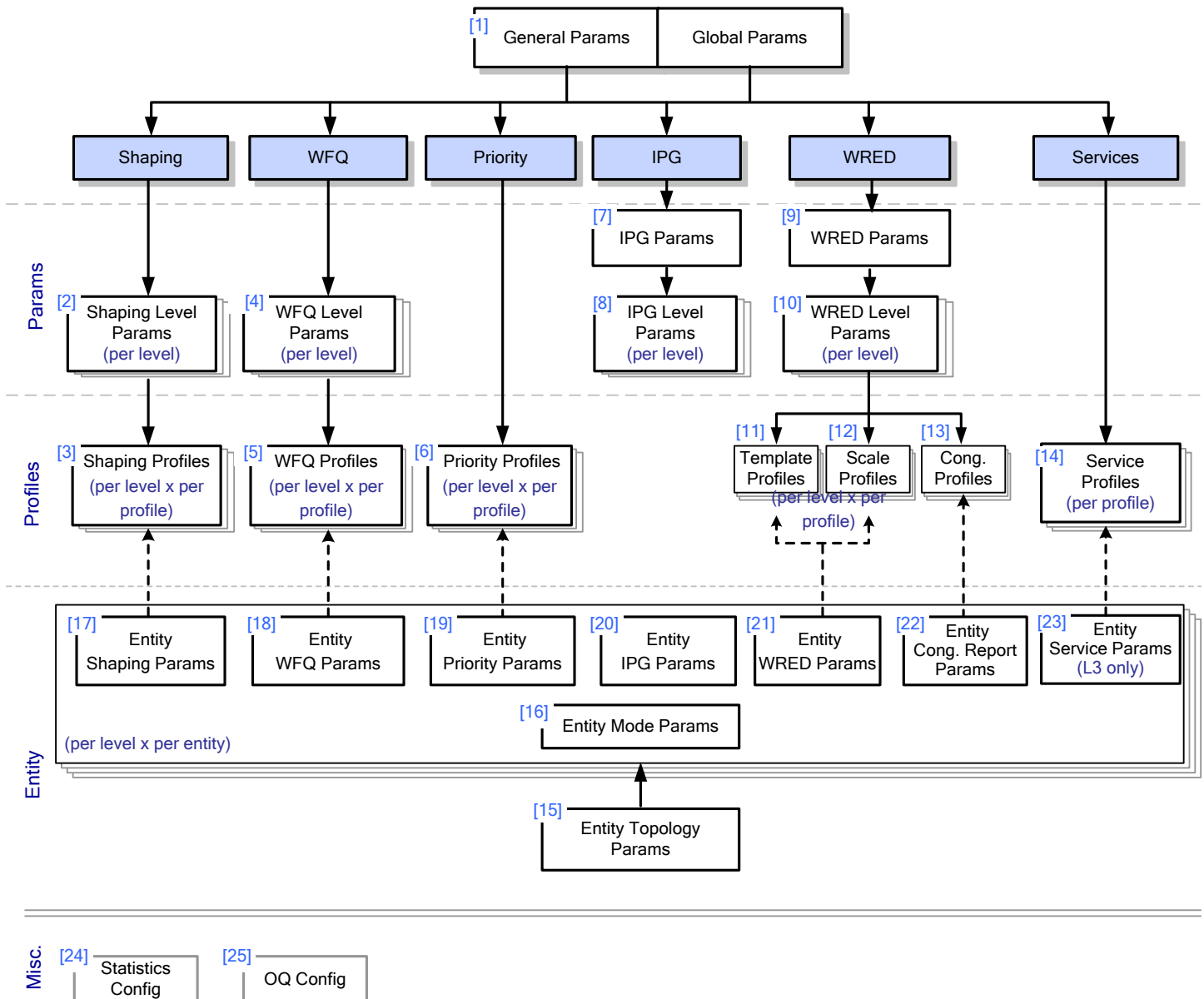
 For a summary of which TM Group commands can be called during which states, see "[Summary of Commands – TM Group](#)".

7.1.2 TM API Organization

The TM Group API routines and commands are logically organized in a combination approach: a vertical approach according to the hierarchical scheduling levels (from Level 0 to Level 4) and a horizontal approach per-topic (WRED, shaping, priority, WFQ, IPG and statistics).

The following diagram illustrates how the TM API group is organized. Throughout the following descriptions, blue numbers [x] refer to [Figure 7-1](#) below.

Figure 7-1. TM API Organization



Configuration of the TM begins with the **General Params** and **Global Params** [1] which configure parameters that have global affect on the TM behavior.

For each QOS topic (shaping, WFQ, priority, etc.) parameters are then configured in a vertical approach. First, for each topic, global and per-level parameters are configured. These affect the overall behavior for the topic (for all levels or for a specific level, respectively). Next, for each topic, profiles are configured

(for each profile in each level). Each profile defines one of several behaviors, and may later be associated to one or more entities. These include:

For **Shaping** these include:

- **Shaping Level Params [2]**: Includes several configurations that affect shaping behavior for each level, such as enabling/disabling of shaping (per-level).
- **Shaping Profiles [3]**: Shaping profiles define the information rate and burst size for shaping. Each profile can be used to define commit and/or excess shaping parameters. Each entity is assigned two independent profiles, one for commit shaping and one for excess shaping.

For **WFQ** these include:

- **WFQ Level Params [4]**: Includes several configurations that affect WFQ behavior for each level, such as the WFQ mode (for each level).
- **WFQ Profiles [5]**: WFQ profiles define the set of weights for each priority level.

For **Priority** these include:

- **Priority Profiles [6]**: Priority profiles define how priorities are propagated. Each profile defines a mapping table which selects the entity's priority based on the propagated priority (the priority of the entity's children) and the entity's shaping color.

For **IPG** these include:

- **IPG Param [7]**: Includes several configurations that affect global inter-packet gap behavior (for all levels), such as per-frame overhead exponents.
- **IPG Level Params [8]**: Includes several configurations that affect inter-packet gap behavior for each level, such as enabling/disabling IPG for WFQ and/or shaping (per-level).

For **WRED** these include:

- **WRED Params [9]**: Includes several configurations that affect global WRED behavior (for all levels), such as the hierarchical WRED decision modes.
- **WRED Level Params [10]**: Includes several configurations that affect WRED for each level, such as enabling/disabling of WRED (per-level).
- **WRED Template Profiles [11]**: WRED template profiles define the WRED drop behavior graph for the entity. The template profile is defined in percentiles of the TM frame resources allocated, and is coupled with a scale profile to define the actual absolute value of the TM frame resources allocated for the entity.
- **WRED Scale Profiles [12]**: WRED scale profiles define the absolute amount of allocated TM frame resources for the entity.
- **WRED Congestion Report Profiles [13]**: WRED congestion report profiles control the resolution of TM queue congestion/occupancy reporting.

In addition, several QOS parameters affecting L4 entities are configured in **Service Profiles [14]**. Service profiles are attached to L3 entities. Each service profile defines the behavior for the set of L4 entities mapped to the L3 entity.

In addition to configuring the various QOS parameters and profiles, the TM topology must be defined. This includes:

- **Entity Topology Params [15]**: Defines the topology mapping between the entities in the various TM levels. Each entity in levels 0-2 may have several sons in the next level.
 - ▶ *Each traffic management queue is mapped to five levels and the logical representation of each of these levels is called an "entity".*

The next step in the TM configuration is to assign each TM entity at each level its own set of parameters, such as assignments to the profiles defined in the previous steps:

- **Entity Mode Params [16]:** Configure if the entity is enabled/used.
- **Entity Shaping Params [17]:** Configure entity specific shaping attributes (shaping profile associations).
- **Entity WFQ Params [18]:** Configure entity specific WFQ attributes (WFQ profile association).
- **Entity Priority Params [19]:** Configure entity specific priority attributes (priority profile association).
- **Entity IPG Params [20]:** Configure entity specific IPG attributes (cell size, constant overhead, etc.)
- **Entity WRED Params [21]:** Configure entity specific WRED attributes (WRED profile associations).
- **Entity Congestion Report Params [22]:** Configure entity specific congestion reporting attributes (enable/disable).
- **Entity Service Params [23]:** Defines the mapping of L3 entities to service profiles.
- **Set Fast Queue Entity Rate Params:** Set rate range parameters for a fast-queue entity.

In addition, the TM API group supports configuration for several other TM-related blocks and features:

- **Statistics Configurations [24]:** Configuration of TM statistic counting functionality.
- **OQ Configurations [25]:** Configuration of the TM's Output Queue and Packet Switch ID mapping table.

7.1.3 TM Chunk Addressing

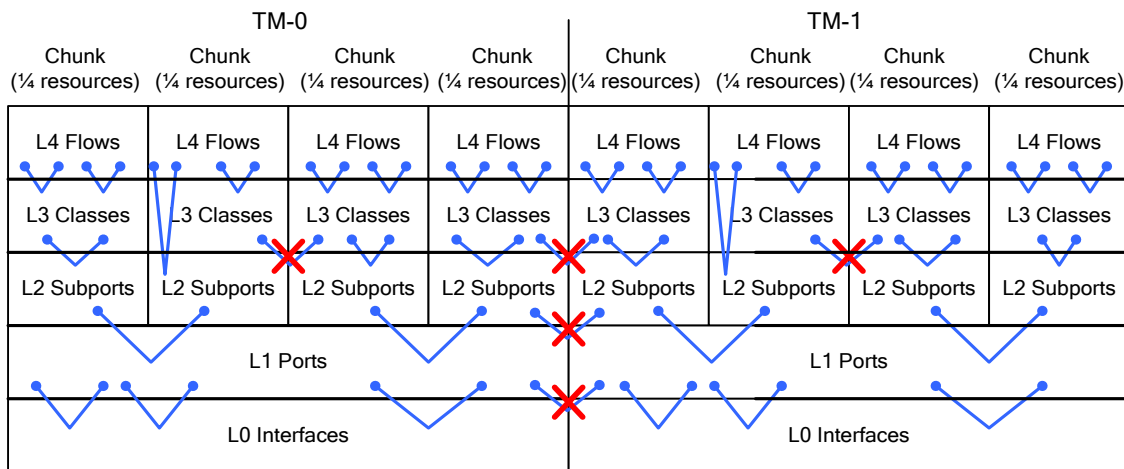
Although Port-to-Interface and Subport-to-Port mapping is fully flexible per TM engine, there is a restriction when it comes to the mapping of Classes to Subports per TM. The TM's mapping notation relates to a 'chunk' entity, which is basically a group of 1K Subports.

Each TM is evenly divided into four such chunks and each chunk holds one-quarter of the higher levels' resources (e.g., a chunk contains 1K Subports, 8K Classes and 128K Flows).

The mapping of L3 entities (classes) and their corresponding L4 entities (flows) to L2 entities (sub-ports) must remain within the chunk's boundaries. For example, L3 entities from the range 8K..16K may only be mapped to L2 entities from the range of 1K..2K.

The mapping of Classes (and their Flows) to Subports must remain within the chunk's boundaries. For example, a Class from the range 8K-16K may only be mapped to a Subport from the range of 1K-2K.

Figure 7-2. Each of the four chunks contains one-quarter of the Subports, Classes and Flows



The various QOS profiles are dedicated for each chunk (i.e. may be configured separately for each chunk), while the various QOS parameters (global and per-level) are identical for all chunks (e.g. are configured once for all chunks).

The TM API group includes a chunk ID parameter in many of the API structures (where relevant). The chunk ID parameter may be used in several manners:

- A value of 0 to 3 indicates that the operation should be performed on a single chunk.
- A value of EZapiTM_CHUNK_ID_ALL indicates that the operation should be performed on all 4 chunks.
- A value of EZapiTM_CHUNK_ID_NA indicates that the operation should be performed as if the TM were not divided into chunks. In this case, the TM topology is viewed as flat/chunkless (4K L2 entities, 32K L3 entities, etc.), with a single set of profiles shared/available for all entities.

The following table summarizes the semantics of the chunk ID throughout the API calls:

| | 0-3 | ALL | NA |
|-----------------------------------|--|--|---|
| QOS parameters (global/per-level) | Not relevant (parameter does not exist) | Not relevant (parameter does not exist) | Not relevant (parameter does not exist) |
| QOS profiles | Configures the profile only in the specified chunk. The profile ID is relative (within the chunk). | Configures the profile in all 4 chunks. The profile ID is relative (within the chunk). | Configures the profile in all 4 chunks. The profile ID is relative (within the chunk). |
| Entity parameters/mapping | Configures/maps the entity only in the specified chunk. Entity IDs are relative (within the chunk). | Configures/maps the entity in all 4 chunks. Entity IDs are relative (within the chunk). | Configures/maps a single entity. Entity IDs are absolute (across all chunks). |

7.1.4 TM Configuration Details


The following section provides more detailed information on TM configuration aspects, and should be regarded as an appendix to the information within the TM Group routines and structure definitions in the following sections.

7.1.4.1 TM Identifier

The NPS contains two TM engines that perform the traffic management functionality. Each TM engine can be used to transmit via any Interface Engine. Parameters for each internal TM maybe configured individually.

7.1.4.2 Global Configuration

The TM global params structure controls the allocation/partition of the external memory relevant for both TMs.


 The TM global parameters are configured in the following manner:

```
Set:      EZapiTM_ConfigCmd_SetGlobalParams
Get:      EZapiTM_StatCmd_GetGlobalParams
Struct:   EZapiTM_GlobalParams
```

7.1.4.3 General Configuration

The TM general params structure controls various general params for each TM including various aspects of the external memory partition.

- MaxL4Entities – This parameter specifies the maximum number of L4 entities to be used at any given time (regardless of the actual IDs). Reducing this value reduces the memory required for TM frame descriptor buffers, thus increasing the number of TM frame data buffers that may be allocated.
- Default init – Enable default initialization of TM topology and the OQ switching table ID parameters.
- Fast Queue parameters – Enable fast queues and determine which queues are used as fast queues.
- Enable Flows in IMEM Mode (Boolean) – Flow data (e.g. profiles, counters) reside in EMEM. It is possible to allocate memory in IMEM for some of the flows. FALSE means not using IMEM.
- TM Frame Buffers – This field can be used to limit the number of packet descriptors (PDs) used by a given TM assuming it does not need all frame buffers.
- IMEM Number of PDs – Additional PD-link queue where its PD buffers reside in internal memory can be defined for the use of each TMs' fast queues. This field states the number of PDs (if any) to allocate for the queue. By default all packet descriptors (PDs; one per packet) are allocated in EMEM. User might wish to allocate also in IMEM (for fast queues use). A value of 0 means no PDs in IMEM; this is not in place of EMEM PDs.
- IMEM PD Link Buffer Size – The PD-link buffer size to be used with the internal queue.

 The TM general parameters are configured in the following manner:

```
Set:      EZapiTM_ConfigCmd_SetGeneralParams
Get:      EZapiTM_StatCmd_GetGeneralParams
Struct:   EZapiTM_GeneralParams
```

7.1.4.4 Scheduling Configuration

 The TM general scheduling parameters are configured in the following manner:

```
Set:      EZapiTM_ConfigCmd_SetSchedulingParams
Get:      EZapiTM_StatCmd_GetSchedulingParams
Struct:   EZapiTM_SchedulingParams
```


7.1.4.5 Shaping Configuration

Shaping Parameters Configuration:

L2 and L3 introduce a mechanism allowing the definition of scale profiles and grid shaping which is a range of rates to be used by entities without the need to define profiles. Any entity can specify a rate for each of its buckets within this range.

Note that resources are limited; scale profiles and grid shaping range use same resources. The limitation is outlined below.

Scale profiles:

The number of scale profiles is not static and can be defined for each type of scale profile (with scaling of 1-256, or scaling of 1-1024).

- `uiScaleProfiles` – sets the number of scale profiles which an entity can scale up to 256, limited to up to 15 profiles.
- `uiExpandedScaleProfiles` – sets the number of scale profiles which an entity can scale up to 1024, limited to up to 3 profiles.

Grid shaping:

It is possible to define a dynamic range of rates which later an entity rate can be set to any rate within this range. Two types of ranges can be defined – normal accuracy range and high accuracy range where its accuracy error is half of that of the normal range and is no worse than 0.2%.

Intersection between the two ranges is allowed. On intersection, the high accuracy setting will be used.

- `uiGridStartIR`, `eGridStartResolution` – these two fields define the start rate range of the normal accuracy range.
- `uiGridExponent` – defines the upper limit range of the normal accuracy range according to following formula: $\text{start_rate} * 2^{\text{uiGridExponent}}$
- In the same manner `uiGridHighPrecStartIR`, `eGridHighPrecStartResolution` and `uiGridHighPrecExponent` define the high accuracy rate range.
- `uiGridTimeBurst` – defines the burst for the entity relative to the configured rate in terms of milliseconds of rate.

Level resources are limited and the limitation is outlined as:

$\text{uiScaleProfiles} * 2 + \text{uiExpandedScaleProfiles} * 8 + \text{uiGridExponent} + \text{uiGridHighPrecExponent} * 2 \leq 30$



The shaping parameters affecting each level are configured in the following manner:

Set: `EZapiTM_ConfigCmd_SetShapingLevelParams`
 Get: `EZapiTM_StatCmd_GetShapingLevelParams`
 Struct: `EZapiTM_ShapingLevelParams`

Shaping Profile Configuration:

Two types of shaping profiles are supported – explicit profiles and scale profiles. Explicit shaping profiles define a single rate (each entity mapped to the profile is shaped to the rate specified in the profile). Scale profiles define a base rate which can be scaled to multiples of 1-256 or 1-1024 according to the profile used. Note that the number of scale profiles should be defined in the `ShapingLevelParams` call. When mapping an entity to a scale profile scaling factor is also configured (per-entity).

Min Burst Size configuration guidelines:

The Min Burst Size [B] $\geq 1/8 * \text{Information Rate [bps]} * \text{Trefresh [seconds]}$

Where Trefresh: TM-L3: 500 microseconds

TM L0-2: 80 microseconds

Example L0 IR=1Gbps, Min BS[B] = $1/8 * 10^9 * 80 * 10^{-6} = 10000\text{B}$.

Formula for low-jitter minimal burst size:

For TM L4:

```
if IR<100Kbps then    MIN_BURST_SIZE=IR*32ms
else if IR<1Mbps then MIN_BURST_SIZE=IR*6720us
else if IR<10Mbps then MIN_BURST_SIZE=IR*672us
else MIN_BURST_SIZE=IR*67.2us
```

Example, if IR=8Mbps then $\text{MIN_BURST_SIZE} = 8\text{Mbps} * 672\text{us} = 5376\text{bit}$

For low-latency queues:

```
if IR<100Kbps then    MIN_BURST_SIZE=IR*32ms
else                  MIN_BURST_SIZE=IR*67.2us
```


There is also limitation for maximum burst size which must be kept: $\text{MAX_BURST_SIZE} = \text{IR} * 100\text{ms}$

Listed below are the refresh rates for the various RTC machines that are used in L0 through L3:

For L0-L2 refresh time is once every 7.8125us (128,000 refreshes per second).


For L3 refresh time is once every 48.828125us (20,480 refreshes per second).

Note that L3 is true for a system clock above 335MHz.

 The shaping profile configuration is done in the following manner:

```
Set:      EZapiTM_ConfigCmd_SetShapingProfile
          EZapiTM_ConfigCmd_SetShapingScaleProfile
          EZapiTM_ConfigCmd_SetShapingGridProfile
Get:      EZapiTM_StatCmd_GetShapingProfile
          EZapiTM_StatCmd_GetShapingScaleProfile
          EZapiTM_StatCmd_GetShapingGridProfile
Struct:   EZapiTM_ShapingProfile
```


In addition, a shaping update profile defines which shaper buckets are updated when a frame is scheduled based on the entity's current color and priority, allowing to control the shaper behavior.

 The shaping update profile configuration is done in the following manner:

```
Set:      EZapiTM_ConfigCmd_SetShapingUpdateProfile
Get:      EZapiTM_StatCmd_GetShapingUpdateProfile
Struct:   EZapiTM_ShapingUpdateProfile
```

L4 Shaping Machine Configuration:

L4 shaping machines perform refresh operations to L4 entity shapers. Five machines are supported; each can be configured to scan a subset of the entities. In addition, each of the machines may be used to periodically scan and activate video replay queues.


 The L4 shaping machine configuration is done in the following manner:

```
Set      EZapiTM_ConfigCmd_SetL4ShapingMachineParams
Get      EZapiTM_StatCmd_GetL4ShapingMachineParams
Struct:  EZapiTM_L4ShapingMachineParams
```


7.1.4.6 WFQ Configuration

WFQ Params configuration is per level, whereas WFQ Profile configuration is per entity.

- L23 Profiles Shared - When working in WFQ shared mode L2 & L3 entities can use all the 128 profiles. If not using shared mode, each level can use only 64 profiles (0-63).

 The WFQ parameters affecting a single level are configured in the following manner:


Set: EZapiTM_ConfigCmd_SetWFQLevelParams
Get: EZapiTM_StatCmd_GetWFQLevelParams
Struct: EZapiTM_WFQLevelParams

 The WFQ profile configuration is done in the following manner:


Set: EZapiTM_ConfigCmd_SetWFQProfile
Get: EZapiTM_StatCmd_GetWFQProfile
Struct: EZapiTM_WFQProfile

- ▶ *The field bDualWeightProfiles in the structure EZapiTM_WFQLevelParams should not be changed after updating/configuring WFQ profiles.*

7.1.4.7 Priority Configuration


 The priority profile configuration for levels 0-2 is done in the following manner:

Set: EZapiTM_ConfigCmd_SetPriorityProfile
Get: EZapiTM_StatCmd_GetPriorityProfile
Struct: EZapiTM_PriorityProfile


 The priority profile configuration for level 3 is done in the following manner:

Set: EZapiTM_ConfigCmd_SetL3PriorityProfile
Get: EZapiTM_StatCmd_GetL3PriorityProfile
Struct: EZapiTM_L3PriorityProfile

7.1.4.8 IPG Configuration

 The IPG parameters affecting all levels are configured in the following manner:


Set: EZapiTM_ConfigCmd_SetIPGParams
Get: EZapiTM_StatCmd_GetIPGParams
Struct: EZapiTM_IPGParams

 The IPG parameters affecting a single level are configured in the following manner:


Set: EZapiTM_ConfigCmd_SetIPGLevelParams
Get: EZapiTM_StatCmd_GetIPGLevelParams
Struct: EZapiTM_IPGLevelParams

7.1.4.9 WRED Configuration

WRED Parameters Configuration:

 The WRED parameters affecting all levels are configured in the following manner:

Set: EZapiTM_ConfigCmd_SetWREDParams
 Get: EZapiTM_StatCmd_GetWREDParams
 Struct: EZapiTM_WREDParams

 The WRED parameters affecting a single level are configured in the following manner:

Set: EZapiTM_ConfigCmd_SetWREDLevelParams
 Get: EZapiTM_StatCmd_GetWREDLevelParams
 Struct: EZapiTM_WREDLevelParams

- ▶ *The resolution for WRED counting may be controlled per level using the fields `bUseLevelCountMode` and `eCountMode` in the structure `EZapiTM_WREDLevelParams`. When used, these override the general configuration specified in the field `eCountMode` in the structure `EZapiTM_WREDParams`.*


WRED Profile Configuration:

 The WRED profile configuration is done in the following manner:

Set: EZapiTM_ConfigCmd_SetWREDTemplateProfile
 Get: EZapiTM_StatCmd_GetWREDTemplateProfile
 Struct: EZapiTM_WREDTemplateProfile

Set: EZapiTM_ConfigCmd_SetWREDScaleProfile
 Get: EZapiTM_StatCmd_GetWREDScaleProfile
 Struct: EZapiTM_WREDScaleProfile

- ▶ *For each color, the min threshold must be no greater than the mid threshold, the mid threshold must be no greater than the max threshold, and the mid drop percent must be no greater than the max drop percent.*

 The WRED congestion report profile is configured is done in the following manner:

Set: EZapiTM_ConfigCmd_SetWREDCongestionReportProfile
 Get: EZapiTM_StatCmd_GetWREDCongestionReportProfile
 Struct: EZapiTM_WREDCongestionReportProfile


Fast Queue WRED Configuration:

Each TM engine supports 128 queues for fast traffic scheduling. The 128 fast queues are directly mapped to 32 L1 ports (4 queues per L1 port). This direct mapping effectively bypasses the TM scheduling hierarchy and associated scheduling latency. The fast queues control structures are managed in internal memory.

Configuration of which queues are fast queues is done in the TM general params structure.

Configuration of the WFQ weights and strict priority for fast queue entities is done in the `EZapiTM_EntityWFQParams` and `EZapiTM_EntityPriorityParams` structures respectively.

Note that the fast queue entity configuration is done using the L4 absolute index of the fast queue and the level should be set to `EZapiTM_LEVEL_FAST_QUEUE` (wherever applicable).

 The fast-queue WRED configuration is done in the following manner:

Set: EZapiTM_ConfigCmd_SetFastQueueWREDThresholdProfile
 Get: EZapiTM_StatCmd_GetFastQueueWREDThresholdProfile

Struct: EZapiTM_FastQueueWREDThresholdProfile

Set: EZapiTM_ConfigCmd_SetFastQueueWREDSegmentProfile

Get: EZapiTM_StatCmd_GetFastQueueWREDSegmentProfile

Struct: EZapiTM_FastQueueWREDSegmentProfile

Set: EZapiTM_ConfigCmd_SetFastQueueEntityWREDParams

Get: EZapiTM_StatCmd_GetFastQueueEntityWREDParams

Struct: EZapiTM_FastQueueEntityWREDParams

WRED Reporting Configuration:


 Get WRED reporting address for optimized DP WRED reporting status read is done in the following manner:

Get: EZapiTM_StatCmd_GetReportingAddress

Struct: EZapiTM_ReportingAddress

Struct: EZapiTM_ReportingProfile

7.1.4.10 Service Profile Configuration

 The service profile configuration is done in the following manner:


Set: EZapiTM_ConfigCmd_SetServiceProfile

Get: EZapiTM_StatCmd_GetServiceProfile

Struct: EZapiTM_ServiceProfile

7.1.4.11 Entity Configuration

Entity Mode Params:

 The entity mode configuration is done in the following manner:

Set: EZapiTM_ConfigCmd_SetEntityModeParams

Get: EZapiTM_StatCmd_GetEntityModeParams

Struct: EZapiTM_EntityModeParams


- ▶ *An entity can only be enabled when its father (e.g. the entity it is mapped to) is enabled. Likewise, unmapped entities cannot be enabled.*
- ▶ *Disabled entities/flows will drop traffic sent to them.*

Entity Shaping Params:

L2 and L3 entities have extended shaping capabilities allowing each entity to use a scale shaping profile (if defined) or a rate within the grid shaping rate range (normal/high accuracy is according to the defined range of each in the shaping level params).

When using the grid profile type for the bucket, *uiGridIR* and *eGridIRResolution* will define the rate configured for this bucket.


When using the scale profile type for the bucket, *uiProfile* and *uiScale* will define the profile index and the scaling of the profile that this bucket will use.

 The entity shaping parameters configuration is done in the following manner:

Set: EZapiTM_ConfigCmd_SetEntityShapingParams

Get: EZapiTM_StatCmd_GetEntityShapingParams


Struct: EZapiTM_EntityShapingParams

 The L3 nibble shaping parameters configuration is done in the following manner:

Set: EZapiTM_ConfigCmd_SetL3NibbleShapingParams
 Get: EZapiTM_StatCmd_GetL3NibbleShapingParams
 Struct: EZapiTM_L3NibbleShapingParams


- ▶ *For levels that are configured with shaping disabled, all entities must be configured to use the always positive or always negative shaping profiles (EZapiTM_SHAPING_COUNTER_POSITIVE, EZapiTM_SHAPING_COUNTER_NEGATIVE).*
- ▶ *For L3 nibbles configured to work with 16 L4 entities per L3 entity, there are 2 shaping profiles and counters for each L3 entity (similar to all other levels). In this case, the L3 entities' shaping profiles can be configured using the standard EntityShapingParams params command/structure.*
- ▶ *For L3 nibbles configured to work with 32/64 L4 entities per L3 entity, there are 4/8 shaping profiles and counters for an L3 entity. In this case, the entities shaping profiles are configured using the L3NibbleShapingParams command/structure – which configures the 8 shaping profiles in the nibble.*
- ▶ *An entity may also specify to invoke a hardware shaper in conjunction with the standard shaper. For entities representing a physical link, it is recommended to enable the hardware shaper in conjunction with a standard shaper matching the interface's physical rate.*

Entity WFQ Params:

 The entity WFQ parameters configuration is done in the following manner:

Set: EZapiTM_ConfigCmd_SetEntityWFQParams
 Get: EZapiTM_StatCmd_GetEntityWFQParams
 Struct: EZapiTM_EntityWFQParams

Entity Priority Params:

 The entity priority parameters configuration is done in the following manner:

Set: EZapiTM_ConfigCmd_SetEntityPriorityParams
 Get: EZapiTM_StatCmd_GetEntityPriorityParams
 Struct: EZapiTM_EntityPriorityParams.

- ▶ *For levels 2 and 3, priority parameters are configured per nibble (4 entities). In this case, the entity priority parameters configuration should be performed to the first entity in the nibble (multiples of 4), but affects all 4 entities in the nibble.*


Entity IPG Params:

 The entity IPG parameters configuration is done in the following manner:

Set: EZapiTM_ConfigCmd_SetEntityIPGParams
 Get: EZapiTM_StatCmd_GetEntityIPGParams
 Struct: EZapiTM_EntityIPGParams

- ▶ *For level 2, IPG parameters are configured per nibble (4 entities). In this case, the entity priority parameters configuration should be performed to the first entity in the nibble (multiples of 4), but affects all 4 entities in the nibble.*


Entity WRED Params:

 The entity WRED parameters configuration is done in the following manner:

Set: EZapiTM_ConfigCmd_SetEntityWREDParams
 Get: EZapiTM_StatCmd_GetEntityWREDParams
 Struct: EZapiTM_EntityWREDParams


- ▶ *L3 and L4 WRED parameters are specified per frame in the TM header.*

Entity Congestion Report Params:

 The entity congestion report parameters configuration is done in the following manner:


Set: EZapiTM_ConfigCmd_SetEntityCongestionReportParams
Get: EZapiTM_StatCmd_GetEntityCongestionReportParams
Struct: EZapiTM_EntityCongestionReportParams

Entity Service Profile Params:

 The L3 entities (and matching L4 entities) are mapped to service profiles in the following manner:


Set: EZapiTM_ConfigCmd_SetL3EntityServiceParams
Get: EZapiTM_StatCmd_GetL3EntityServiceParams
Struct: EZapiTM_L3EntityServiceParams

Entity Fast Queue Rate Params:

 The rate range parameters for a fast-queue entity are configured in the following manner:

Set: EZapiTM_ConfigCmd_SetFastQueueEntityRateParams
Get: EZapiTM_StatCmd_GetFastQueueEntityRateParams
Struct: EZapiTM_FastQueueEntityRateParams

Queue Flush Configuration:


 The queue flush configuration is done in the following manner:

Set: EZapiTM_ConfigCmd_SetQueueFlushParams
Get: EZapiTM_StatCmd_GetQueueFlushParams
Struct: EZapiTM_QueueFlushParams

 The queue depth status is retrieved in the following manner:

Get: EZapiTM_StatCmd_GetQueueDepth
Struct: EZapiTM_QueueDepth


7.1.4.12 Topology Configuration

 Entity topology mapping is done in the following manner:

```
Set:      EZapiTM_ConfigCmd_AddEntityTopologyParams
Set:      EZapiTM_ConfigCmd_RemoveEntityTopologyParams
Get:      EZapiTM_StatCmd_GetEntityTopologyParams
Get:      EZapiTM_StatCmd_GetEntityTopologyParamsFirst
Get:      EZapiTM_StatCmd_GetEntityTopologyParamsNext
Get:      EZapiTM_StatCmd_GetEntityMappingParams
Struct:    EZapiTM_EntityTopologyParams
```

- ▶ *Each entity can only be mapped to a single “father” entity.*
- ▶ *L2 entities are mapped in nibbles (groups of 4). The first and number of L2 entities mapped to an L1 entity must be a multiple of 4.*
- ▶ *L3 entities are mapped in nibbles (groups of 4). The first and number of L3 entities mapped to an L2 entity must be a multiple of 4. When mapping L3 entities, the L4 entities associated with the L3 entities are also mapped (there is no need to explicitly map L4 entities). In addition, the L3 and L4 entities may be mapped in one of 3 modes:*
 - ▶ *16 L4 entities for each L3 entity – in this case all 4 L3 entities are available, each with 16 L4 entities. Each L3 entity (and related 16 L4 entities) supports 2 shaping counters/profiles, based on the L4 service profile configuration.*
 - ▶ *32 L4 entities for each L3 entity – in this case only the even numbered L3 entities are available (odd L3 entities cannot be enabled), each with 32 L4 entities. Each L3 entity (and related 32 L4 entities) supports 4 shaping counters/profiles, based on the L4 service profile configuration.*
 - ▶ *64 L4 entities for each L3 entity – in this case only the first L3 entity in each nibble is available (the other three L3 entities cannot be enabled), and all 64 L4 entities are mapped to this L3 entity. The L3 entity (and related 64 L4 entities) supports 8 shaping counters/profiles, based on the L4 service profile configuration.*
- ▶ *L3 entities may only be mapped to L2 entities within the same chunk. For example, L3 entities from the range 8K..16K may only be mapped to L2 entities from the range of 1K...2K.*

7.1.4.13 TM Statistics Configuration

 The TM statistics parameters configuration is done in the following manner:

```
Set:      EZapiTM_ConfigCmd_SetStatisticParams
Get:      EZapiTM_StatCmd_GetStatisticParams
Struct:    EZapiTM_StatisticParams
```

 The mapping of the statistics events to codes (offsets) is done in the following manner:


```
Set:      EZapiTM_ConfigCmd_SetStatisticEventCode
Get:      EZapiTM_StatCmd_GetStatisticEventCode
Struct:    EZapiTM_StatisticEventCode
```

 The TM and probed statistic counters can be read in the following manner:


```
Get:      EZapiTM_StatCmd_GetStatisticCounter
Struct:    EZapiTM_StatisticCounter
```

- ▶ *For TX and timeout events, codes are relevant for the per-level events only, global/probed statistics are using internal dedicated counters. For WRED events, the per-level codes are also used for global and probed WRED events (with 32 dedicated internal counters for WRED global/probed statistics). Note that global and probed statistics cannot be counted together.*


7.1.4.14 TM Output Queue (OQ) Configuration

 The current switch table in use (main or alternate) is done in the following manner:

Set: Set: EZapiTM_ConfigCmd_SetPacketSwitchParams
Get: EZapiTM_StatCmd_GetPacketSwitchParams
Struct: EZapiTM_PacketSwitchParams

 The mapping of the packet switch ID to the output queue is done in the following manner:

Set: EZapiTM_ConfigCmd_SetPacketSwitchIDParams
Get: EZapiTM_StatCmd_GetPacketSwitchIDParams
Struct: EZapiTM_PacketSwitchIDParams

 The configuration of the profile for the output queue backpressure mechanism is done in the following manner:

Set: EZapiTM_ConfigCmd_SetOQBPPProfile
Get: EZapiTM_StatCmd_GetOQBPPProfile
Struct: EZapiTM_OQBPPProfile

▶ *The default switch table ID mapping is a one-to-one mapping (in both switch tables).*

7.2 Summary of Commands – TM Group

Table 7-1. TM Group routines

| API Routines | |
|------------------|--|
| EZapiTM_Config() | |
| EZapiTM_Status() | |

Table 7-2. Summary of TM Group commands

This table provides a summary of the commands in the TM Group, as well as the channel state in which each command may be performed.

| | Undefined | Created | Powered up | Initialized | Finalized | Running |
|--|-----------|---------|------------|-------------|-----------|---------|
| TOPIC & COMMAND | U | C | P | I | F | R |
| General TM Configuration | | | | | | |
| General configuration | | | | | | |
| Set: EZapiTM_ConfigCmd_SetGlobalParams | - | ✓ | - | - | - | - |
| Get: EZapiTM_StatCmd_GetGlobalParams | - | ✓ | ✓ | ✓ | ✓ | ✓ |
| Set: EZapiTM_ConfigCmd_SetGeneralParams | - | ✓ | - | - | - | - |
| Get: EZapiTM_StatCmd_GetGeneralParams | - | ✓ | ✓ | ✓ | ✓ | ✓ |
| Scheduling configuration | | | | | | |
| Set: EZapiTM_ConfigCmd_SetSchedulingParams | - | ✓ | - | - | - | - |
| Get: EZapiTM_StatCmd_GetSchedulingParams | - | ✓ | ✓ | ✓ | ✓ | ✓ |
| QOS Configuration | | | | | | |
| Shaping configuration | | | | | | |
| Set: EZapiTM_ConfigCmd_SetShapingLevelParams | - | - | - | ✓ | - | - |
| Set: EZapiTM_ConfigCmd_SetShapingProfile | - | - | - | ✓ | ✓ | ✓ |
| Set: EZapiTM_ConfigCmd_SetShapingScaleProfile | - | - | - | ✓ | ✓ | ✓ |
| Set: EZapiTM_ConfigCmd_SetShapingGridProfile | - | - | - | ✓ | ✓ | ✓ |
| Set: EZapiTM_ConfigCmd_SetShapingUpdateProfile | - | - | - | ✓ | ✓ | ✓ |
| Set: EZapiTM_ConfigCmd_SetL4ShapingMachineParams | - | ✓ | - | - | - | - |
| Get: EZapiTM_StatCmd_GetShapingLevelParams | - | - | - | ✓ | ✓ | ✓ |
| Get: EZapiTM_StatCmd_GetShapingProfile | - | - | - | ✓ | ✓ | ✓ |
| Get: EZapiTM_StatCmd_GetShapingScaleProfile | - | - | - | ✓ | ✓ | ✓ |
| Get: EZapiTM_StatCmd_GetShapingGridProfile | - | - | - | ✓ | ✓ | ✓ |
| Get: EZapiTM_StatCmd_GetShapingUpdateProfile | - | - | - | ✓ | ✓ | ✓ |
| Get: EZapiTM_StatCmd_GetL4ShapingMachineParams | - | ✓ | ✓ | ✓ | ✓ | ✓ |
| WFQ configuration | | | | | | |
| Set: EZapiTM_ConfigCmd_SetWFQLevelParams | - | - | - | ✓ | - | - |
| Set: EZapiTM_ConfigCmd_SetWFQProfile | - | - | - | ✓ | ✓ | ✓ |
| Get: EZapiTM_StatCmd_GetWFQLevelParams | - | - | - | ✓ | ✓ | ✓ |
| Get: EZapiTM_StatCmd_GetWFQProfile | - | - | - | ✓ | ✓ | ✓ |
| Priority configuration | | | | | | |
| Set: EZapiTM_ConfigCmd_SetPriorityProfile | - | - | - | ✓ | ✓ | ✓ |
| Set: EZapiTM_ConfigCmd_SetL3PriorityProfile | - | - | - | ✓ | ✓ | ✓ |

| TOPIC & COMMAND | | U | C | P | I | F | R |
|--------------------------------|--|----------|----------|----------|----------|----------|----------|
| Get: | EZapiTM_StatCmd_GetPriorityProfile | - | - | - | ✓ | ✓ | ✓ |
| Get: | EZapiTM_StatCmd_GetL3PriorityProfile | - | - | - | ✓ | ✓ | ✓ |
| Inter Packet Gap configuration | | | | | | | |
| Set: | EZapiTM_ConfigCmd_SetIPGParams | - | - | - | ✓ | - | - |
| Set: | EZapiTM_ConfigCmd_SetIPGLevelParams | - | - | - | ✓ | - | - |
| Get: | EZapiTM_StatCmd_GetIPGParams | - | - | - | ✓ | ✓ | ✓ |
| Get: | EZapiTM_StatCmd_GetIPGLevelParams | - | - | - | ✓ | ✓ | ✓ |
| WRED configuration | | | | | | | |
| Set: | EZapiTM_ConfigCmd_SetWREDParams | - | - | - | ✓ | - | - |
| Set: | EZapiTM_ConfigCmd_SetWREDLevelParams | - | - | - | ✓ | - | - |
| Set: | EZapiTM_ConfigCmd_SetWREDTemplateProfile | - | - | - | ✓ | ✓ | ✓ |
| Set: | EZapiTM_ConfigCmd_SetWREDScaleProfile | - | - | - | ✓ | ✓ | ✓ |
| Set: | EZapiTM_ConfigCmd_SetWREDCongestionReportProfile | - | - | - | ✓ | ✓ | ✓ |
| Get: | EZapiTM_StatCmd_GetWREDParams | - | - | - | ✓ | ✓ | ✓ |
| Get: | EZapiTM_StatCmd_GetWREDLevelParams | - | - | - | ✓ | ✓ | ✓ |
| Get: | EZapiTM_StatCmd_GetWREDTemplateProfile | - | - | - | ✓ | ✓ | ✓ |
| Get: | EZapiTM_StatCmd_GetWREDScaleProfile | - | - | - | ✓ | ✓ | ✓ |
| Get: | EZapiTM_StatCmd_GetWREDCongestionReportProfile | - | - | - | ✓ | ✓ | ✓ |
| Get: | EZapiTM_StatCmd_GetReportingAddress | - | - | - | - | ✓ | ✓ |
| Fast-queue WRED configuration | | | | | | | |
| Set: | EZapiTM_ConfigCmd_SetFastQueueWREDThresholdProfile | - | - | - | ✓ | ✓ | ✓ |
| Set: | EZapiTM_ConfigCmd_SetFastQueueWREDSegmentProfile | - | - | - | ✓ | ✓ | ✓ |
| Set: | EZapiTM_ConfigCmd_SetFastQueueEntityWREDParams | - | - | - | ✓ | ✓ | ✓ |
| Get: | EZapiTM_StatCmd_GetFastQueueWREDThresholdProfile | - | - | - | ✓ | ✓ | ✓ |
| Get: | EZapiTM_StatCmd_GetFastQueueWREDSegmentProfile | - | - | - | ✓ | ✓ | ✓ |
| Get: | EZapiTM_StatCmd_GetFastQueueEntityWREDParams | - | - | - | ✓ | ✓ | ✓ |
| Service profile configuration | | | | | | | |
| Set: | EZapiTM_ConfigCmd_SetServiceProfile | - | - | - | ✓ | ✓ | ✓ |
| Get: | EZapiTM_StatCmd_GetServiceProfile | - | - | - | ✓ | ✓ | ✓ |
| Entity Configuration | | U | C | P | I | F | R |
| QOS configuration | | | | | | | |
| Set: | EZapiTM_ConfigCmd_SetEntityModeParams | - | - | - | ✓ | ✓ | ✓ |
| Set: | EZapiTM_ConfigCmd_SetEntityShapingParams | - | - | - | ✓ | ✓ | ✓ |
| Set: | EZapiTM_ConfigCmd_SetL3NibbleShapingParams | - | - | - | ✓ | ✓ | ✓ |
| Set: | EZapiTM_ConfigCmd_SetEntityWFQParams | - | - | - | ✓ | ✓ | ✓ |
| Set: | EZapiTM_ConfigCmd_SetEntityPriorityParams | - | - | - | ✓ | ✓ | ✓ |
| Set: | EZapiTM_ConfigCmd_SetEntityIPGParams | - | - | - | ✓ | ✓ | ✓ |
| Set: | EZapiTM_ConfigCmd_SetFastQueueEntityRateParams | | | | ✓ | ✓ | ✓ |
| Set: | EZapiTM_ConfigCmd_SetEntityWREDParams | - | - | - | ✓ | ✓ | ✓ |
| Set: | EZapiTM_ConfigCmd_SetEntityCongestionReportParams | - | - | - | ✓ | ✓ | ✓ |
| Set: | EZapiTM_ConfigCmd_SetL3EntityServiceParams | | | | ✓ | ✓ | ✓ |
| Get: | EZapiTM_StatCmd_GetEntityModeParams | - | - | - | ✓ | ✓ | ✓ |
| Get: | EZapiTM_StatCmd_GetEntityShapingParams | - | - | - | ✓ | ✓ | ✓ |
| Get: | EZapiTM_StatCmd_GetL3NibbleShapingParams | - | - | - | ✓ | ✓ | ✓ |
| Get: | EZapiTM_StatCmd_GetEntityWFQParams | - | - | - | ✓ | ✓ | ✓ |

| TOPIC & COMMAND | | U | C | P | I | F | R |
|--------------------------------------|---|----------|----------|----------|----------|----------|----------|
| Get: | EZapiTM_StatCmd_GetEntityPriorityParams | - | - | - | ✓ | ✓ | ✓ |
| Get: | EZapiTM_StatCmd_GetEntityIPGParams | - | - | - | ✓ | ✓ | ✓ |
| Get: | EZapiTM_StatCmd_GetFastQueueEntityRateParams | - | - | - | ✓ | ✓ | ✓ |
| Get: | EZapiTM_StatCmd_GetEntityWREDParams | - | - | - | ✓ | ✓ | ✓ |
| Get: | EZapiTM_StatCmd_GetEntityCongestionReportParams | - | - | - | ✓ | ✓ | ✓ |
| Get: | EZapiTM_StatCmd_GetL3EntityServiceParams | - | - | - | ✓ | ✓ | ✓ |
| Topology configuration | | | | | | | |
| Set: | EZapiTM_ConfigCmd_AddEntityTopologyParams | - | - | - | ✓ | ✓ | ✓ |
| Set: | EZapiTM_ConfigCmd_RemoveEntityTopologyParams | - | - | - | ✓ | ✓ | ✓ |
| Get: | EZapiTM_StatCmd_GetEntityTopologyParams | - | - | - | ✓ | ✓ | ✓ |
| Get: | EZapiTM_StatCmd_GetEntityTopologyParamsFirst | - | - | - | ✓ | ✓ | ✓ |
| Get: | EZapiTM_StatCmd_GetEntityTopologyParamsNext | - | - | - | ✓ | ✓ | ✓ |
| Get: | EZapiTM_StatCmd_GetEntityMappingParams | - | - | - | ✓ | ✓ | ✓ |
| Queue depth status | | | | | | | |
| Get: | EZapiTM_StatCmd_GetQueueDepth | - | - | - | ✓ | ✓ | ✓ |
| Queue flush configuration | | | | | | | |
| Set: | EZapiTM_ConfigCmd_SetQueueFlushParams | - | - | - | - | - | ✓ |
| Get: | EZapiTM_StatCmd_GetQueueFlushParams | - | - | - | - | - | ✓ |
| TM Statistic Configuration | | U | C | P | I | F | R |
| Set: | EZapiTM_ConfigCmd_SetStatisticParams | - | - | - | ✓ | ✓ | ✓ |
| Set: | EZapiTM_ConfigCmd_SetStatisticEventCode | - | - | - | ✓ | ✓ | ✓ |
| Get: | EZapiTM_StatCmd_GetStatisticParams | - | - | - | ✓ | ✓ | ✓ |
| Get: | EZapiTM_StatCmd_GetStatisticEventCode | - | - | - | ✓ | ✓ | ✓ |
| Get: | EZapiTM_StatCmd_GetStatisticCounter | - | - | - | ✓ | ✓ | ✓ |
| TM Output Queue Configuration | | U | C | P | I | F | R |
| TM OQ configuration | | | | | | | |
| Set: | EZapiTM_ConfigCmd_SetOQBPPProfile | - | - | - | ✓ | ✓ | ✓ |
| Get: | EZapiTM_StatCmd_GetOQBPPProfile | - | - | - | ✓ | ✓ | ✓ |
| Packet Switch Configuration | | U | C | P | I | F | R |
| Set: | EZapiTM_ConfigCmd_SetPacketSwitchParams | - | - | - | ✓ | ✓ | ✓ |
| Set: | EZapiTM_ConfigCmd_SetPacketSwitchIDParams | - | - | - | ✓ | ✓ | ✓ |
| Get: | EZapiTM_StatCmd_GetPacketSwitchParams | - | - | - | ✓ | ✓ | ✓ |
| Get: | EZapiTM_StatCmd_GetPacketSwitchIDParams | - | - | - | ✓ | ✓ | ✓ |

8. Statistics Group

The Statistics Group contains API routines for configuration and operating with the NPS device's statistic counter accelerations.

8.1 Overview

The Statistics Group contains API routines for configuration and management of on-demand and posted statistics counters.

The CP library allocates the statistics groups contiguously in the statistics memory segment based on their respective memory interfaces (e.g. statistics internal SRAM, EMEM).

8.1.1 Statistics States

The Statistics Group is an extension of the Channel Group, and thus does not have states of its own, but rather shares the Channel Group states (see "[Channel States](#)" section).


When the channel is initially created, all statistics groups are inactive, all configurations/profiles have default values, and all statistics machines are disabled.

Statistics groups must be defined when the channel is in the created state. When the channel is initialized (by calling **EZapiChannel_Initialize()**), the statistics group configurations are transferred to the NP, and all counters within the specified groups are initialized to default values. Counters in standard groups are initialized as long.

When the channel is in the initialized state, the Statistics Group routines may be called to initialize the statistic counters to non-default values, as well as to configure the various statistics related machines and profiles. Statistics machine configurations are generally cached in the CP library, and only transferred to the NP during the initialized state.

When the channel is in the running state, the Statistics Group routines may be called to operate with the statistics counters and profiles at runtime.

Most statistics machine parameters should not be changed when the machine is active. Thus, the correct sequence for changing the machine configuration parameters after a call to the **EZapiChannel_Go()** routine is to first disable the machine, and then reconfigure and enable the machine.

 For a summary of which Statistics commands can be called during which states, see "[Summary of Commands – Stat Group](#)".

8.1.2 On-demand Counters

The handling of on-demand counters and posted counters is completely independent in the Statistics Group API. All posted counter configuration commands in the Statistics Group contain the word “Posted” in the command name. All other commands relate to on-demand counters.

The Statistics Group supports commands to define statistics groups. Statistics groups define the division of a statistic partition into ranges of counters, with each group configured to operate in a memory area. Each of the eight on-demand partitions may be configured with up to 8 statistics groups.

Statistics counters may be stored in internal dedicated SRAM (128 Kbytes) or external DDR memory (EMEM).

The counters within statistics groups of type standard may be configured as these counter types: long, double, bitwise, watchdog, and token bucket. There are also separate group types for specialized token buckets.

Once statistics groups are defined, the Statistics Group API commands configure and operate with the statistics counters and associated profiles (where relevant). A dedicated set of structures and commands is provided for operating with each counter type. The statistic counter commands may be used to configure a single counter, configure a range of sequential counters to the same value, or configure an arbitrary set of counters to different values, all within a single API command. Similarly, the API supports commands to read a single counter, read a range of sequential counters, or read an arbitrary set of counters, all within a single API command.

The Statistics Group also provides commands for configuring various additional configurations affecting and/or related to the Statistics block, such as the general purpose scan machines.

Default values are configured for most Statistics block configuration items. There is no need to explicitly configure these items, unless defaults do not match required behavior. When changing defaults, it is recommended to first perform a get operation to retrieve the previous (default) values, then modify only the fields whose values need to be modified to attain the required behavior, and then perform a set operation with the updated software structure.

8.1.2.1 Partition Parameters

The partition is a logical entity which represents a counter space. There can be up to 8 partitions. Each partition can be spread across one or more on demand units.

The following parameters are defined:

- Partition – Indicates which partition will be configured.
- Enable – Enabling partition.
- Application Mask – NPS can run multiple applications where each cluster may run a different application. The application mask indicates to which applications the partition is relevant.
- Unit Mask – Mask of on demand hardware units
 - Bits 0-3 represent south, 4-8 represent north.
 - An on demand unit can only be part of a single partition.
 - Number of units which can share a partition can be 1, 2, 4 or 8.
- MSID – The memory space ID used by the CTOPs to access the counter range represented by the partition.



Statistic partition parameter configuration is done in the following manner:

```
Set:      EZapiStat_ConfigCmd_SetPartitionParams
Get:      EZapiStat_StatCmd_GetPartitionParams
Struct:   EZapiStat_PartitionParams
```

- ▶ The statistic partition parameter configuration must be performed while the channel is in the created state – before the call to **EZapiChannel_Initialize()**.

8.1.2.2 Group Types and Counter Types

Each on-demand partition may be configured with up to 8 statistics groups. Which of the physical on-demand units are configured depends on the unit mask of the partition.

Each group needs to be assigned a start counter and number of counters within the partition's logical address range. Counter groups do not need to be consecutive.

Each group needs to be assigned one of the group types listed below, and then counters need to be assigned to the groups.

Possible group types:

- Standard group – Each counter assigned to a standard group can be of a separate counter type:
 - Long counters – 59 bit counter value.
 - Double counters – Support byte and frame accounting in a single counter. Each double counter supports 96 data bits, with byte and frame portions.
 - Bitwise counters – Support bitwise operations on a subset of bits in the counters. This can be used to implement semaphores, state machines or lookup structures with small results.
 - Watchdog counters – Monitor the number of event in a period of time.
 - Token Bucket/DiffServ counters – Used for measuring traffic rates and marking packets using one of several protocols.
- Optimized Token Bucket group – Functionally equivalent to Token Bucket/DiffServ counters but with the color bits (2 or 4 bits) stored within the dedicated internal statistics memory for improved response time, and the actual buckets are stored separately in EMEM.
 - Only Token Bucket type counters may be assigned to this group which will in effect make them Optimized Token Buckets.
- Hierarchical Token Bucket group – A multi token bucket structure where several counters may be updated by each token bucket command.
 - Only Hierarchical Token Bucket type counters may be assigned to this group.
- Optimized Hierarchical Token Bucket group – Functionally equivalent to a Hierarchical Token Buckets but with the color bits stored within the dedicated internal statistics memory for improved response time.
 - Only Hierarchical Token Bucket type counters may be assigned to this group which will in effect make them Optimized Hierarchical Token Buckets.

Table 8-1. Matrix of statistic counter groups and types supported

| COUNTER TYPE | GROUP TYPE | | | |
|---------------------------|------------|---------------------------|------------------------------|---|
| | STANDARD | OPTIMIZED TOKEN BUCKET | HIERARCHICAL TOKEN BUCKET | OPTIMIZED HIERARCHICAL TOKEN BUCKET |
| Long counter | Supported | -- | -- | -- |
| Double counter | Supported | -- | -- | -- |
| Bitwise counter | Supported | -- | -- | -- |
| Token Bucket | Supported | Supported | -- | -- |
| Hierarchical Token Bucket | -- | -- | Supported | Supported |
| Watchdog counter | Supported | -- | -- | -- |
| General purpose counter | Supported | -- | -- | |

Configuration dictates the memory interface in which the group will reside:

- EMEM
- INTERNAL_SRAM – Dedicated internal SRAM. Each on-demand unit has a dedicated INTERNAL_SRAM of 2K lines x 128 bits not including ECC, unlike EMEM that is shared by all the units. Each line in the dedicated INTERNAL_SRAM can be a counter, or Optimized Token Bucket colors.

For each counter group, the following parameters are defined:

- Group type – The type of the group (Standard, Optimized Token Bucket, Hierarchical Token Bucket, Optimized Hierarchical Token Bucket).
- Memory type – The type of memory in which these counters are stored (dedicated internal SRAM or EMEM).
- Start counter – The first counter in the group.
- Num counters – The number of counters in the group.
- Enable shadow counters – Supported only for standard counter groups that contain only long and/or double counters.



Statistics group configuration is done in the following manner:

Set: EZapiStat_ConfigCmd_SetGroupParams
 Get: EZapiStat_StatCmd_GetGroupParams
 Struct: EZapiStat_GroupParams

See also “[On-demand General Counter Operations](#)”.

- ▶ *The statistics group configuration must be performed while the channel is in the created state – before the call to **EZapiChannel_Initialize()**.*
- ▶ *The statistics group configuration is transferred to the NPS in the initialize routine.*
- ▶ *The start counter and number of counters must be multiples of 32.*
- ▶ *Groups which have shadow counters enabled effectively use twice the configured size. The shadow counters are located immediately after the main counters (within the same partition).*



Statistic shadow group configuration is done in the following manner:


Set: EZapiStat_ConfigCmd_SetShadowGroupParams
 Get: EZapiStat_StatCmd_GetShadowGroupParams
 Struct: EZapiStat_ShadowGroupParams

8.1.2.3 Long Counters

Statistics long counters maintain a 59 bit value.

The Statistics API Group provides commands to initialize long counters to user-defined values, reset long counters, increment/decrement long counters, as well as to read the current value of long counters.

In addition, statistics on-demand general purpose machines may be configured to scan the counters in the background and periodically reset the counters and generate messages to the message queues with the counter values (before the reset).

 Long counter configuration is done in the following manner:

```
Set:      EZapiStat_ConfigCmd_SetLongCounters
          EZapiStat_ConfigCmd_SetLongCounterValues
          EZapiStat_ConfigCmd_IncrementLongCounters
          EZapiStat_ConfigCmd_DecrementLongCounters
          EZapiStat_ConfigCmd_ResetLongCounters
          EZapiStat_ConfigCmd_CondDecrementLongCounters
Get:      EZapiStat_StatCmd_GetLongCounters
          EZapiStat_StatCmd_GetLongCounterValues
Struct:   EZapiStat_LongCounterConfig
```


- ▶ *The EZapiStat_LongCounterConfig structure can be used for operation with a single statistic counter, with a sequential range of statistic counters or with a number of non-sequential counters.*
- ▶ *The counters specified in the EZapiStat_LongCounterConfig structure must be a subset of a standard statistic counter group that was previously defined.*

8.1.2.4 Double Counters

Statistics double counters support byte and frame accounting in a single counter. Each double counter supports 96 data bits, with a configurable bit division between the byte and frame portions.

The Statistics API Group provides commands to initialize double counters to user-defined values, reset double counters, increment/decrement double counters, as well as to read the current value of double counters.

In addition, statistics on-demand general purpose machines may be configured to scan the counters in the background and periodically reset the counters and generate messages to the message queues with the counter values (before the reset).

 Double counter configuration is done in the following manner:

```
Set:      EZapiStat_ConfigCmd_SetDoubleCounters
          EZapiStat_ConfigCmd_IncrementDoubleCounters
          EZapiStat_ConfigCmd_DecrementDoubleCounters
          EZapiStat_ConfigCmd_ResetDoubleCounters
Get:      EZapiStat_StatCmd_GetDoubleCounters
Struct:   EZapiStat_DoubleCounterConfig
```


- ▶ *The EZapiStat_DoubleCounterConfig structure can be used for operation with a single statistic double counter, with a sequential range of statistic double counters or with a number of non-sequential double counters.*
- ▶ *The counters specified in the EZapiStat_DoubleCounterConfig structure must be a subset of a standard statistic counter group that was previously defined.*

8.1.2.5 Bitwise Counters

Bitwise counters support bitwise operations on a subset of bits in the counters. This can be used to implement semaphores, state machines or lookup structures with small results.

Each bitwise counter is comprised of 64 bits. Bitwise counters support performing bitwise set, get, clear, write, increment and decrement operations on any aligned subset of 1, 2, 4, 8 or 16 bits of the counter. In addition, conditional set operations may be performed on any aligned subset of 1, 2 or 4 bits of the counter.

Bitwise counters also support general 64-bit read/write operations.

 Bitwise counter configuration is done in the following manner:

| | |
|---------|--|
| Set: | EZapiStat_ConfigCmd_SetBitwiseCounters |
| | EZapiStat_ConfigCmd_SetBitsBitwiseCounters |
| | EZapiStat_ConfigCmd_ClearBitsBitwiseCounters |
| | EZapiStat_ConfigCmd_WriteBitsBitwiseCounters |
| | EZapiStat_ConfigCmd_IncrementBitsBitwiseCounters |
| | EZapiStat_ConfigCmd_DecrementBitsBitwiseCounters |
| | EZapiStat_ConfigCmd_CondSetBitsBitwiseCounters |
| Get: | EZapiStat_StatCmd_GetBitwiseCounters |
| | EZapiStat_StatCmd_GetBitsBitwiseCounters |
| Struct: | EZapiStat_BitwiseCounterConfig |


8.1.2.6 Token Bucket Profiles

Token Bucket profiles are used to define the parameters for the Token Bucket counters. 1024 profiles are supported for each statistics partition. The Token Bucket algorithm is part of the counter, not the profile.

For each profile, the following parameters are defined:

- Committed Information Rate (CIR) – in bytes per seconds.
- Committed Burst Size (CBS) – in bytes.
- Peak Information Rate (PIR) – in bytes per seconds.
- Peak Burst Size (PBS) or Excess Burst Size (EBS) – in bytes.
- srTCM Excess and Commit – when the Peak fields in the profile are used for the srTCM algorithm, the srTCM Excess field must be enabled, while the srTCM Commit field indicates the matching commit number for the algorithm.

Once initialized, each profile can be associated to one or more Token Buckets. Profiles may be associated to Token Bucket counters of all the various types, and counters of varying types may share profiles.

 Token Bucket profile configuration is done in the following manner:

```
Set:      EZapiStat_ConfigCmd_SetTokenBucketProfile
Get:      EZapiStat_StatCmd_GetTokenBucketProfile
Struct:   EZapiStat_TBProfile
```

- ▶ *The valid range for CIR/PIR is 1 byte/sec - 100Gbyte/sec.*
- ▶ *The valid range for CBS/PBS is 1-0xFFFFFFFF bytes.*
- ▶ *The token bucket profile associated with a token bucket must be configured before configuring the counter.*
- ▶ *By default token bucket counters are associated with token bucket profile #0.*

8.1.2.7 Token Bucket Counters

Token Bucket counters are used for measuring traffic rates and marking packets using one of several protocols. Token Buckets are configured with an associated profile that defines the parameters for the token bucket algorithms used.

 See “[Token Bucket Profiles](#)” for more details.

 Token Bucket counter configuration is done in the following manner:

```
Set:      EZapiStat_ConfigCmd_SetTokenBucketCounters
Get:      EZapiStat_StatCmd_GetTokenBucketCounters
          EZapiStat_StatCmd_GetTokenBucketColor
          EZapiStat_StatCmd_CheckTokenBucketColor
Struct:   EZapiStat_TBCounterConfig
```

- ▶ *The EZapiStat_TBCounterConfig structure can be used for operation with a single token bucket, with a sequential range of token buckets or with a number of non-sequential token buckets.*
- ▶ *The counters specified in the EZapiStat_TBCounterConfig structure must be a subset of a standard group that was previously defined.*
- ▶ *The token bucket profile associated with a token bucket must be configured before configuring the token bucket.*
- ▶ *All token bucket counters must be scanned using a refresh machine (token-bucket-enabled general purpose machine).*

8.1.2.8 Optimized Token Bucket Counters

When Token Bucket type counters are assigned to an Optimized Token Bucket type group, they in affect become Optimized Token Bucket counters. The Optimized Token Bucket storage is split between the internal statistics memory and the EMEM. The color bits (2 or 4 bits) are stored within the internal statistics memory for an improved response time, and the actual buckets are stored separately in EMEM.


Group Type should be configured to EZapiStat_GroupType_OPT_TB.

Token Bucket counters are configured with an associated profile; see “[Token Bucket Profiles](#)”.

8.1.2.9 Hierarchical Token Bucket Counters

Hierarchical Token Bucket counters implement operations on two hierarchical token bucket counters in a single command. The statistics block reads the two counters, returns the metering color according to the color of both counters, and updates the counter states. The color returned and counters updated are both configurable, supporting implementation of multiple schemes.

Group Type should be configured to EZapiStat_GroupType_HIER_TB.

 Hierarchical Token Bucket counter configuration is done in the following manner:

| | |
|---------|---|
| Set: | EZapiStat_ConfigCmd_SetHierarchicalTokenBucketCounters |
| Set: | EZapiStat_ConfigCmd_UpdateHierarchicalTokenBucketCounters |
| Get: | EZapiStat_StatCmd_GetHierarchicalTokenBucketCounters |
| Get: | EZapiStat_StatCmd_GetHierarchicalTokenBucketColor |
| Struct: | EZapiStat_HIER_TBCounterConfig |

Token Bucket counters are configured with an associated profile; see “[Token Bucket Profiles](#)”.

8.1.2.10 Optimized Hierarchical Token Bucket Counters

When Hierarchical Token Bucket type counters are assigned to an Optimized Hierarchical Token Bucket type group, they in affect become Optimized Hierarchical Token Bucket counters. The Optimized Hierarchical Token Bucket storage is split between the internal statistics memory and the EMEM. The color bits are stored within the internal statistics memory for an improved response time, and the actual buckets are stored separately in EMEM.


Group Type should be configured to EZapiStat_GroupType_OPT_HIER_TB.

Token Bucket counters are configured with an associated profile; see “[Token Bucket Profiles](#)”.

8.1.2.11 Watchdog Profiles

Watchdog profiles are used to define the behavior and thresholds for the watchdog counters. Eight watchdog counter profiles are supported for each statistics partition.


Once initialized, each profile can be associated to one or more watchdog counters.

 Watchdog profile configuration is done in the following manner:

Set: EZapiStat_ConfigCmd_SetWatchdogProfile
Get: EZapiStat_StatCmd_GetWatchdogProfile
Struct: EZapiStat_WatchdogProfile

8.1.2.12 Watchdog Counters

After defining a statistics watchdog counter group, the watchdog counters within the group can be configured separately. Watchdog counters are configured with an initial value and an associated watchdog profile that defines thresholds (see “[Watchdog Profiles](#)” for more details).

 Watchdog counter configuration is done in the following manner:

Set: EZapiStat_ConfigCmd_SetWatchdogCounters
Set: EZapiStat_ConfigCmd_StartWatchdogCounters
Get: EZapiStat_StatCmd_GetWatchdogCounters
Struct: EZapiStat_WatchdogCounterConfig

- ▶ *The EZapiStat_WatchdogCounterConfig structure can be used for operation with a watchdog counter, with a sequential range of watchdog counters or with a number of non-sequential watchdog counters.*
- ▶ *The counters specified in the EZapiStat_WatchdogCounterConfig structure must be a subset of a standard group that was previously defined.*
- ▶ *The watchdog counter profile associated with a watchdog counter must be configured before configuring the counter.*
- ▶ *By default watchdog counters are associated with watchdog profile #0.*

8.1.2.13 On-demand General Purpose Machines

The NPS has 16 general purpose machines per unit that scan counters for refreshing and garbage collection, which may result in issuing messages.

The refresh feature periodically updates token buckets that are not accessed by the CTOPs.

 On-demand general purpose machine configuration is done in the following manner:

Set: EZapiStat_ConfigCmd_SetGPMachineParams
 Get: EZapiStat_StatCmd_GetGPMachineParams
 Struct: EZapiStat_GPMachineParams

Garbage Collection

The machines periodically scan long or double counters.

When the machines scan a counter, they reset the counter value, and generate a message to a message queue indicating the counter value before the reset. This can be used to periodically export the statistic counter data to the control plane or to the CTOPs (via the message DMAs).

The machines perform a configurable amount of intermediate cycles between each main cycle. In order to reduce the amount of updates to the control plane, the scan machines only reset and reports counter with values above a configurable set of thresholds (one threshold for the main cycles and another threshold for the intermediate cycles). This can be used to implement a scheme where the reported counter data is correct up to a number of bytes/events within every intermediate cycle period and is more accurate or even fully accurate within every main cycle period.

Furthermore, in order to distribute the messages over time, the implementation interleaves the intermediate and main cycles across counters. Thus, for example, if the main cycle is performed once in every 4 cycles (i.e. there are 3 intermediate cycles between each main cycle), the hardware implementation would send messages on any non-zero counter on counters 0, 4, 8, etc. on the first scan, on counters 1, 5, 9, etc. on the second scan, on counters 2, 6, 10, etc. on the third scan and on counters 3, 7, 11, etc. on the fourth scan.

Refresh

The machines periodically update Token Bucket counters (of all the various types) that are not accessed by the CTOPs.


The following configurations exist for each machine:

- Enable/disable – Determines if the machine is activated.
 - Start counter/number of counters – Determines the counters scanned by this machine.
 - Min/max scan interval – Configures the rate at which the machine scans the counters. Both a minimum and maximum time period between each 2 counters can be configured.
 - Intermediate cycles – Configures the number of intermediate cycles between main cycles. A value of 0 indicates that there are no intermediate cycles (every scan is a main cycle).
 - Reporting Thresholds – Configures the thresholds for reporting events for the main and intermediate cycles.
 - Enable Token Bucket – Enables the scan of Token Bucket counters; affects the scan interval.
- ▶ *General purpose machines can scan more than one group, as long as the groups' counters are logically contiguous.*
- ▶ *Token Bucket enabled machines should be configured to scan all Token Bucket counters. Using Token Bucket counters (of all the various types) that are not refreshed by any machine may cause inaccuracies when Token Bucket counters are not accessed from the CTOPs for a time period.*

- ▶ *Except for machine enable, all of the machine's configurations must be set during Created state. Only machine enable/disable can be set afterwards.*
- ▶ *General purpose machines are only enabled during channel Initialize.*

8.1.2.14 On-demand General Counter Operations


The Statistics Group offers commands to set a range of general counters and to retrieve a range of statistics counters including their counter types and values. This command can be used with any on-demand type of statistics counter making it useful when the counter type is unknown or there are several different types within the range.

 General counter set/get is done in the following manner:

Set: EZapiStat_ConfigCmd_SetGeneralCounters
Get: EZapiStat_StatCmd_GetGeneralCounters
Struct: EZapiStat_GeneralCounterConfig

Overlap:


A group's colors cache in the statistics internal SRAM may be shared with other optimized groups in the partition. The shared area size will match the largest group. Only two to four groups from each unit can have *bOptOverlap* enabled.

 Freeing of the internal SRAM lines that match the input counters is done in the following manner:

Set: EZapiStat_ConfigCmd_FreeOptimizedOverlapCounters
Struct: EZapiStat_GeneralCounterConfig

8.1.2.15 On-demand Messaging Status

The Statistics Group offers a command to get the status of the on-demand statistics message queue. Reading of messages is done through the Data Path APIs.

 On-demand messaging status is read in the following manner:

Get: EZapiStat_StatCmd_GetMessageQueueStatus
Struct: EZapiStat_MessageQueueStatus

8.1.3 Posted Counters

Posted counters are most useful for applications where counters are incremented or decremented frequently but whose values are rarely needed. There is no direct read or get operation for posted counters, rather their values can be received as messages. There are four different types of posted messages generated by posted commands, the counter overflow threshold or the posted garbage collection machines.

All posted counters reside in the external DDR memory (EMEM).

There are two posted counter spaces – optimized and standard. Operations on counters in the optimized space are executed with lower latency since operations on counters in the standard space must pass through two levels of command FIFOs.

Unlike the on-demand counters, there are no posted counter types and all posted counters are 64 bits long. Posted counter operations may be performed on either a single 64b counter or on two 64b successive posted counters.

All posted counter configuration commands in the Statistics Group contain the word “Posted” in the command name to distinguish them from the on-demand configuration commands.

8.1.3.1 Posted Partition Parameters

The posted partition is a logical entity which represents a posted counter space. There can be up to 4 posted partitions. Each partition can be spread across one or more posted units.

The following parameters are defined:

- Partition – Indicates which partition will be configured.
- Enable – Enabling partition.
- Application Mask – NPS can run multiple applications where each cluster may run a different application. The application mask indicates to which applications the partition is relevant.
- Unit Mask – Mask of posted hardware units
 - Bits 0-1 represent south, 2-3 represent north.
 - A posted unit can only be part of a single partition.
 - Number of units which can share a partition can be 1, 2, or 4.
- MSID – The memory space ID used by the CTOPs to access the counter range represented by the partition.



Statistic posted partition parameter configuration is done in the following manner:

Set: EZapiStat_ConfigCmd_SetPostedPartitionParams
 Get: EZapiStat_StatCmd_GetPostedPartitionParams
 Struct: EZapiStat_PostedPartitionParams

- ▶ *The statistic partition parameter configuration must be performed while the channel is in the created state – before the call to **EZapiChannel_Initialize()**.*

8.1.3.2 Posted Groups

Up to two posted counter groups are supported.

For each of posted counter group, the following parameters are defined:

- Partition – Indicates which partition the shadow range is in.
- Group Type – Indicates whether the group represents an optimized or standard posted range.
- Start counter – The first counter in the group.
- Num counters – The number of counters in the group.



Posted statistics group configuration is done in the following manner:

Set: EZapiStat_ConfigCmd_SetPostedGroupParams
 Get: EZapiStat_StatCmd_GetPostedGroupParams
 Struct: EZapiStat_PostedGroupParams

- ▶ *The statistic posted group parameter configuration must be performed while the channel is in the created state – before the call to **EZapiChannel_Initialize()**.*

8.1.3.3 Posted Shadow Group Parameters

The posted shadow group allows user to remap a range of posted addresses within a partition to another range in the partition. Remapping is bidirectional. Each posted partition contains up to two shadow remapping ranges, one for the optimized and one for the standard posted counters.

The following parameters are defined:

- Partition – Indicates which partition the shadow range is in.
- Group Type – Indicates whether the shadow range is within the optimized or standard posted-group/counter-range.
- Enable – Enabling shadow remapping.
- Start Counter – First destination counter of shadow remapping.
- Num Counters – Number of remapped counters
- Remap Counter – First source counter of shadow remapping.



Posted shadow group configuration is done in the following manner:

Set: EZapiStat_ConfigCmd_SetPostedShadowGroupParams
 Get: EZapiStat_StatCmd_GetPostedShadowGroupParams
 Struct: EZapiStat_PostedShadowGroupParams

8.1.3.4 Posted Overflow Profile

Up to 7 posted overflow profiles are supported. Profiles dictate the thresholds and other overflow related configurations for the counter range that they cover. Counters not specifically covered by an enabled posted overflow profile will not generate overflow messages. Counters within range of more than one profile, will be affected by the profile with the higher index (Profile).

For each of the posted overflow profiles, the following parameters are defined:

- Partition – Indicates which posted partition for the overflow profile.
- Profile – Profile to configure.
- Enable – Dictates if counters covered by this profile will generate messages.
- Start counter – The first counter affected by profile.
- Num counters – The number of counters in the affected by profile.
- Signed – Whether the counters covered by the profile may have negative values.

- Range Type – Profile will affect only even indexed counters, odd indexed counters, or all counters within the range.
- Low and high thresholds – Configure the counter values that trigger an overflow message.



Posted statistics group configuration is done in the following manner:

Set: EZapiStat_ConfigCmd_SetPostedOverflowProfile
 Get: EZapiStat_StatCmd_GetPostedOverflowProfile
 Struct: EZapiStat_PostedOverflowProfile

8.1.3.5 Posted GC Profile

Up to 7 posted garbage collection (GC) profiles are supported. Profiles dictate the thresholds and other GC related configurations for the counter range that they cover. Counters not specifically covered by an enabled posted GC profile will not generate garbage collection messages. Counters within range of more than one profile, will be affected by the profile with the higher index (Profile).

For each of posted GC profiles, the following parameters are defined:

- Partition – Indicates which posted partition for the GC profile.
- Profile – Profile to configure.
- Enable – Dictates if counters covered by this profile will generate messages.
- Start counter – The first counter affected by profile.
- Num counters – The number of counters in the affected by profile.
- Signed – Whether the counters covered by the profile may have negative values.
- Range Type – Profile will affect only even indexed counters, odd indexed counters, or all counters within the range.
- Low and high thresholds – Configure the counter values that trigger a GC message.
- Low and high negative thresholds – Configure the negative counter values that trigger a GC message.



Posted statistics garbage collection profile configuration is done in the following manner:

Set: EZapiStat_ConfigCmd_SetPostedGCProfile
 Get: EZapiStat_StatCmd_GetPostedGCProfile
 Struct: EZapiStat_PostedGCProfile

8.1.3.6 Posted Counter Operations

The statistics API group provides commands to set posted counters, increment/decrement posted counters, as well as to report/recycle posted counters to issue a message with the current value of posted counters.

Report and recycle commands return the counter values through the message queue. The array of posted counters is used only if the function is not range enabled. In this case, messages will be sent on the counters indices that are saved in the array. In case of a range function, messages will be sent on a range of counters, according to parameters *uiStartCounter*, *uiNumCounters*, *uiRangeStep* and *uiSubRange* in *EZapiStat_PostedCounterConfig*.



Posted counter operations are done in the following manner:

Set: EZapiStat_ConfigCmd_SetPostedCounters
 EZapiStat_ConfigCmd_IncrementPostedCounters
 EZapiStat_ConfigCmd_DecrementPostedCounters
 EZapiStat_ConfigCmd_ReportPostedCounters
 EZapiStat_ConfigCmd_RecyclePostedCounters
 EZapiStat_ConfigCmd_ClearPostedCounters

Get: EZapiStat_StatCmd_GetPostedCounters
Struct: EZapiStat_PostedCounterConfig

- ▶ *The command EZapiStat_StatCmd_GetPostedCounters reads the posted counter value directly from memory, and thus it is not atomic. The field bValid that is returned in the counter structure indicates whether the read counter value is coherent. Operation order is not guaranteed.*



A check if all requested posted commands have been executed can be done in the following manner:

Get: EZapiStat_StatCmd_GetPostedStatus
Struct: EZapiStat_PostedStatus

8.1.3.7 Posted Messaging Status

The Statistics Group offers a command to get the status of the posted statistics message queue. Reading of messages is done through the Data Path APIs.



Posted messaging status is read in the following manner:

Get: EZapiStat_StatCmd_GetPostedMessageQueueStatus
Struct: EZapiStat_MessageQueueStatus

8.2 Summary of Commands – Stat Group

Table 8-2. Statistics Group routines

| API Routines |
|---------------------|
| EZapiStat_Config() |
| EZapiStat_Status() |

Table 8-3. Summary of Statistics Group commands

This table provides a summary of the commands in the Statistics Group, as well as the channel state in which each command may be performed.

| | Undefined | Created | Powered up | Initialized | Finalized | Running |
|---|-----------|---------|------------|-------------|-----------|---------|
| TOPIC & COMMAND | U | C | P | I | F | R |
| On-demand Counter Configuration | | | | | | |
| Configure global parameters | | | | | | |
| Set: EZapiStat_ConfigCmd_SetPartitionParams | - | ✓ | - | - | - | - |
| Get: EZapiStat_StatCmd_GetPartitionParams | - | ✓ | ✓ | ✓ | ✓ | ✓ |
| Configure on-demand statistics groups | | | | | | |
| Set: EZapiStat_ConfigCmd_SetGroupParams | - | ✓ | - | - | - | - |
| Get: EZapiStat_StatCmd_GetGroupParams | - | ✓ | ✓ | ✓ | ✓ | ✓ |
| Set: EZapiStat_ConfigCmd_SetShadowGroupParams | - | - | - | ✓ | ✓ | ✓ |
| Get: EZapiStat_StatCmd_GetShadowGroupParams | - | - | - | ✓ | ✓ | ✓ |
| Long counter configuration and operation | | | | | | |
| Set: EZapiStat_ConfigCmd_SetLongCounters | - | - | - | ✓ | ✓ | ✓ |
| Get: EZapiStat_StatCmd_GetLongCounters | - | - | - | ✓ | ✓ | ✓ |
| Set: EZapiStat_ConfigCmd_SetLongCounterValues | - | - | - | ✓ | ✓ | ✓ |
| Get: EZapiStat_StatCmd_GetLongCounterValues | - | - | - | ✓ | ✓ | ✓ |
| Set: EZapiStat_ConfigCmd_IncrementLongCounters | - | - | - | ✓ | ✓ | ✓ |
| Set: EZapiStat_ConfigCmd_DecrementLongCounters | - | - | - | ✓ | ✓ | ✓ |
| Set: EZapiStat_ConfigCmd_ResetLongCounters | - | - | - | ✓ | ✓ | ✓ |
| Set: EZapiStat_ConfigCmd_CondDecrementLongCounters | - | - | - | ✓ | ✓ | ✓ |
| Double counter configuration | | | | | | |
| Set: EZapiStat_ConfigCmd_SetDoubleCounters | - | - | - | ✓ | ✓ | ✓ |
| Get: EZapiStat_StatCmd_GetDoubleCounters | - | - | - | ✓ | ✓ | ✓ |
| Set: EZapiStat_ConfigCmd_IncrementDoubleCounters | - | - | - | ✓ | ✓ | ✓ |
| Set: EZapiStat_ConfigCmd_DecrementDoubleCounters | - | - | - | ✓ | ✓ | ✓ |
| Set: EZapiStat_ConfigCmd_ResetDoubleCounters | - | - | - | ✓ | ✓ | ✓ |
| Bitwise counter configuration | | | | | | |
| Set: EZapiStat_ConfigCmd_SetBitwiseCounters | - | - | - | ✓ | ✓ | ✓ |
| Get: EZapiStat_StatCmd_GetBitwiseCounters | - | - | - | ✓ | ✓ | ✓ |
| Set: EZapiStat_ConfigCmd_SetBitsBitwiseCounters | - | - | - | ✓ | ✓ | ✓ |
| Get: EZapiStat_StatCmd_GetBitsBitwiseCounters | - | - | - | ✓ | ✓ | ✓ |
| Set: EZapiStat_ConfigCmd_ClearBitsBitwiseCounters | - | - | - | ✓ | ✓ | ✓ |
| Set: EZapiStat_ConfigCmd_WriteBitsBitwiseCounters | - | - | - | ✓ | ✓ | ✓ |
| Set: EZapiStat_ConfigCmd_IncrementBitsBitwiseCounters | - | - | - | ✓ | ✓ | ✓ |

| TOPIC & COMMAND | | U | C | P | I | F | R |
|--|---|---|---|---|---|---|---|
| Set: | EZapiStat_ConfigCmd_DecrementBitsBitwiseCounters | - | - | - | ✓ | ✓ | ✓ |
| Set: | EZapiStat_ConfigCmd_CondSetBitsBitwiseCounters | - | - | - | ✓ | ✓ | ✓ |
| Token Bucket profile configuration | | | | | | | |
| Set: | EZapiStat_ConfigCmd_SetTokenBucketProfile | - | - | - | ✓ | ✓ | ✓ |
| Get: | EZapiStat_StatCmd_GetTokenBucketProfile | - | - | - | ✓ | ✓ | ✓ |
| Token Bucket configuration | | | | | | | |
| Set: | EZapiStat_ConfigCmd_SetTokenBucketCounters | - | - | - | ✓ | ✓ | ✓ |
| Get: | EZapiStat_StatCmd_GetTokenBucketCounters | - | - | - | ✓ | ✓ | ✓ |
| Get: | EZapiStat_StatCmd_GetTokenBucketColor | - | - | - | ✓ | ✓ | ✓ |
| Get: | EZapiStat_StatCmd_CheckTokenBucketColor | - | - | - | ✓ | ✓ | ✓ |
| Hierarchical Token Bucket configuration | | | | | | | |
| Set: | EZapiStat_ConfigCmd_SetHierarchicalTokenBucketCounters | - | - | - | ✓ | ✓ | ✓ |
| Get: | EZapiStat_StatCmd_GetHierarchicalTokenBucketCounters | - | - | - | ✓ | ✓ | ✓ |
| Set: | EZapiStat_ConfigCmd_UpdateHierarchicalTokenBucketCounters | - | - | - | ✓ | ✓ | ✓ |
| Get: | EZapiStat_StatCmd_GetHierarchicalTokenBucketColor | - | - | - | ✓ | ✓ | ✓ |
| Watchdog profile configuration | | | | | | | |
| Set: | EZapiStat_ConfigCmd_SetWatchdogProfile | - | ✓ | - | ✓ | ✓ | ✓ |
| Get: | EZapiStat_StatCmd_GetWatchdogProfile | - | ✓ | - | ✓ | ✓ | ✓ |
| Watchdog counter configuration | | | | | | | |
| Set: | EZapiStat_ConfigCmd_SetWatchdogCounters | - | - | - | ✓ | ✓ | ✓ |
| Set: | EZapiStat_ConfigCmd_StartWatchdogCounters | - | - | - | ✓ | ✓ | ✓ |
| Get: | EZapiStat_StatCmd_GetWatchdogCounters | - | - | - | ✓ | ✓ | ✓ |
| On-demand general-purpose machine configurations | | | | | | | |
| Set: | EZapiStat_ConfigCmd_SetGPMachineParams | - | ✓ | ✓ | ✓ | ✓ | ✓ |
| Get: | EZapiStat_StatCmd_GetGPMachineParams | - | ✓ | ✓ | ✓ | ✓ | ✓ |
| General counter operations | | | | | | | |
| Set: | EZapiStat_ConfigCmd_SetGeneralCounters | - | - | - | ✓ | ✓ | ✓ |
| Get: | EZapiStat_StatCmd_GetGeneralCounters | - | - | - | ✓ | ✓ | ✓ |
| Set: | EZapiStat_ConfigCmd_FreeOptimizedOverlapCounters | - | - | - | ✓ | ✓ | ✓ |
| Get: | EZapiStat_StatCmd_GetGeneralCounters | - | - | - | ✓ | ✓ | ✓ |
| On-demand counter messaging status | | | | | | | |
| Get: | EZapiStat_StatCmd_GetMessageQueueStatus | - | - | - | ✓ | ✓ | ✓ |

(continued)

| TOPIC & COMMAND | U | C | P | I | F | R |
|---|---|---|---|---|---|---|
| Posted Counter Configuration | | | | | | |
| Posted counter configurations and operations | | | | | | |
| Set: EZapiStat_ConfigCmd_SetPostedPartitionParams | - | ✓ | ✓ | ✓ | ✓ | ✓ |
| Get: EZapiStat_StatCmd_GetPostedPartitionParams | - | ✓ | ✓ | ✓ | ✓ | ✓ |
| Set: EZapiStat_ConfigCmd_SetPostedGroupParams | - | ✓ | - | - | - | - |
| Set: EZapiStat_ConfigCmd_SetPostedCounters | - | - | - | ✓ | ✓ | ✓ |
| Set: EZapiStat_ConfigCmd_IncrementPostedCounters | - | - | - | ✓ | ✓ | ✓ |
| Set: EZapiStat_ConfigCmd_DecrementPostedCounters | - | - | - | ✓ | ✓ | ✓ |
| Set: EZapiStat_ConfigCmd_ReportPostedCounters | - | - | - | ✓ | ✓ | ✓ |
| Set: EZapiStat_ConfigCmd_RecyclePostedCounters | - | - | - | ✓ | ✓ | ✓ |
| Set: EZapiStat_ConfigCmd_ClearPostedCounters | - | - | - | ✓ | ✓ | ✓ |
| Set: EZapiStat_ConfigCmd_SetPostedShadowGroupParams | - | - | - | - | ✓ | ✓ |
| Set: EZapiStat_ConfigCmd_SetPostedOverflowProfile | - | ✓ | - | ✓ | ✓ | ✓ |
| Set: EZapiStat_ConfigCmd_SetPostedGCProfile | - | - | - | ✓ | ✓ | ✓ |
| Get: EZapiStat_StatCmd_GetPostedCounters | - | - | - | ✓ | ✓ | ✓ |
| Get: EZapiStat_StatCmd_GetPostedGroupParams | - | ✓ | ✓ | ✓ | ✓ | ✓ |
| Get: EZapiStat_StatCmd_GetPostedShadowGroupParams | - | - | - | - | ✓ | ✓ |
| Get: EZapiStat_StatCmd_GetPostedOverflowProfile | - | ✓ | - | ✓ | ✓ | ✓ |
| Get: EZapiStat_StatCmd_GetPostedGCProfile | - | - | - | ✓ | ✓ | ✓ |
| Get: EZapiStat_StatCmd_GetPostedStatus | - | - | - | ✓ | ✓ | ✓ |
| Posted counter messaging status | | | | | | |
| Get: EZapiStat_StatCmd_GetPostedMessageQueueStatus | - | - | - | ✓ | ✓ | ✓ |

9. TCAM Group

The TCAM Group contains API routines for configuration and management of the network processor's internal and external TCAMs.

9.1 Overview

The TCAM Group contains configurations for the chip's two embedded TCAMs ("internal TCAMs") and also for the network processor's linked external TCAMs ("external TCAMs").

9.1.1 TCAM States

The TCAM Group is an extension of the Channel Group, and thus does not have states of its own, but rather shares the Channel Group states (see "[Channel States](#)" section).

TCAM parameters affecting external NPS memory allocation must be defined when the channel is in the created state. When the channel is initialized (by calling **EZapiChannel_Initialize()**), the requested memory sizes are allocated.

The internal and external TCAM configurations should be done in the channel initialized state. The configured parameters are aggregated in an internal software cache and are only written to the network processor in the **EZapiChannel_Finalize()** operation.

The writing of internal and external entries can be done in the initialized, finalized or running channel state.

9.1.2 Configuration of the Internal TCAM

Internal TCAM Lookup Table:

The configuration of the internal TCAM is done by defining logical tables ("Internal TCAM Lookup Tables") and their physical locations in the network processor's internal TCAMs. Each table resides in one of the two internal TCAMs; one TCAM on each side of the NPS. Up to 64 logical tables can be defined in parallel on each side of the network processor.

Other than the chip side, the user also specifies the physical parameters of the designated logical table: key size, start bank, number of banks to use, start row and number of rows to use.

The supported key sizes are 10, 20, 30, 40, 50, 60, 70 and 80 bytes.

There are also two special modes for the internal TCAM lookup table: associated data mode and algorithmic TCAM mode.

In Associated Data mode, the developer can attach extra data to each TCAM entry. In case of a hit in the lookup sequence, the attached data will be returned as a part of the lookup result (instead of the hit index, which is returned when the Associated Data feature is disabled).

When declaring an internal TCAM lookup table which uses associated data, the user should also specify the data size which is associated with every result and also the base line in the Associated Data table, a database which holds all the associated data. The developer should separate the associated data lines of each lookup table by declaring tables with non-overlapping associated databases.

In algorithmic TCAM mode, the developer can specify an internal TCAM lookup table as a part of the algorithmic TCAM feature. This should go hand to hand with declaring an Algorithmic TCAM search structure (see the "[Algorithmic TCAM](#)" section in the *Search Structures Group* chapter).

Internal TCAM Lookup Profile:

In addition to defining the logical internal TCAM lookup tables, the developer should also declare an internal TCAM lookup profile which defines a lookup process. This is a set of logical tables which are looked-up in parallel. Up to 4 internal TCAM lookup tables can be looked-up in parallel. The definition of an internal TCAM lookup profile consists of specification of the IDs of the internal lookup tables which are looked up in parallel.

When defining internal TCAM lookup tables which physically overlap, the developer should configure the SequentialLookup parameter as True. In addition, when defining internal TCAM lookup tables which overlap in groups of 256 rows, the developer should also configure the SequentialLookup parameter as True. For example, if one database consists of rows 0-31, no other database can use rows 0-255 and be searched in parallel.

In addition, only the first internal lookup table can be of either associated data type or algorithmic TCAM type.

Internal TCAM Entries:

When adding internal TCAM entries, the developer should specify the exact logical table to which the relevant entry is added. This includes specifying the network processor's side and the ID of the relevant internal TCAM lookup table.

In addition, the developer must specify the relevant index of the entry. When running an internal TCAM lookup on a specific table, the lowest index which results in a hit according to the given key is returned.

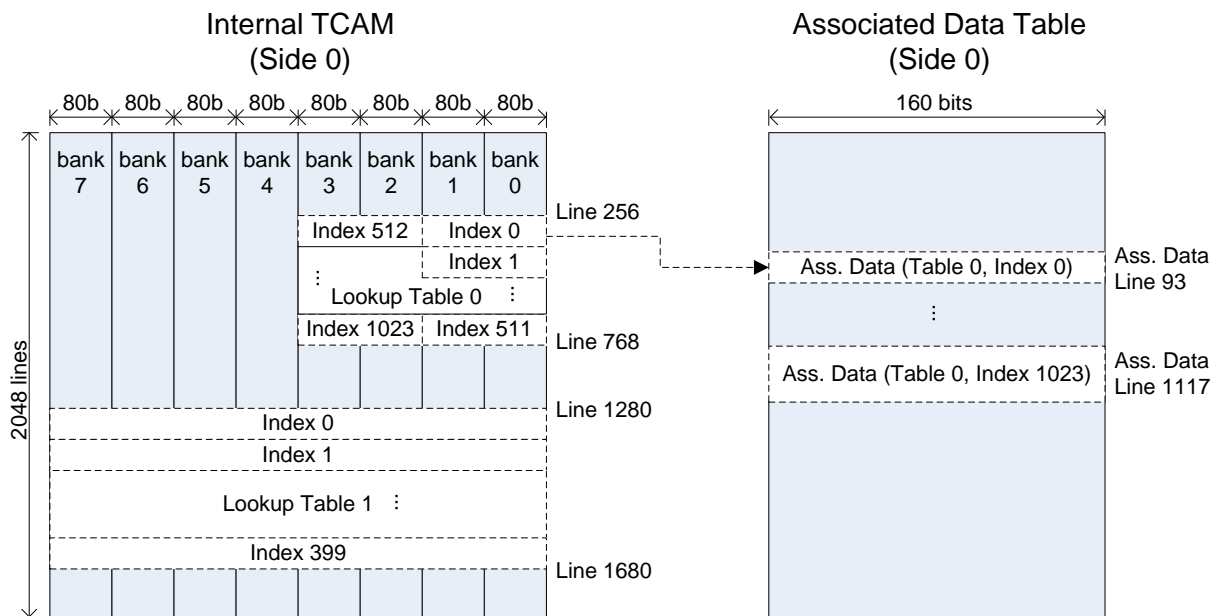
When writing an entry to an internal TCAM lookup table which is attributed as holding associated data, the developer should also specify the relevant associated data string.

Configuration Example:

The below above depicts a configuration of two internal TCAM lookup tables in the internal TCAM unit in Side 0 of the NPS:

- Lookup Table 0:
key size 20 bytes, start row 256, num rows 512, num of banks 4, associated database 93.
- Lookup Table 1:
key size 80 bytes, start row 1280, num rows 400, num of banks 8, no associated data.

Figure 9-1. Example configuration



An internal TCAM lookup profile comprising of the above two internal TCAM lookup tables can be declared, enabling a parallel search in both tables.

In case of several tables that hold associated data or Algorithmic TCAM data, it is the user's responsibility to set the Associated Data base addresses of the tables so that data will not be overrun.

9.1.3 Configuration of the External TCAM

External TCAM Lookup Table:

The configuration of the external TCAM is done by defining logical tables ("External TCAM Lookup Tables"). Up to 128 logical tables can be declared in parallel. Each table is connected through one of the two network processor's sides.

Each external TCAM lookup table is associated with a specific NetLogic LTR.

External TCAM Lookup Profile:

In addition to defining the logical external TCAM lookup table, the developer should also define an external TCAM lookup profile. Each external TCAM lookup profile consists of the relevant external TCAM lookup table which is associated with the lookup. Up to 64 external TCAM lookup profiles can be defined.

- ▶ *The external TCAM controller commands only configure the NPS to work with an external TCAM device. This does not configure the external TCAM device itself. In order to configure the actual TCAM device, the specific TCAM device vendor's SDK should be used.*
- ▶ *An internal TCAM lookup profile and external TCAM lookup profile with the same profile ID cannot be configured together.*

9.2 Summary of Commands – TCAM Group

Table 9-1. TCAM Group routines

| API Routines |
|---------------------|
| EZapiTCAM_Config() |
| EZapiTCAM_Status() |

Table 9-2. Summary of TCAM Group commands

This table provides a summary of the commands in the TCAM Group, as well as the channel state in which each command may be performed.

| | | Undefined | Created | Powered up | Initialized | Finalized | Running |
|--------------------------------------|--|-----------|---------|------------|-------------|-----------|---------|
| TOPIC & COMMAND | | U | C | P | I | F | R |
| TCAM controller configuration | | | | | | | |
| Set: | EZapiTCAM_ConfigCmd_SetExtTCAMLookupTable | - | - | - | ✓ | - | - |
| Set: | EZapiTCAM_ConfigCmd_SetExtTCAMLookupProfile | - | - | - | ✓ | - | - |
| Set: | EZapiTCAM_ConfigCmd_ExtTCAMCommand | - | - | - | ✓ | ✓ | ✓ |
| Set: | EZapiTCAM_ConfigCmd_SetExtTCAMInterruptControl | - | - | - | ✓ | ✓ | ✓ |
| Set: | EZapiTCAM_ConfigCmd_SetIntTCAMLookupTable | - | - | - | ✓ | - | - |
| Set: | EZapiTCAM_ConfigCmd_SetIntTCAMLookupProfile | - | - | - | ✓ | - | - |
| Set: | EZapiTCAM_ConfigCmd_WriteIntTCAMData | - | - | - | ✓ | ✓ | ✓ |
| Set: | EZapiTCAM_ConfigCmd_SetIntTCAMInterruptControl | - | - | - | ✓ | ✓ | ✓ |
| Get: | EZapiTCAM_StatCmd_GetExtTCAMLookupTable | - | - | - | ✓ | ✓ | ✓ |
| Get: | EZapiTCAM_StatCmd_GetExtTCAMLookupProfile | - | - | - | ✓ | ✓ | ✓ |
| Get: | EZapiTCAM_StatCmd_GetExtTCAMInterruptStatus | - | - | - | ✓ | ✓ | ✓ |
| Get: | EZapiTCAM_StatCmd_GetIntTCAMLookupTable | - | - | - | ✓ | ✓ | ✓ |
| Get: | EZapiTCAM_StatCmd_GetIntTCAMLookupProfile | - | - | - | ✓ | ✓ | ✓ |
| Get: | EZapiTCAM_StatCmd_ReadIntTCAMData | - | - | - | ✓ | ✓ | ✓ |
| Get: | EZapiTCAM_StatCmd_GetIntTCAMInterruptStatus | - | - | - | ✓ | ✓ | ✓ |

10. Primitive Group

The Primitive Group contains routines for low-level services to access the NPS device's memories and registers.

10.1 Overview

The Primitive Group is supplied to allow convenient low-level access to NPS registers and memories.

The other API groups provide the high-level services required for most control plane applications. The services in the Primitive Group are generally to be used only where high-level APIs do not exist for the required functionality.

IMPORTANT: The routines in the Primitive Group should be used with extreme care, as they provide low-level accesses which may cause the NPS to malfunction if used improperly.

10.1.1 Register Access

The Primitive Group API provides routines for accessing NPS device registers:

- **EZapiPrm_WriteReg()** – Write to a register.
- **EZapiPrm_ReadReg()** – Read from a register. If the register is readable, the value is returned from the NPS device. Otherwise, the value is returned from the CP library's software cache based on the last value written.
- **EZapiPrm_RMWReg()** – Modify a register. The register is read (from the NPS device or from the CP library's software cache), modified based on the supplied value and mask, and written back to the NPS device.
- **EZapiPrm_GetRegInfo()** – Retrieve information on a register – such as its address and access permissions.

The Primitive Group API provides a routine for printing register information, **EZapiPrm_PrintInfo()**.

Command: EZapiPrm_PrintInfoCmd_PrintRegisters

Struct: EZapiPrm_RegisterInfoParams

All primitive register access routines use the EZapiRegId enum to indicate the register to use (refer to EZapiRegId.h). The various register details (such as addresses and field divisions) are defined in EZapiRegDef.h.

10.1.2 Memory Access

The Primitive Group API provides routines for accessing various NPS device memories (internal and external) and queues.

- **EZapiPrm_WriteSRAM()** – Write to an SRAM.
- **EZapiPrm_ReadSRAM()** – Read from an SRAM. If the SRAM is readable, the value is returned from the NPS device. Otherwise, the value is returned from the CP library's software cache based on the last value written. If the memory is not readable and not cached in the CP library, an error is returned.
- **EZapiPrm_RMWSRAM()** – Modify an SRAM. If the memory supports a modify operation in the NPS device hardware, the modify operation is executed based on the supplied value and mask. Otherwise, the memory is read (from the NPS device or from the CP library's software cache), modified based on the supplied value and mask, and written back to the NPS device. If the memory is not readable and not cached in the CP library, an error is returned.

- **EZapiPrm_EncodeSRAM()** – Encode an SRAM line. This is a service routine which constructs a memory line from a structure containing a series of numeric fields, based on the memory line definitions.
- **EZapiPrm_DecodeSRAM()** – Decode an SRAM line. This is a service routine which divides a memory line into a structure containing a series of numeric fields, based on the memory line definitions.
- **EZapiPrm_GetSRAMInfo()** – Retrieve information on an SRAM, such as its size, access permissions and field division.
 - ▶ *EZapiPrm_WriteSRAM(), EZapiPrm_ReadSRAM() and EZapiPrm_RMWSRAM() routines accept as an input a special parameter which is a pointer to a structure defined per memory ID if needed. This additional parameter allows supplying additional input fields for memories when standard input fields are insufficient to access the memory. The parameter should be NULL for all memories unless otherwise defined in the memory ID documentation.*

The Primitive Group API provides a routine for printing SRAM information, **EZapiPrm_PrintInfo()**.

Command: EZapiPrm_PrintInfoCmd_PrintSRAMS

Struct: EZapiPrm_SRAMInfoParams

All SRAM primitive memory access routines use the EZapiMemId enum to indicate the memory to use (refer to EZapiMemId.h). The various memory details (such as addresses and field divisions) are defined in EZapiMemDef.h.

- **EZapiPrm_ReadMem()** – Read an NPS memory or pop from a queue. If the memory is readable, the value is returned from the NPS. Otherwise, an error is returned.
- **EZapiPrm_WriteMem()** – Write an NPS memory or push to queue.

Memory routines handle memories that are defined in the enum EZapiPrm_MemId in EZapiPrm.h, usually these are memories that require special handling and additional specific input parameters in order to read from or write to them (external memory, for example).

10.1.3 Performance Module

The performance module provides flexible implementation and standard ways to configure NPS-400 blocks' performance metering.

The base performance module comprises two 32-bit accumulators. A selectable trigger event is provided to signal the performance event.

Besides selecting the signals or conditions to probe, the performance module configures the way an event is considered true, the action taken when the event is true, provides value filters to further qualify the event, and defines the way measurements are taken in the different accumulation methods.

The Primitive Group API provides the routines:

- **EZapiPrm_SetPerfModuleParams()** – allows configuring each of the performance models in the NPS. There may be up to 3 modules in cluster blocks and up to 4 modules in other blocks.
- **EZapiPrm_GetPerfModuleParams()** – allows getting a specific performance model.
- **EZapiPrm_GetPerfModuleResult()** – allows reading the result of a specific performance model.

10.1.4 Advanced Channel Features

The Primitive Group API allows setting and reading of CP library debug flags.

- **EZapiPrm_AdvConfig()** – Configures various advanced features of an NPS channel.
- **EZapiPrm_AdvStatus()** – Provides the status for NPS channel advanced features.

Struct: EZapiPrm_DebugFlagParams

10.2 Summary of Commands – PRM Group

Table 10-1. Primitive Group routines

This table provides a summary of the routines in the Primitive Group, as well as the channel state in which each command may be performed.

| | Undefined | Created | Powered up | Initialized | Finalized | Running |
|---|-----------|----------|------------|-------------|-----------|----------|
| TOPIC & COMMAND | U | C | P | I | F | R |
| Register Access Routines | | | | | | |
| EZapiPrm_WriteReg() | - | ✓ | ✓ | ✓ | ✓ | ✓ |
| EZapiPrm_ReadReg() | - | ✓ | ✓ | ✓ | ✓ | ✓ |
| EZapiPrm_RMWReg() | - | ✓ | ✓ | ✓ | ✓ | ✓ |
| EZapiPrm_GetRegInfo() | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| EZapiPrm_PrintInfo | - | - | ✓ | ✓ | ✓ | ✓ |
| Memory Access Routines | U | C | P | I | F | R |
| EZapiPrm_WriteSRAM() | - | - | ✓ | ✓ | ✓ | ✓ |
| EZapiPrm_ReadSRAM() | - | - | ✓ | ✓ | ✓ | ✓ |
| EZapiPrm_WriteMem() | - | - | ✓ | ✓ | ✓ | ✓ |
| EZapiPrm_ReadMem() | - | - | ✓ | ✓ | ✓ | ✓ |
| EZapiPrm_RMWSRAM() | - | - | ✓ | ✓ | ✓ | ✓ |
| EZapiPrm_EncodeSRAM() | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| EZapiPrm_DecodeSRAM() | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| EZapiPrm_GetSRAMInfo() | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| EZapiPrm_PrintInfo | - | - | ✓ | ✓ | ✓ | ✓ |
| Performance Module Commands | U | C | P | I | F | R |
| EZapiPrm_GetPerfModuleParams() | - | - | - | - | - | ✓ |
| EZapiPrm_GetPerfModuleResult() | - | - | - | - | - | ✓ |
| EZapiPrm_SetPerfModuleParams() | - | - | - | - | - | ✓ |
| Advanced Channel Features | U | C | P | I | F | R |
| EZapiPrm_AdvConfig() | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| EZapiPrm_AdvStatus() | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Debug | U | C | P | I | F | R |
| EZapiPrm_AdvConfigCmd_SetDebugFlag | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| EZapiPrm_AdvConfigCmd_SetLogPerBlockGeneralParams | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| EZapiPrm_AdvConfigCmd_SetLogPerBlockParams | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| EZapiPrm_AdvConfigCmd_SendStatisticCommand | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| EZapiPrm_AdvStatCmd_GetDebugFlag | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| EZapiPrm_AdvStatCmd_GetLogPerBlockGeneralParams | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| EZapiPrm_AdvStatCmd_GetLogPerBlockParams | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| EZapiPrm_PrintInfoCmd_PrintRegisters | - | - | ✓ | ✓ | ✓ | ✓ |
| EZapiPrm_PrintInfoCmd_PrintSRAMS | - | - | ✓ | ✓ | ✓ | ✓ |

11. Search Structures Group

The Search Structures Group defines and maintains each of the search data structures (lookup tables), such as routing and policy tables, which are used for lookups by the network processors (channels). These routines configure the search structure, its properties, its data entries (rules) and memory management. Each search structure is configured independently and may be used for lookups by any or all of the NPS devices/channels in the system.

Following are detailed explanations regarding several Search Structure Group API related topics.

11.1 Overview

EZchip's NPS devices utilize user-defined search structures to enable lookups in a combination of tables for implementing various forwarding, routing, classification, policy and state tables. Multiple search structures may be defined for storing routing entries, policy rules or other information as required by the application.

CTOPs perform lookups by matching the keys it receives to the various search structures' stored entries. The keys are built by extracting various fields from the frame and/or from previous lookup results.

The search structures are stored in either the NPS's internal memory, external memory, or a combination of both, depending upon the type of lookup structure and its configuration. Table sizes are restricted only by the available memory resources.

Each NPS can support different search data structures (lookup tables), which can be defined for storing routing entries, policy rules or other information. Several types of data structures (direct access table, hash table, UltraIP, etc.) are supported. For maximum flexibility, each structure may be configured independently, and may be used for various applications, such as forwarding and routing, flow classification, access control, etc. In addition, the entries (rules) for each structure are fully programmable at runtime.

The user application uses the EZcp's Search Structure API Group to define each of the search structure, including the structure's type (table, hash, etc.), key size, result size (i.e. associated data size), and number of entries (rules). Additional parameters specific to each structure type are also configurable. The EZcp configures search structure parameters, such as structure base addresses, in a designated place in the NPS external memory, from which the EZdp can read and initiate actions such as lookups.

The Search Structure Group provides commands to manipulate the entries (rules) for each structure. Entries, which include both the search keys and their associated results, may be added, updated or deleted. In addition, the API supports commands to query a specific entry or to iterate over all entries in a search structure. For table and hash structures, entries may also be added, updated or deleted from the data plane (EZdp). For all other structure types, entries may only be updated by EZcp.

The search structure data is stored in the NPS's internal memory and/or external memory. The allocation and management of memory for the structures is done by EZcp. Nevertheless, the Search Structure Group supports commands to allow the developer to control the memory management. Memory management parameters may be configured for each structure, allowing developers to control the way individual structures are managed.

In systems with more than one channel (NPS), each search structure may be flexibly assigned to one or more channels and shared for lookups by multiple NPS devices.

NPS supports a total of 128 different search structures.

This section provides an overview of the Search Structure API Group support. More detailed information specific to each structure type is provided in the sections that follow it.

11.1.1 Defining Search Structures


The **EZapiStruct_ConfigCmd_SetStructParams** command (EZapiStruct_Config() routine) is either used to define a new search structure (when given with the Enable parameter as set) or delete an existing search structure (when the Enable parameter is unset).

This command receives the EZapiStruct_StructParams structure, which defines the basic parameters for the search structure, such as:

- Structure ID is passed as a parameter to the routine.
- Structure type – table, hash, etc.
- Structure memory location – internal memory, external memory, or mixed.
- Key size – defines the size of the structure’s rule key data.
- Result size – defines the size of the structure’s rule associated data.
- Number of entries (rules) – defines the maximum number of rules in the structure.
- Partition – defined the search memory partition in which the structure is managed.
- Channel map – defines the channel or set of channels (NPS devices) to which the structure is assigned.

When used to delete a structure, this routine frees all resources used by the structure object. After this routine, the structure is in the undefined state.

Additional parameters specific to each structure type may be defined using the EZapiStruct_Config() routine and its associated commands. These parameters may, in some cases, also be changed at runtime. See the structure type specific sections for more information.

 See “[Search Memory Management](#)” for information on memory partitions and structure management.

- ▶ *The search structures must be defined before the search memory partition is loaded.*
- ▶ *The search structure’s parameters are cached in the EZcp, and are only written to the NPS when the partition is loaded.*
- ▶ *A structure cannot be deleted while it is loaded. The structure’s partition must first be unloaded.*

11.1.2 Search Memory Management

The developer can map each structure component to search memory space index using the EZapiStruct_ConfigCmd_SetTableMemMngParams, EZapiStruct_ConfigCmd_SetHashMemMngParams, etc. commands.

This enables the developer to control memory attributes such as internal/external memory location and duplications for each structure. The user must ensure that the memory space index which is used is initiated by the channel group. The used space index should have sufficient allocated memory to store all of the search structures that are mapped to it.

When using external space indexes, their Type parameter should be configured as SEARCH.

- ▶ *Only a table with a 16-byte result size can reside in an x0.5 or x1 internal memory space.*


11.1.3 Search Structure Entries (Rules)

The Search Structure Group provides commands to manipulate the entries (rules) for each structure. Entries may be added, updated or deleted. In addition, the API supports commands to query a specific entry or to iterate over all entries in a search structure.

An entry is composed of a key and an associated result.

For UltraIP and Algorithmic TCAM structures, a mask should also be specified, whereby each key and mask together define a set of one or more possible matching keys.

For the Algorithmic TCAM structure, entries are defined using an array of EZapiStructField structures. Each field is comprised of its own key and a mask, along with the field type. The Algorithmic TCAM entry must also include a Rank. The lookup returns the lowest rank to which a given key matches.

 The API includes several routines for performing entry manipulation operations on search structures.

```
Set:      EZapiStruct_AddEntry( )
          EZapiStruct_UpdateEntry( )
          EZapiStruct_DeleteEntry( )
          EZapiStruct_DeleteAllEntries( )
Struct:   EZapiEntry, EZapiStruct_EntryParams
```

► *Entries may only be defined after the structures are loaded.*

When adding or updating an entry, the operation can be specified as being performed in one of two modes that affect its outcome:

Enforced:

Enforced operations are used to ensure the requested outcome is achieved regardless of the current state of the entry within the search structure. If an add entry operation is performed in enforced mode, and the entry already exists, the operation is converted into a modify operation on the existing entry. In a similar manner, if a modify operation is performed in enforced mode, and the entry does not exist, the operation is converted into an add operation. In either case, an information return code is returned to indicate when an enforced entry operation was automatically converted.

Non-enforced:

Non-enforced operations always operate according to the current state of the entry within the search structure. If an add entry operation is performed in non-enforced mode, and the entry already exists, the operation fails. In a similar manner, if a modify operation is performed in non-enforced mode, and the entry does not exist, the operation fails. In either case, an error return code is returned to indicate that the operation failed.

Following is a table summarizing the behavior of each relevant entry operation in each mode, including the returned information or error code:


Table 11-1. Enforced vs. non-enforced operations when adding or updating an entry

| | ENFORCED | | NOT ENFORCED | |
|--------|--|--|---|--|
| | ENTRY EXISTS | ENTRY DOES NOT EXIST | ENTRY EXISTS | ENTRY DOES NOT EXIST |
| Add | Entry modified Info returned: EZrc_CP_SRH_ENTRY_ ALREADY_EXISTS_INF | Entry added EZok returned | Error returned: EZrc_CP_SRH_ENTRY_ ALREADY_EXISTS | Entry added EZok returned |
| Modify | Entry modified EZok returned | Entry added Info returned: EZrc_CP_SRH_ENTRY_ NOT_FOUND_INF | Entry modified EZok returned | Error returned: EZrc_CP_SRH_ENTRY_ NOT_FOUND |

| | ENFORCED | | NOT ENFORCED | |
|--------|--------------------------------|--|---------------------|-----------------------------|
| | ENTRY EXISTS | ENTRY DOES NOT EXIST | ENTRY EXISTS | ENTRY DOES NOT EXIST |
| Delete | Entry deleted EZok returned | No action Info returned: EZrc_CP_SRH_ENTRY_ NOT_FOUND_INF | -- | -- |

- ▶ *Delete entry operations always behave in enforced mode. Delete entry operations on entries that do not exist in the search structure return an information return code indicating the requested outcome was achieved, but that no entry was actually deleted from the structure.*

The API also includes routines to query the status of an entry, as well as to iterate over all entries on a structure:

 Retrieving an entry from a search structure:

Get: EZapiStruct_GetEntry()

Struct: EZapiEntry

 Iterating over all entries within a search structure:


Get: EZapiStruct_Iterate()

Struct: EZapiStruct_IteratorParams

 Perform a lookup operation on a structure using a key and return the entry matched:


Get: EZapiStruct_Lookup()

Struct: EZapiEntry

 Get the number of entries in a structure:

Get: EZapiStruct_StatCmd_GetNumEntries

Struct: EZapiStruct_NumEntries

 Get the memory usage status of a structure:

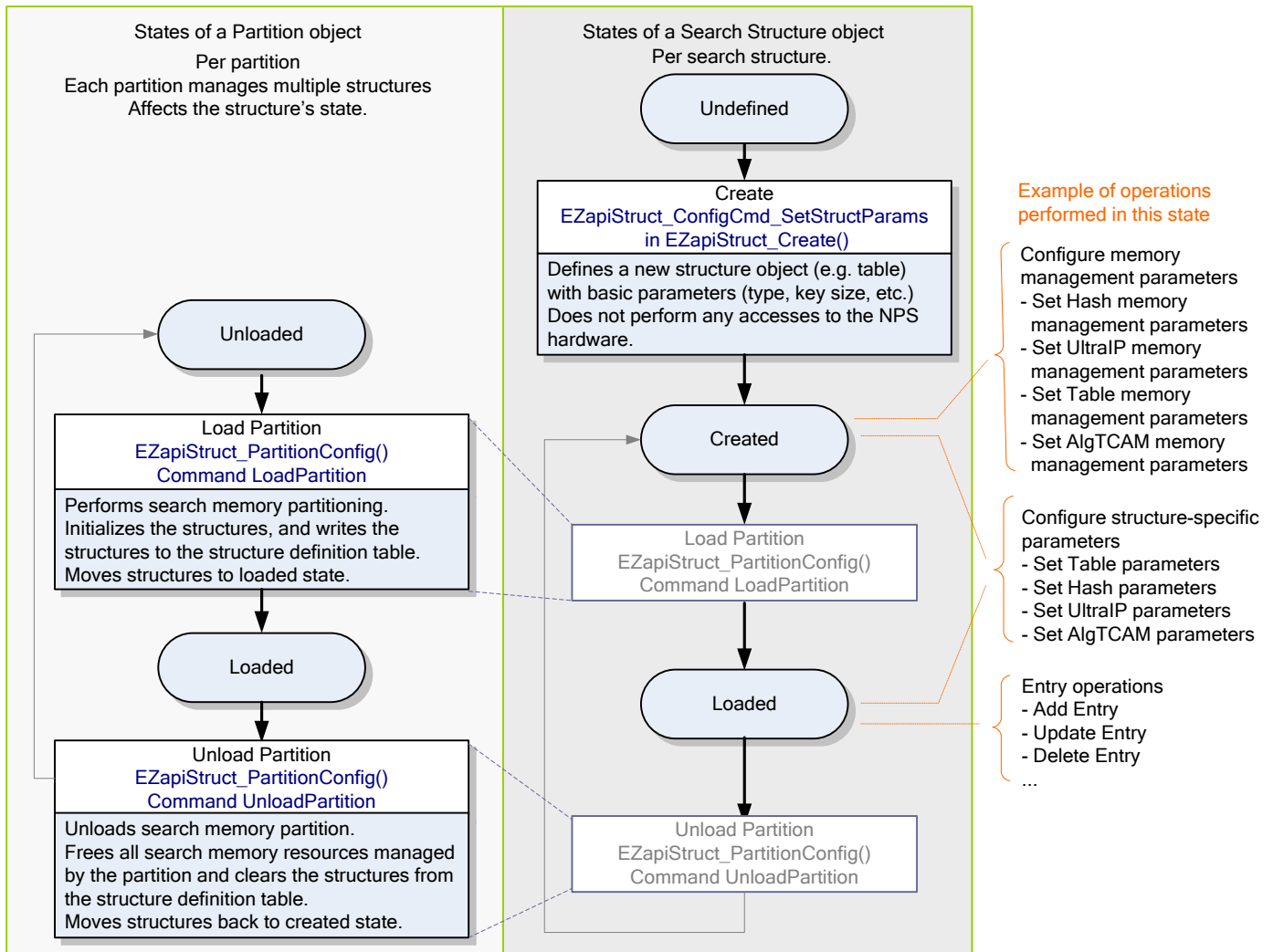
Get: EZapiStruct_StatCmd_GetMemUsage

Struct: EZapiStruct_MemUsage

11.1.4 Search Structure States

There is a search structure object representing each of the search structures in the system. The following diagram illustrates the search structure object's states.

Figure 11-1. Search Structure states



The lifetime of a search structure (and memory partitions) is comprised of the following main stages:

A search structure is initially in the undefined state.

The **EZapiStruct_ConfigCmd_SetStructParams** command defines a new search structure, including its basic parameters (structure type, key size, result size, etc.) as well as the partition it is managed in. This routine only defines the search structure software object, and does not perform any accesses to the NPS hardware device. After this routine, the structure is in the created state.

While in the **Created** state, the application may perform configuration commands, defining the parameters specific to the structure type. Some structure configurations must be performed before the structure is loaded and cannot be changed afterwards. In addition, the structure's memory management parameters can be configured while in the created state.

The **Load Partition** command in the **EZapiStruct_PartitionConfig()** routine loads the search memory partition, along with all structures managed by the partition. This routine performs the initial NPS memory allocations for the structures, initializes the structure in search memory (initially the structure is empty, e.g. has no rules) and then writes specific parameters for the structures to a designated space in the

external memory called the structure definition table (SDT). The SDT is used by EZdp to learn different attributes of each loaded search structure, such as table memory locations and entry sizes. These attributes allow EZdp to perform lookup operations and dynamic entry operations. After the Load Partition command, the structure is in the loaded state.

While in the **loaded** state, entry operations may be performed on the structure. In addition, some of the structure configurations may also be updated when in the loaded state.

- ▶ *Currently unloading a partition is not supported.*
- ▶ *Currently using more than one partition is not supported.*

11.2 Table Structures

Direct table structures allow direct-access lookups. Direct tables are implemented as a sequential array of results. The lookup key to a direct table is the index into the array. The result is the data in the indexed element/line.

Direct tables may reside in either the internal memory or external memory.

Direct tables structures support DP Update Mode. This is often used, for example, to maintain dynamic resources within the table entries' results.

11.2.1 Defining Table Structures

The following main parameters are defined for direct table structures:

General Parameters:

- Key size – The key size defines the number of bytes/bits used for specifying the line. Supported key sizes are 1, 2, 3 and 4 bytes.
- Result Size – The result size defines the size of each line. Supported result sizes are 16, 32, 64 and 128 bytes.
- Number of entries – The number of entries defines the maximum number of results in the array.
- Memory Area – The memory area to use is internal or external search memory.

Table-Specific Parameters:

- Multi Channel Data Mode – Specifies if the structure has identical data on all channels, or if the structure's data may differ between channels.
- Update Mode – Specifies if the structure is updated from the data plane (DP) or is only updated from the control plane processor (CP); see the “[Operation and Entries](#)” section.
- Cache Mode – Specifies if table data is cached in the CP library (see the “[Operation and Entries](#)” section).



Table parameters are set in the following manner:

```
Set:      EZapiStruct_ConfigCmd_SetTableParams
Get:      EZapiStruct_StatCmd_GetTableParams
Struct:   EZapiStruct_TableParams
```

- ▶ *The key size must be large enough to allow specifying values of between 0 and the maximum number of entries defined. For example, if the table contains 1024 entries, 10 bits are required to specify the table line/index, and the key size should thus normally be set to 2 bytes.*
- ▶ *Table structures defined as having DP update mode must have different multi channel data modes and Disabled Cache Mode.*

11.2.2 Memory Management

Direct tables may reside in either the internal memory or external memory.

Direct tables require a single continuous memory area. The size of the memory area used for a direct table is the number of entries/lines multiplied by the result size.

A direct table is mapped to a single memory space index. The space index that the table is mapped to must reside in the correct memory area (Internal for an internal structure or External for an external structure) and must have sufficient free space for it.



Table memory management parameters are set in the following manner:

```
Set:      EZapiStruct_ConfigCmd_SetTableMemMngParams
```

Get: EZapiStruct_StatCmd_GetTableMemMngParams
 Struct: EZapiStruct_TableMemMngParams

11.2.3 Operation and Entries

Update Mode:

Table structures can be configured to be updated from CP only or updated from DP only.

Cache Mode:

CP table structures can be configured to work with a cache in the CP library to enhance performance.

Three modes are supported:

- Full cache – EZcp maintains a full image of the table structure data. In this case, all read operations (entry state, lookup, iteration, number of entries) are returned from the cache, without needing to access the NPS.
 - In this case, EZcp can handle an ECC error even if the structure has only a single copy.
- Reduced cache – EZcp maintains a single validity bit for each table entry. In this case, the entry state and number of entries are always returned from the cache, without needing to access the NPS. Lookup and iteration operations will check the validity bit, and only access the NPS memory for valid entries.
- No cache – EZcp does not maintain any cache information. The number of entries is not maintained in EZcp, and is implemented by iterating over all table lines (which may be a lengthy operation).

The following table summarizes the affect of the update mode and cache mode configurations on table structure operations from the CP library.

Table 11-2. Affect of Update Mode and Cache Mode on Table Operations

| | ENTRY OPERATIONS (Add/Update/ Delete) | LOOKUP OPERATIONS | ITERATOR OPERATIONS | GET NUM ENTRIES OPERATIONS |
|----------------------|---|---|---|---|
| DP | NA | NA | NA | NA |
| CP, Full Cache | NPS memory access | CP cache | CP cache | CP counter |
| CP, Reduced Cache | NPS memory access | Validity from CP cache, NPS memory access if valid | Validity from CP cache, NPS memory access if valid | CP counter |
| CP, No Cache | NPS memory access | NPS memory access | NPS memory access | Iteration on all lines in NPS memory |

- ▶ *In future releases, the cache mode will also affect EZcp behavior on hot-swap operations. Table structures with full cache will be written to the NPS when a channel is added to an existing structure, while table structures with reduced or no cache will be reset to an empty state.*
- ▶ *In future releases, the cache mode will also affect EZcp behavior on warm-boot operations. For table structures with reduced or full cache, the cache must be restored as part of the warm boot operation. This can be done by replaying the entry data from the application (this requires the application to persist the table entry data), or by reading the data from the NPS (which may be a lengthy operation).*

11.3 Hash Structures

Hash structures perform lookups by calculating a hash functions on a given key. Entries are defined using any binary key. Hash table structures are usually used for tables with fixed length keys, e.g. DA, SA, 5-tuple IPv4/IPv6 sessions.

Hash tables may reside in the internal memory, external memory or in mixed memory (a combination of the two).

Hash structures support DP update mode. This is often used, for example, to maintain dynamic resources within the hash entries' results.

11.3.1 Defining Hash Structures

The following main parameters are defined for direct table structures:

General Parameters:

- Key size – The key size defines the number of bytes of the key data. Supported key sizes are 1-64 bytes.
- Result Size – The result size defines the size of each of the user results. Supported result sizes are 4-256 bytes in multiples of 4.
- Number of entries – The number of entries defines the maximum number of rules in the hash.
- Memory Area – The memory area to use can be internal memory, external memory or mixed (combination of the two).

Hash-Specific Parameters:

- Multi Channel Data Mode – Specifies if the structure has identical data on all channels, or if the structure's data may differ between channels.
- Update Mode – Specifies if the structure is updated from the CP or DP.
- Single Cycle – Specify if this is a single-cycle or two-cycle hash structure.
- Cache Mode – Static hash structures can be configured to work with a cache in the CP library to enhance performance.

 Hash parameters are set in the following manner:

| | |
|---------|-------------------------------------|
| Set: | EZapiStruct_ConfigCmd_SetHashParams |
| Get: | EZapiStruct_StatCmd_GetHashParams |
| Struct: | EZapiStruct_HashParams |

- ▶ *Hash structures defined as having DP Update Mode must have a Different multi channel data mode.*

11.3.2 Memory Management

Hash tables may reside in the internal memory, external memory or in a combination of the two.

The following parameters define the hash memory management:

Hash Size:

The hash size controls the number of bits used to index the hash structure's main table. For example, a hash size of 10 bits will generate a hash table of $2^{10}=1024$ entries. This parameter is used to control the tradeoff between memory required for the main table, and the likelihood of collisions in the hash structure.

Supported hash sizes are 8-15 bits for internal memory (or mixed), and 8-26 bits for external memory.

By default, EZcp will automatically calculate this to provide the best tradeoff.

Signature Page Memory Area:

The hash structure may reside in the internal memory, external memory or in a combination of the two. When residing in a combination of the two, the initial hash table resides in internal memory and the result pages reside in external memory. The additional signature pages may reside in either internal or external memory.

By default, the signature page memory area is in external memory.

Signature Page Percent:

The signature page percent controls the amount of memory allocated for additional signature pages. Additional signature pages are used when collisions occur on the hash table. The value specified is the percent out of the worse cases required to store the specified number of entries, assuming full collisions between all entries.

By default, EZcp will automatically calculate this to provide sufficient signature pages for all real-world cases, but still not require the worst case memory usage.

Result Page Percent:

The result page percent controls the amount of memory allocated for result pages in single cycle hashes. Result pages are used in single cycle hashes when collisions occur on the hash table. The value specified is the percent out of the worse cases required to store the specified number of entries, assuming full collisions between all entries.

By default, EZcp will automatically calculate this to provide sufficient result pages for all real-world cases, but still not require the worst case memory usage.

Signature Page Index Pool:


The index pool used for signature must be specified for hashes having DP update mode. Additional signature pages are used when collisions occur on the hash table. The allowed pool indexes are 1-64. The index pool used should also be defined as enabled for search by the EZapiChannel_IndexPoolParams command. Stating an index pool number in the Hash Memory Management Parameters will initiate the pool in the NPS hardware when the structure is loaded.

Result Page Index Pool:

The index pool used for result pages must be specified for hashes having DP update mode. Result pages are used to store the key and result of the rule. The allowed pool indexes are 1-64. The index pool used should also be defined as enabled for search by the EZapiChannel_IndexPoolParams command. Stating an index pool number in the Hash Memory Management Parameters will initiate the pool in the NPS hardware when the structure is loaded.

Main Table/Signature Pages/Result Pages Memory Indexes:

Defines the memory space indexes to which the hash components are mapped. Using separate space indexes for the different components allows the user to control different memory index parameters, such as replication number, per component.

 Hash memory management parameters are set in the following manner:

| | |
|---------|---|
| Set: | EZapiStruct_ConfigCmd_SetHashMemMngParams |
| Get: | EZapiStruct_StatCmd_GetHashMemMngParams |
| Struct: | EZapiStruct_HashMemMngParams |

11.3.3 Operation and Entries

Hash structures support both CP and DP update modes.

Update Mode:

Hash structures can be configured to be updated from CP only or DP only.

Cache Mode:

CP hash structures can be configured to work with a cache in the CP library to enhance performance. Two modes are supported:

- Full cache – EZcp maintains a full image of the hash structure entries. In this case, all read operations (entry state, lookup, iteration, number of entries) are returned from the cache without needing to access the NPS.
In this case EZcp can handle an ECC error even if the structure has only a single copy.
- No cache – EZcp does not maintain any cache information. Since EZcp does not have access to the current entry state, all read operations access the NPS.

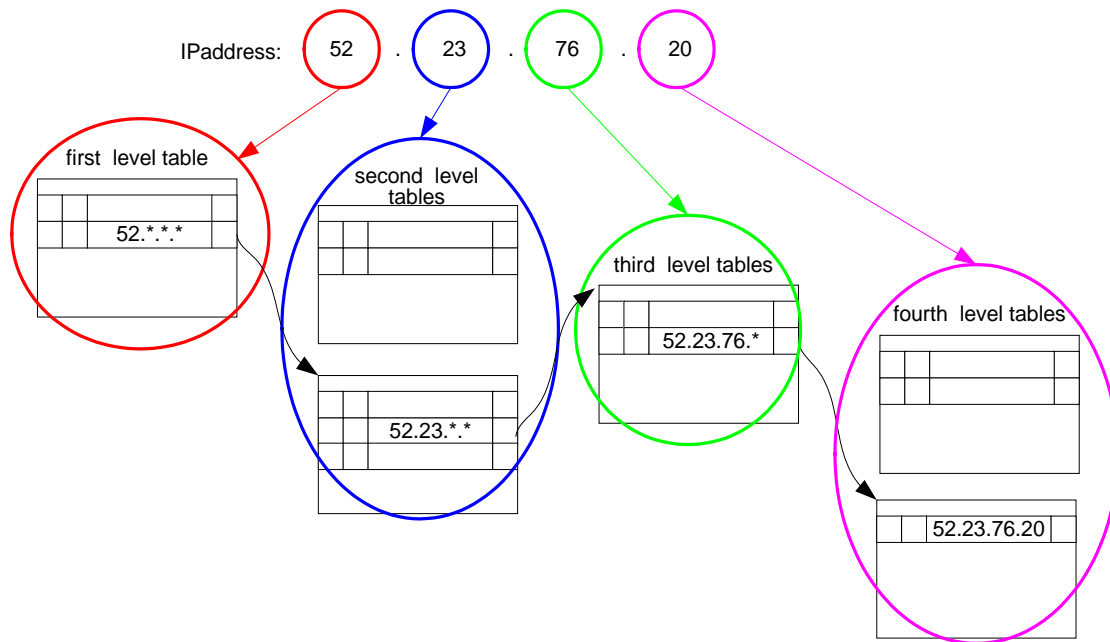
11.4 UltraIP Structures

EZchip's UltraIP data structure implements the routing table as a series of direct access tables as opposed to a Patricia tree structure. The UltraIP is ideally suited for IPv4 and IPv6 routing tables.

The UltraIP data structure greatly improves lookup performance and the control plane update performance by reducing both the number of memory accesses required during the lookup operation and the number of PCI accesses required during the control plane update operations.

The UltraIP lookup structure implements a series of direct tables fashioned in a hierarchy. Each portion of the IP address (segment) is represented by a direct table in the hierarchy. The most significant portion of the IP address is represented in the first-level table and the least significant portion of the IP address is represented in the last-level tables. Each table entry corresponds to a different value in the segment.

Figure 11-2. Table hierarchy based on segments of IP address



UltraIP is based on the M-trie algorithm, taking advantage of the fact that a relatively small percentage of IPv4 routing rules have more than 24 specified bits, and distributes the routing table among several levels of direct access tables. This ensures small and deterministic number of lookups for both the worst case and average case.

A standard M-trie implementation uses a direct table lookup for each 8-bit segment of the IP address. This can lead to cases where a single entry requires allocating a full table for every segment of the IP address, leading to very large worst case memory usage.

The UltraIP structure improves this by compressing sparse data using more efficient encodings. The UltraIP structure adds additional information to each line of the table, allowing the use of several compressed table formats in addition to the full table format.

11.4.1 Defining UltraIP Structures


The following main parameters are defined for UltraIP structures:

General Parameters:

- **Key size** – The key size defines the number of bytes of the key data. Supported key sizes are 2-36 bytes.
- **Result Size** – The result size defines the size of each of the user results. UltraIP structures support a user result sizes of 4 bytes (1 byte control, 3 bytes user data).
- **Number of entries** – The number of entries defines the maximum number of rules in the UltraIP structure.
- **Memory Area** – The memory area to use can be external memory, or mixed (combination of the two).

UltraIP-specific Parameters:

- **Multi-Channel Data Mode** – Specifies if the structure has identical data on all channels, or if the structure's data may differ between channels.

 UltraIP parameters are set in the following manner:

```
Set:      EZapiStruct_ConfigCmd_SetUltraIPParams
Get:      EZapiStruct_StatCmd_GetUltraIPParams
Struct:   EZapiStruct_UltraIPParams
```

11.4.2 Memory Management

For an external UltraIP structure, all its tables reside in the external memory.

For an internal UltraIP structure, all its tables reside in the internal memory.

For a mixed UltraIP structure, the root table always resides in the internal memory, while all the next tables reside in the external memory, except for the tables chosen by the Optimize operation to be moved to the internal memory.

The following parameters define the UltraIP memory management:

Memory Size:

The memory size controls the amount of memory allocated, including the root table. This allows users to explicitly control the amount of memory allocated. By default, this field is set to “AUTO”, in which case EZcp internally determines the amount of allocated memory.

Memory Size Percent:

This is used to specify the percentage of worst case memory to allocate for the structure. By default, EZcp attempts to allocate 100% of the worst case.

Internal Memory Size:

The internal memory size controls the amount of internal memory allocated in case of a mixed UltraIP. The valid values are all the values between 1KB and up to 16MB. The default value is set to 256KB.

Root Table/Next Tables Memory Space Indexes:

Defines the memory space used for root table and next tables.

 UltraIP memory management parameters are set in the following manner:

```
Set:      EZapiStruct_ConfigCmd_SetUltraIPMemMngParams
Get:      EZapiStruct_StatCmd_GetUltraIPMemMngParams
Struct:   EZapiStruct_UltraIPMemMngParams
```

11.4.3 Operation and Entries

UltraIP structures support entries defined using ternary keys, i.e. keys comprised of bits with values 0, 1 or wildcard (don't care). UltraIP structures are used for ternary based applications with fixed length keys, and prefix wildcards (e.g. IP Longest Prefix Match).

UltraIP entries consist of a key and mask representing a subnet, as well as an associated result.

All entry operations to UltraIP structure are immediately written to the NPS.

For UltraIP structures, EZcp can handle an ECC error even if the structure has only a single copy.

Optimize UltraIP:

UltraIP structures support an Optimize operation which aims to decrease the number of external memory accesses. This is done by finding the "hotspots" in the UltraIP tree and choosing better formats for them, as well as moving tables from external memory to internal memory.

11.5 Algorithmic TCAM

Algorithmic TCAM can be used for ternary lookup operations. It uses two parallel accesses to the internal TCAM and then a series of memory accesses to get the final result.

IMPORTANT: In order to use the Algorithmic TCAM, the Algorithmic TCAM structure should be enabled and an internal TCAM profile configured using the TCAM Group API routines.

The first table in the internal TCAM profile should be of Algorithmic TCAM type.


11.5.1 Defining Algorithmic TCAM Structures

The following main parameters are defined for Algorithmic TCAM structures:

General Parameters:

- Key size – The key size defines the number of bytes of the key data. Supported key sizes are 1-80 bytes.
- Result Size – The result size defines the size of each of the user results.
- Number of entries – The number of entries defines the maximum number of rules in the Algorithmic TCAM structure.
- Memory Area – The memory area to use can be external memory, or mixed (combination of the two).
- Maximum number of external accesses – The number of external accesses allowed in the Algorithmic TCAM lookup flow.
- Maximum number of internal accesses – The number of internal accesses allowed in the Algorithmic TCAM lookup flow.
- Maximum number of cycles – The number of hardware cycles allowed in the Algorithmic TCAM lookup flow.

In future releases, advanced parameters for controlling the number of internal TCAM lines and footprint will be supported.

 Algorithmic TCAM parameters are set in the following manner:

```
Set:      EZapiStruct_ConfigCmd_AlgTCAMParams
Get:      EZapiStruct_StatCmd_GetAlgTCAMParams
Struct:   EZapiStruct_AlgTCAMParams, EZapiStruct_AlgTCAMRangeParams
```

The AlgTCAM entry has a key (data) and a mask. In order for a given key to match an entry, all the key's bits must match their respective masked bits in the entry.


The AlgTCAM lookup returns the Rank of the entry it matched. This entry is the one with the lowest rank that matches the given lookup key.

The commands supported for AlgTCAM entries are:

- EZapiStruct_AddEntry – for the addition of entries
- EZapiStruct_ConfigCmd_RebuildAlgTCAM Rebuild – to rebuild the AlgTCAM search structure database; it is also used for the first build of the database.
- EZapiStruct_DeleteAllEntries – to delete all entries

11.5.2 Memory Management

In future releases EZcp will support advanced parameters controlling memory type and index for each Algorithmic TCAM component.

 Algorithmic TCAM memory management parameters are set in the following manner:


| | |
|---------|--|
| Set: | EZapiStruct_ConfigCmd_SetAlgTCAMMemMngParams |
| Get: | EZapiStruct_StatCmd_GetAlgTCAMMemMngParams |
| Struct: | EZapiStruct_AlgorithmicTCAMMemMngParams |

11.5.3 Operation and Entries

For the Algorithmic TCAM structure, entries can be defined using the EZapiStructField structure.

- The initial set of entries is added to the algorithmic TCAM structure, the algorithmic TCAM needs to be built for the first time. The initial algorithmic TCAM build is also performed using the algorithmic TCAM rebuild API.
- When performing pending entry operations, the changes in the entries are not immediately incorporated into the algorithmic TCAM structure, but are rather buffered within the CP library until the next rebuild operation. Thus, the algorithmic TCAM structure needs to be rebuilt in order to incorporate the changes in the entries into the existing algorithmic TCAM.
- When performing immediate entry operations, the changes in the entries are performed on the existing algorithmic TCAM structure within the NPS. Since the operations are performed on an existing algorithmic TCAM in a local manner, the algorithmic TCAM structure's balancing and memory usage may degrade over time. To handle such degradations the algorithmic TCAM may be rebuilt periodically (see explanations on the Algorithmic TCAM State API in "[Search Structure States](#)" section regarding how to determine when an Algorithmic TCAM structure should be rebuilt).

▶ *In the current release only pending entry operations are supported.*

 To rebuild an Algorithmic TCAM structure use:

| | |
|---------|--|
| Set: | EZapiStruct_ConfigCmd_RebuildAlgTCAM |
| Struct: | EZapiStruct_AlgorithmicTCAMRebuildParams |

11.6 Summary of Commands – Struct Group

Table 11-3. Search Structures Group routines

| API Routines | | |
|---------------------------------|----------------------------|--------------------------------|
| EZapiStruct_AddEntry() | EZapiStruct_DeleteEntry() | EZapiStruct_PartitionConfig() |
| EZapiStruct_AdvancedLookup() | EZapiStruct_GetEntry() | EZapiStruct_PartitionStatus() |
| EZapiStruct_Config() | EZapiStruct_Iterate() | EZapiStruct_Status() |
| EZapiStruct_DeleteAllEntries() | EZapiStruct_Lookup() | EZapiStruct_UpdateEntry() |

Table 11-4. Summary of Search Structures Group commands

This table provides a summary of the routines/commands in the Search Structures Group, as well as the structure state in which each command may be performed.

| | Undefined | Created | Loaded |
|---|-----------|----------|----------|
| TOPIC & COMMAND | U | C | L |
| Search Structures Routines | | | |
| Set: EZapiStruct_ConfigCmd_SetStructParams | - | ✓ | ✓ |
| Get: EZapiStruct_StatCmd_GetStructParams | - | ✓ | ✓ |
| Table Structure | U | C | L |
| Defining table structures | | | |
| Set: EZapiStruct_ConfigCmd_SetTableParams | - | ✓ | ✓ |
| Get: EZapiStruct_StatCmd_GetTableParams | - | ✓ | ✓ |
| Memory management | | | |
| Set: EZapiStruct_ConfigCmd_SetTableMemMngParams | - | ✓ | - |
| Get: EZapiStruct_StatCmd_GetTableMemMngParams | - | ✓ | ✓ |
| Hash Structure | U | C | L |
| Defining hash structures | | | |
| Set: EZapiStruct_ConfigCmd_SetHashParams | - | ✓ | ✓ |
| Get: EZapiStruct_StatCmd_GetHashParams | - | ✓ | ✓ |
| Memory management | | | |
| Set: EZapiStruct_ConfigCmd_SetHashMemMngParams | - | ✓ | - |
| Get: EZapiStruct_StatCmd_GetHashMemMngParams | - | ✓ | ✓ |
| UltraIP Structure | U | C | L |
| Defining UltraIP structure | | | |
| Set: EZapiStruct_ConfigCmd_SetUltraIPParams | - | ✓ | ✓ |
| Get: EZapiStruct_StatCmd_GetUltraIPParams | - | ✓ | ✓ |
| Memory management | | | |
| Set: EZapiStruct_ConfigCmd_SetUltraIPMemMngParams | - | ✓ | - |
| Get: EZapiStruct_StatCmd_GetUltraIPMemMngParams | - | ✓ | ✓ |
| UltraIP structure rebuild | | | |
| Set: EZapiStruct_ConfigCmd_RebuildUltraIP | - | - | ✓ |
| Algorithmic TCAM Structure | U | C | L |
| Defining Algorithmic TCAM structures | | | |
| Set: EZapiStruct_ConfigCmd_SetAlgTCAMParams | - | ✓ | ✓ |
| Get: EZapiStruct_StatCmd_GetAlgTCAMParams | - | ✓ | ✓ |

| TOPIC & COMMAND | U | C | L |
|---|----------|----------|----------|
| Memory management | | | |
| Set: EZapiStruct_ConfigCmd_SetAlgTCAMMemMngParams | - | ✓ | - |
| Get: EZapiStruct_StatCmd_GetAlgTCAMMemMngParams | - | ✓ | ✓ |
| Algorithmic TCAM rebuild | | | |
| Set: EZapiStruct_ConfigCmd_RebuildAlgTCAM | - | - | ✓ |
| Entry Operations | U | C | L |
| Manipulating search structure entries | | | |
| Set: EZapiStruct_AddEntry() | - | - | ✓ |
| Set: EZapiStruct_UpdateEntry() | - | - | ✓ |
| Set: EZapiStruct_DeleteEntry() | - | - | ✓ |
| Set: EZapiStruct_DeleteAllEntries() | - | - | ✓ |
| Get: EZapiStruct_GetEntry() | - | - | ✓ |
| Get: EZapiStruct_Lookup() | - | - | ✓ |
| Get: EZapiStruct_AdvancedLookup() | - | - | ✓ |
| Get: EZapiStruct_Iterate() | - | - | ✓ |
| Iterate Commands | U | C | L |
| EZapiStruct_IterateCmd_Create | - | - | ✓ |
| EZapiStruct_IterateCmd_Locate | - | - | ✓ |
| EZapiStruct_IterateCmd_GetNext | - | - | ✓ |
| EZapiStruct_IterateCmd_GetPrev | - | - | ✓ |
| EZapiStruct_IterateCmd_Delete | - | - | ✓ |
| Misc. Commands | U | C | L |
| Get: EZapiStruct_StatCmd_GetNumEntries | - | - | ✓ |
| Get: EZapiStruct_StatCmd_GetMemUsage | - | ✓ | ✓ |

Undefined
Loaded

| TOPIC & COMMAND | U | L |
|--|----------|----------|
| Partition Commands (based on the partition state) | U | L |
| Memory partition | | |
| Set: EZapiStruct_PartitionConfigCmd_LoadPartition | ✓ | - |
| Set: EZapiStruct_PartitionConfigCmd_UnloadPartition | - | ✓ |
| Set: EZapiStruct_PartitionConfigCmd_AddChannel | ✓ | ✓ |
| Set: EZapiStruct_PartitionConfigCmd_DeleteChannel | ✓ | ✓ |
| Get: EZapiStruct_PartitionStatCmd_GetPartitionStateInfo | ✓ | ✓ |

12. Reference

EZcp API routines, structures and enumerations are not provided in this document; they can be found in the on-line help file accessible from the EZide SDK.