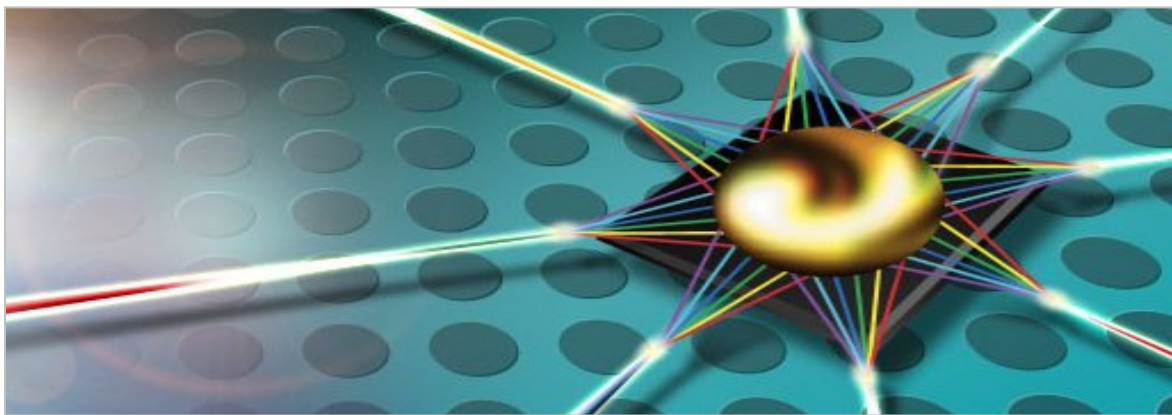




# NPS-400 Programming Manual

## Programming Basics for NPS-400 Network Processors

Document Version 3.0



**Document Number: 27-8158-03**

The information contained is proprietary and confidential.

## Preface

©2015 EZchip Semiconductor Ltd. EZchip is a registered trademark of EZchip Semiconductor Ltd. Linux<sup>®</sup> is the registered trademark of Linus Torvalds in the U.S. and other countries. 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](mailto:info@ezchip.com), Web: [www.ezchip.com](http://www.ezchip.com)

EZchip welcomes your comments on this publication. Please address them to: [supportNP@ezchip.com](mailto:supportNP@ezchip.com).

## About this Manual

This document offers programmers with the basic information required to code a data plane processing application for the EZchip NPS-400 network processor.

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

The *Release Notes* may contain information supplemental to this document.

## Related Documents

For additional information refer to:

Document	Content
<i>NPS-400 Architectural Specification</i>	Overview of the architecture, feature set and functionality of the NPS-400 network processor.
<i>NPS-400 CTOP Architecture Manual</i>	Architecture, programming and instruction set for NPS C-programmable Task Optimized Processors (CTOPs).
<i>NPS-400 EZdp Reference Manual</i>	Describes the EZchip Data Plane Services library (EZdp) and its related APIs. The EZdp library provides an application programming interface (API) for data-plane applications running on NPS-400 network processors, abstracting the complexities of the underlying CTOP core instruction set and various hardware accelerators.
<i>NPS-400 EZcp Reference Manual</i>	Describes the EZchip Control Plane Services library and its related APIs. The EZcp library provides an API for initialization, configuration, management and monitoring of the NPS network processor for control plane applications.
<i>NPS-400 CPE Developer's Guide</i>	Describes the EZdk Control Plane Environment (CPE) libraries used to development control plane applications for systems based on NPS network processors.
Various application notes	Outlining the reference applications supplied.

## This Document

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

Chapter	Name	Description
Section 1	<b>Overview</b>	General information required for programming of the NPS-400 including its data flow.
Section 2	<b>NPS Control Plane / Data Plane Overview</b>	TBD
Section 3	<b>First Steps</b>	Step by step description on how to build your first data plane application, describing the basic steps and details needed to write a new data plane processing application.
Section 4	<b>Advanced Features</b>	Discusses thread scheduling and managing the job pipeline from a software point of view.
Section 5	<b>Sample Reference Applications and Available Libraries</b>	The NPS-400 software environment includes a rich set of sample applications and libraries.
Appendix A	<b>Acronyms and Abbreviations</b>	Lists the acronyms and abbreviations used in this document.

## Revision History

Revision	Date	Description of Modification
3.0	Sept. 30, 2015	Significant improvements to: Chapter 4 <a href="#">Advanced Features</a> Chapter 5 <a href="#">Sample Reference Applications and Available Libraries</a> .
2.1	June 15, 2015	This release includes chapters 1-2 which have been fully revised, including the sample code to reflect EZdk v1.8a. Future releases will contain improvements to the rest of the chapters.
1.1	April 30, 2014	Updates throughout the document.
1.0	Feb. 21, 2013	Initial release.

## Terminology and Conventions

### General

The following terminology is used throughout this document:

Term	Description
BMU	<a href="#">Buffer Management Unit</a>
COS	Class of Service
CP	Control Plane which is the term used for the EZcp code
CTOP	<a href="#">C-programmable Task Optimized Processor</a> , the term used for each of the NPS-400 256 processor cores
ECC	Error-correcting code memory
ELF	Executable and Linkable Format
FMT	<a href="#">Fixed Mapping Table</a>
FW	Firewall
Host	Refers to the host CPU. In other systems, the host may also be referred to as the target.
ICU	<a href="#">Input Classification Unit</a>
LAG	Link Aggregation Group
LBD	<a href="#">Linked Buffer Descriptors</a>
MTM	<a href="#">Multi-Threading Management unit</a>
NDMA	<a href="#">Network DMA</a>
NPC	Network Processing
NPS	Refers to the EZchip NPS-400 network processor device and/or software simulator.
PMU	<a href="#">Processor Management Unit</a>
RxIFU	<a href="#">Receive interface unit</a>
SoC	System on Chip
STAT	<a href="#">Statistics Counters Manager</a>
Structure	Used for both NPS-400 search structures (i.e. lookup tables) as well as C code structures (i.e. group of parameters).
TCAM	<a href="#">Ternary Content Addressable Memory</a>
TLB	<a href="#">Translation Look aside Buffers</a>
TM	<a href="#">Traffic Manager</a>
TxIFU	<a href="#">Transmit interface unit</a>
ZOL	<a href="#">Zero Overhead Linux</a>

### Typographical Conventions



Provide additional information that is not necessarily mandatory, such as tips and notes.



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

**Warning!** Contains information that is mandatory for proper configuration 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
Contents .....	5
List of Figures .....	11
<b>A. Acronyms and Abbreviations.....</b>	<b>A-1</b>
<b>1. Overview .....</b>	<b>12</b>
1.1 About this Manual .....	12
1.2 NPS-400 Processor Overview .....	13
1.2.1 C-programmable Task Optimized Processor (CTOP).....	13
1.2.2 Interface Units (RxIFU/TxIFU) .....	13
1.2.3 Processor Management Units (PMU).....	13
1.2.4 Input Classification Unit (ICU).....	14
1.2.5 Buffer Manager Unit (BMU) .....	14
1.2.6 Network DMA (NDMA).....	14
1.2.7 Multi-Threading Management Unit (MTM) .....	14
1.2.8 Network Processing Cluster (NPC).....	14
1.2.9 EZnet.....	14
1.2.10 Traffic Managers (TM) .....	14
1.2.11 Statistics Counters Manager (STAT) .....	15
1.2.12 Ternary Content Addressable Memory (TCAM) .....	15
1.2.13 DMA .....	15
1.2.14 Cryptographic Engines.....	15
1.3 Zero Overhead Linux .....	16
1.3.1 How does one use Zero Overhead Linux? .....	16
1.3.2 How does Zero Overhead Linux work? .....	17
1.3.2.1 The Fixed Mapping Table.....	17
1.3.2.2 Isolating the data plane application from OS interference.....	17
1.3.3 Data Plane Application Life Cycle.....	18
1.3.3.1 Startup.....	18
1.3.3.2 Packet Processing .....	18
1.3.3.3 System Call .....	18
1.3.3.4 Exceptions and Debugging .....	19
1.3.4 Performance Provided by Zero Overhead Linux.....	19
1.4 Memory Hierarchy .....	20
1.4.1 Memory Types .....	20
1.4.1.1 External Memory (EMEM).....	20
1.4.1.2 Internal Memory (IMEM).....	21
1.4.1.3 Core's Local Memory (CMEM) .....	21
1.4.1.3.1 Thread Private CMEM Area .....	22
1.4.1.3.2 Core Shared CMEM Area.....	22
1.4.1.3.3 L1 Data Cache Area.....	22
1.4.2 Memory Space Configuration .....	23
1.4.2.1 External Memory Spaces .....	23
1.4.2.2 Internal Memory Spaces .....	23
1.4.3 Address Translation.....	24
1.4.3.1 Core Level Virtual Memory Translation .....	24

1.4.3.2	The Translation Look Aside Buffers .....	24
1.4.3.3	The Fixed Mapping Table.....	24
1.5	Multi-threading.....	25
1.5.1	Thread States .....	26
1.5.2	Thread Scheduling.....	26
1.5.2.1	Scheduling Events.....	26
1.5.2.2	Pending Transactions.....	27
1.5.2.3	Interrupts.....	27
1.5.2.4	Eligibility Criteria .....	28
1.6	Network Interfaces .....	29
1.6.1	Engines and Interfaces .....	29
1.6.2	Input and Output Channels and Queues .....	29
1.6.3	Logical Channels.....	29
1.6.4	Traffic Manager.....	29
1.7	Frame Flow .....	30
<b>2.</b>	<b>NPS Control Plane / Data Plane Overview .....</b>	<b>32</b>
<b>3.</b>	<b>First Steps.....</b>	<b>33</b>
3.1	Basic Guidelines.....	33
3.2	Hello Data Plane: frame in, frame out.....	33
3.2.1	The Bare Necessities: a data plane application program skeleton.....	34
3.2.1.1	Include files.....	34
3.2.1.2	Code and data section assignments.....	34
3.2.1.2.1	Placing private variables in the core's private local memory.....	34
3.2.1.2.2	Placing shared variables in the core's local memory .....	35
3.2.1.2.3	Placing functions into a specific fixed mapping memory .....	35
3.2.1.2.4	Placing variables into a specific fixed mapping memory.....	36
3.2.1.3	Initialization.....	36
3.2.1.4	Running frame processing code.....	38
3.2.2	Working with Jobs .....	38
3.2.2.1	The job ID.....	38
3.2.2.2	The job descriptor .....	38
3.2.2.2.1	The job embedded frame descriptor.....	39
3.2.2.2.2	Receive-side job information structure .....	40
3.2.2.2.3	Transmit-side job information structure.....	40
3.2.2.3	Getting a new job.....	40
3.2.2.3.1	Receiving a job and a job descriptor .....	40
3.2.2.4	Finalizing a job .....	41
3.2.2.4.1	Transmitting a frame .....	41
3.2.2.4.2	Dropping a frame .....	43
3.2.2.4.3	Advanced options.....	43
3.3	Frame Headers and Payload.....	44
3.3.1	The Frame .....	44
3.3.1.1	Frame buffers.....	44
3.3.1.1.1	Buffer management.....	44
3.3.1.2	Buffer descriptors .....	44
3.3.1.2.1	Linked Buffer Descriptors.....	45
3.3.1.3	The frame descriptor.....	45
3.3.2	Working with Frame Buffers .....	46
3.3.2.1	Accessing and modifying frame data.....	47
3.3.2.1.1	Getting a descriptor of the frame's first data buffer .....	47
3.3.2.1.2	Loading the frame data into local core memory.....	47
3.3.2.1.3	Querying and modifying frame data .....	48
3.3.2.1.4	Adjusting the frame metadata .....	48
3.3.2.1.5	Synching frame data to global memory .....	49
3.3.2.1.6	Tying all the details together.....	50

3.3.2.2	Accessing the frame payload .....	51
3.3.2.3	Allocating and releasing data buffers.....	51
3.3.2.3.1	Buffer pools.....	51
3.3.2.3.2	Allocating a single buffer.....	52
3.3.2.3.3	Releasing a single buffer.....	52
3.3.2.3.4	Allocating and releasing multiple buffers .....	52
3.4	Parsing Frame Headers.....	53
3.4.1	Protocol Decoders .....	53
3.4.1.1	Principle of operation.....	53
3.4.1.2	An example: the MAC protocol decoder .....	53
3.5	Counter Acceleration.....	57
3.5.1	On-demand Counters.....	58
3.5.1.1	Single counters.....	58
3.5.1.2	Dual counters .....	59
3.5.2	Posted Counters.....	59
3.5.2.1	Initializing the counter .....	60
3.5.2.2	Increment the counter .....	60
3.6	Hardware-assisted Search Structures .....	61
3.6.1	Search Memory .....	61
3.6.2	Direct Table Example.....	62
3.6.2.1	Key and result structures.....	62
3.6.2.2	Table descriptor initialization .....	62
3.6.2.3	Looking up a table entry .....	63
3.6.2.4	Updating the table .....	64
3.6.3	Hash Example .....	65
3.6.3.1	Key and result structures.....	65
3.6.3.2	Hash descriptor initialization .....	65
3.6.3.3	Lookup a hash entry.....	66
3.6.3.4	Updating the hash .....	67
<b>4.</b>	<b>Advanced Features .....</b>	<b>69</b>
4.1	DMA and Memory Operations.....	69
4.1.1	Memory Subsystem.....	69
4.1.1.1	Memory Protection .....	69
4.1.1.1.1	In-band ECC Scheme.....	69
4.1.1.1.2	Out-of-band ECC Scheme.....	70
4.1.1.2	Memory Allocation.....	70
4.1.1.2.1	IMEM.....	70
4.1.1.2.1	EMEM.....	71
4.1.2	Referring to Data Allocated Memory.....	72
4.1.2.1	Virtual Addressing.....	72
4.1.2.2	HW Accelerator Addressing.....	72
4.1.3	Memory Operations.....	73
4.1.4	Atomic Operations .....	75
4.2	Controlling the Jobs Pipeline.....	77
4.2.1	Mapping Job to Application .....	78
4.2.1.1	Order restoration .....	80
4.2.2	Handling Jobs in Data Plane Applications .....	80
4.2.2.1	Getting a job.....	80
4.2.2.2	Job Information.....	81
4.2.2.2.1	Job type specific receive-side information.....	81
4.2.2.2.2	Job type agnostic receive-side information .....	83
4.2.3	Transmit a Job .....	84
4.2.3.1	Transmitting a frame to an interface through TM.....	84
4.2.3.2	Transmitting a frame to an interface (TM bypass).....	84
4.2.3.3	Discarding a job .....	84

4.2.4	Pipeline vs. Run to Completion.....	85
4.2.4.1	Run to completion.....	85
4.2.4.2	Pipeline .....	86
4.2.5	Creating a Job.....	87
4.2.5.1	Sequence numbering.....	88
4.2.5.2	Outstanding job limit .....	88
4.3	Timer Mechanisms .....	89
4.3.1	Handling Timer Jobs in DP Application .....	89
4.4	Advanced Frame and Frame Buffer Manipulation.....	91
4.4.1	Linked Buffer Descriptors.....	91
4.4.2	Frame Types.....	93
4.4.3	The Frame Descriptor.....	94
4.4.4	Manipulating the frame buffer structure.....	96
4.4.4.1	Getting hold of the buffer descriptor of a standard frame.....	96
4.4.4.2	Determining the frame buffer data length.....	96
4.4.4.3	Working with linked buffer descriptors .....	97
4.4.4.4	Loading the first data buffer of an extended frame.....	98
4.4.4.5	Calculating the size of the linked buffer descriptor .....	98
4.4.4.5.1	Iterating over all of a frame's data buffers.....	99
4.4.4.5.2	Storing a linked buffer descriptor back to global memory.....	100
4.4.4.6	Copying data between frame buffers .....	100
4.4.4.7	Copying linked buffer descriptors between frame buffers.....	100
4.4.4.8	Cloning linked buffer descriptors between frame buffers.....	101
4.5	Advanced Search Features .....	102
4.5.1	Scanning and Aging Mechanism.....	102
4.5.1.1	Aging direct tables .....	102
4.5.1.2	Aging hash structures.....	102
4.5.2	TCAM.....	103
4.5.2.1	Internal TCAM .....	104
4.5.2.2	External TCAM .....	106
4.5.2.3	Algorithmic TCAM .....	106
4.5.3	UltraIP.....	107
4.5.4	Low level API .....	107
4.6	Precision Time Protocol IEEE 1588.....	109
4.6.1	Sampling the Receive Timestamp of an Incoming Frame .....	110
4.6.2	Sampling the Transmit Timestamp of an Outgoing Frame in 2-Step Mode .....	111
4.6.3	Sampling the Transmit Timestamp of an Outgoing Frame in 1-Step Mode .....	112
4.7	Traffic Management.....	114
4.7.1	QoS Scheduling.....	114
4.7.2	Output Channel Selection.....	115
4.7.2.1	Topology-based packet-switch index selection mode.....	115
4.7.2.2	Fixed packet-switch index selection mode .....	115
4.7.2.3	Explicit packet-switch index selection mode.....	115
4.7.3	Sending a Frame to the TM.....	116
4.7.3.1	TM information.....	116
4.7.3.2	Finalizing the job .....	117
4.7.4	Frame Loopback.....	118
4.7.5	TM IMEM Data Cache .....	118
4.8	Multicast and Replication.....	119
4.8.1	Multicast Reference Counters .....	119
4.8.1.1	Frame multicast control .....	119
4.8.1.2	Setting frame buffer multicast reference counter.....	119
4.8.1.3	Additional multicast reference counter operations .....	120
4.8.2	Creating a replica specific buffer .....	120
4.8.3	Heavyweight Multicast (HW/TM Multicast).....	120



4.8.3.1	Requesting replication service .....	120
4.8.3.2	Receiving replicated frames .....	120
4.8.4	Lightweight Multicast .....	121
4.8.5	Multicast Example.....	121
4.9	General Purpose Pools.....	122
4.9.1	Index Pool .....	122
4.9.2	Memory Pool.....	123
4.10	Mathematical Operations .....	125
4.10.1	Logical, Arithmetical and Bit Manipulation Operations .....	125
4.10.2	Hash, CRC and Internet Checksum Computation .....	125
4.10.2.1	Hash .....	125
4.10.2.2	CRC .....	126
4.10.2.3	Checksum.....	126
4.11	Security.....	129
4.11.1	Architecture.....	130
4.11.1.1	Context Memory .....	130
4.11.2	Launching a Security Task .....	131
4.11.2.1	Digest Task .....	131
4.11.2.1.1	Initialization .....	131
4.11.2.1.2	Data Processing.....	131
4.11.2.1.3	Finalization.....	132
4.11.2.1.4	MD5 Calculation Example.....	132
4.11.2.1.5	HMAC-SHA1 Calculation Example .....	133
4.11.2.2	Encryption/Decryption.....	134
4.11.2.2.1	Initialization .....	134
4.11.2.2.2	Processing Data.....	134
4.11.2.2.3	Finalization.....	134
4.11.2.2.4	3DES-CBC Encryption Example .....	134
4.11.2.2.5	AES-CBC-256 Encryption Example.....	135
4.11.2.3	GCM .....	136
4.11.2.3.1	Initialization .....	136
4.11.2.3.2	Processing Data.....	137
4.11.2.3.3	Finalization.....	137
4.11.2.3.4	AES-GCM-128 Encryption and Digest Example .....	137
4.11.2.4	Security Context State.....	139
4.12	FCU – Budgeting - TBD .....	140
4.13	IPC – TBD.....	140
4.14	PCI.....	140
4.14.1	NPS PCI Overview.....	140
4.14.2	PCI Related Terminology.....	140
4.14.3	NPS PCI Programming Approaches .....	141
4.14.4	High level SW Architecture .....	142
4.14.5	PNI Application Overview .....	143
4.14.6	PNI Application Integration .....	143
4.14.7	Multiple PCI Device Implementations.....	143
4.15	Lock Mechanism – TBD .....	143
4.16	Data Structures – TBD .....	143
4.16.1	Queue – TBD .....	143
4.16.2	List – TBD.....	143
4.17	Advanced Statistic Counter Features.....	144
4.17.1	On-demand Statistic Counters.....	144
4.17.1.1	Bitwise Counters.....	144
4.17.1.1.1	Implementing semaphores .....	144
4.17.1.1.2	Implementing a state-machine .....	144
4.17.1.2	Counters Message Queue.....	144

4.17.1.3	Shadow Counters with Synchronized Swapping .....	145
4.17.1.4	Token Bucket Counters.....	146
4.17.1.5	Hierarchical Token Bucket Counters .....	146
4.17.1.6	Watchdog Counters.....	146
4.17.2	Posted Statistic Counters .....	147
4.17.2.1	Shadow Counters with Synchronized Swapping .....	147
4.17.2.2	Reporting and counters message queue .....	147
4.18	Controlling Hardware Thread Scheduling.....	149
4.18.1	Asynchronous Operations .....	149
4.18.2	Synchronization and Scheduling .....	149
4.18.2.1	Schedule out and synchronize on all pending transactions .....	149
4.18.2.2	Schedule out and synchronize on read transactions .....	150
4.18.3	Using Asynchronous Operations to Achieve Parallelism.....	150
<b>5.</b>	<b>Sample Reference Applications and Available Libraries .....</b>	<b>151</b>
5.1	Background .....	151
5.2	Software Libraries .....	151
5.2.1	Stateful Flow Table (SFT).....	151
5.2.1.1	Library Overview .....	151
5.2.1.2	Library Description .....	152
5.2.1.3	What is an SFT Flow? .....	153
5.2.1.4	Library Feature List .....	153
5.2.1.5	Sample Code .....	154
5.2.2	Crypto Library .....	155
5.3	Sample Applications.....	156
5.3.1	CESR Reference Application .....	156
5.3.1.1	Application Overview .....	156
5.3.1.2	Application Description .....	156
5.3.1.3	Application Feature List .....	157
5.3.2	DPI based Application Recognition .....	158
5.3.3	Lawful Interception and Content Filtering .....	159
5.3.4	IPsec Sample Application .....	159
5.3.5	L23QoS Sample Application.....	160
5.3.6	TCP Stack.....	160
5.4	Smaller Reference Applications .....	162
5.4.1	PCI Endpoint Stack Reference Application .....	162
<b>Appendix A:</b>	<b>Acronyms and Abbreviations .....</b>	<b>A-1</b>

## List of Figures

Figure 1. NPS-400 block diagram .....	13
Figure 2. NPS-400 Zero Overhead Linux .....	16
Figure 3. Data plane application life cycle .....	18
Figure 4. Zero Overhead Linux performance for 3 environment configurations (less is better) .....	19
Figure 5. NPS-400 memory diagram.....	20
Figure 6. CMEM virtual and physical views.....	21
Figure 7. Memory Translation.....	24
Figure 8. NPS-400 thread states .....	26
Figure 9. Typical frame flow .....	30
Figure 10. Hello data plane - data plane application skeleton example.....	33
Figure 11. Handling the job by the application .....	77
Figure 12. Mapping of jobs to an application.....	78
Figure 13. Example of packet processing applications organized in a software pipeline .....	86
Figure 14. NPS-400 buffer descriptors.....	91
Figure 15. NPS-400 frame types .....	93
Figure 16. Internal TCAM lookup example .....	105
Figure 17. NPS TM QoS scheduling .....	114
Figure 18. Relationship between NPS security components .....	130
Figure 19. Security session diagram.....	131
Figure 20. PNI Application Approach (“closed model”) vs. PNI Library Approach (“distributed model”)....	141
Figure 21. NPS software system architecture as relevant to PNI solution .....	142
Figure 22. Watchdog counter operation .....	147
Figure 23. Sample NPS feature path .....	152
Figure 24. IPsec frame encapsulation, encryption and authentication.....	155
Figure 25. CESR reference application - network topology.....	157
Figure 26. CESR reference application - chassis module.....	157

# 1. Overview

## 1.1 About this Manual

The NPS-400 is a high-end network processor. Its superior performance and functionality are due to EZchip NPS-400's best of breed architecture and hardware accelerators. Programmers using the NPS-400 use C as a programming language and enjoy a standard development environment (e.g. Linux, Eclipse, GCC). While the tools and language are standard, getting the most of the NPS-400 requires knowledge of the underlying architecture and the available libraries and APIs.

This manual is aimed at NPS programmers. [Chapter 1](#) provides an overview of the NPS-400. It addresses key issues such as the memory architecture available, multi-threading model and Zero Overhead Linux. It also includes an overview of the underlying hardware.

Chapter 2, [NPS Control Plane / Data Plane Overview](#), discusses the control plane architecture, APIs and features.

Chapter 3, [First Steps](#), provides the readers with a step by step guide on how to build their first data plane application, describing the basic steps and details needed to write a new data plane processing application.

Chapter 4, [Advanced Features](#), discusses the many NPS-400 data plane features including thread scheduling, job pipeline management and more with all described from a software point of view.

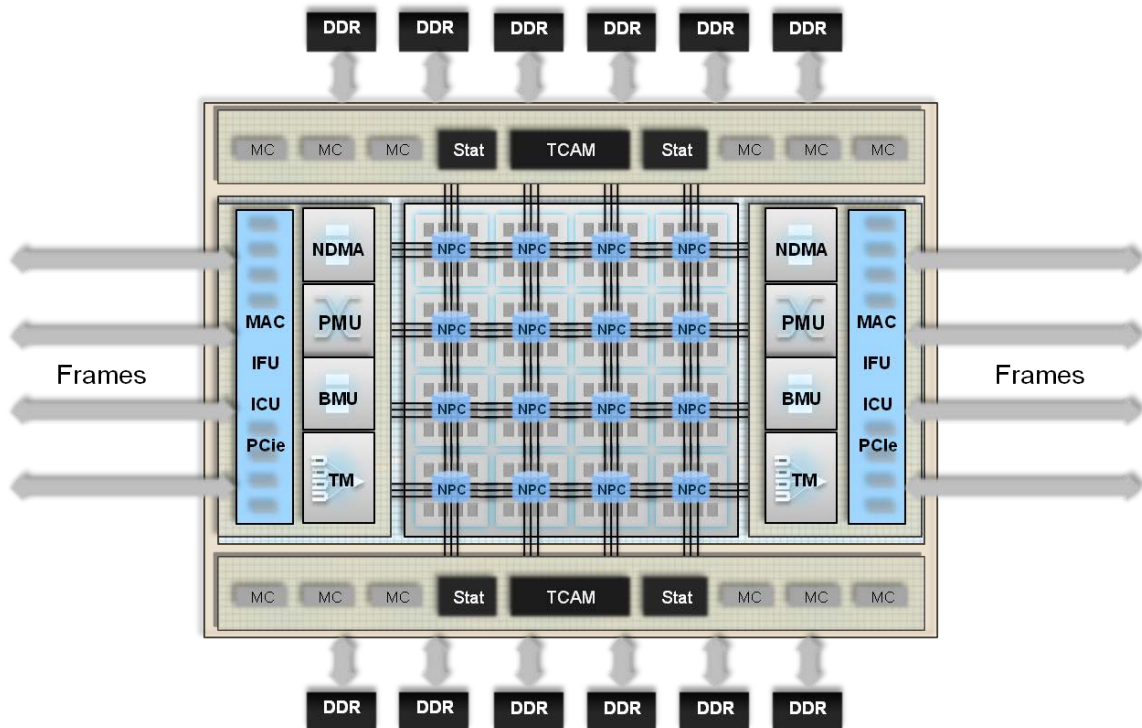
[Chapter 5](#) lists the reference applications, libraries and samples available from EZchip.

EZchip has invested over 500 man years in developing the NPS-400. The breadth and complexity of it cannot be summarized in one document. In this manual we intend to cover the breadth of software features and capabilities. References are provided to the specifications and manuals that described particular aspects in higher depth.

## 1.2 NPS-400 Processor Overview

The NPS-400 is a system-on-a-chip comprised of several main functional modules (see figure below). While being a programming manual it is important that the programmers have a grasp of the underlying hardware. The following sections provide a high level overview of the various hardware units, their names and functions.

Figure 1. NPS-400 block diagram



### 1.2.1 C-programmable Task Optimized Processor (CTOP)

The C-programmable Task Optimized Processor, henceforth referred to as CTOP in this document, is the NPS computing core. The NPS-400 includes 256 such cores.

The CTOP is a multi-threaded core capable of executing general purpose instructions, algorithmic instructions specially crafted to assist in high performance packet processing, and accelerator instructions that invoke the SoC accelerator engines. Each CTOP supports up to 16 hardware threads. This manual will use the terms core and CTOP interchangeably.

### 1.2.2 Interface Units (RxIFU/TxIFU)

The receive and transmit Interface Units, which will be referred to in this document as RxIFU and TxIFU, are the units through which frames flow in and out of the system.

The RxIFU and TxIFU are configurable and allow a variety of MAC configurations and speeds.

Frames may arrive from any input interface and can be sent to any of the output interfaces.

### 1.2.3 Processor Management Units (PMU)

The Processor Management Unit, henceforth referred to as PMU in this document, provides job queuing. The PMU maintains 256 queues of jobs, each representing single quanta of work: a frame to process, an expired timer event or a software context.

Frames flowing into the system are delivered to one of the PMU queues and from there scheduled to be processed by one of the CTOPs.

The PMU is responsible for maintaining ordering (i.e. network order) between different frames belonging to the same queue; frames belonging to the same queue will be processed in parallel in any order by the CTOP cores but will only advance to the next processing state, such as transmission, according to the order enforced by the PMU queue. The PMU distributes the frames *randomly* across the CTOP cores maintaining maximal parallelism while balancing the loads of the cores.

### 1.2.4 Input Classification Unit (ICU)

The Input Classification Unit, henceforth referred to as ICU in this document, provides initial frame parsing and classification services on the incoming traffic and classifies the incoming frames into one of the 256 PMU queues.

### 1.2.5 Buffer Manager Unit (BMU)

The Buffer Management Unit, henceforth referred to as BMU in this document, provides buffer management services for the system. The BMU maintains pools of buffers that are used for frames and ancillary data structures, such as jobs and frame descriptors, as well as software defined buffers.

### 1.2.6 Network DMA (NDMA)

The Network DMA engines, henceforth referred to as NDMA in this document, move incoming frames' data from the RxIFU into BMU supplied buffers in IMEM and EMEM, and from the same buffers to the TxIFU to be transmitted, releasing the used buffers back to the BMU for reuse.

### 1.2.7 Multi-Threading Management Unit (MTM)

The CTOP Multi-Threading Management unit, henceforth referred to as MTM in this document, is responsible for tracking the eligibility of the various CTOP threads to gain access to the execution unit and to select the next thread to be designated as the warm thread; that is, the thread that will be context switched into the execution unit at the next scheduling event.

### 1.2.8 Network Processing Cluster (NPC)

The 256 CTOP cores of NPS-400 are arranged in 16 groups of 16 cores, called Network Processing Clusters, or NPCs. Each NPC consists of an array of 16 CTOPs with local memory, L2 cache for lookup data, protocol parser, DMA and dedicated security accelerator engines and a data bus interface unit to the EZnet and the PMU. In total the NPS-400 has 16 NPCs, 256 CTOPs and 4096 HW threads.

### 1.2.9 EZnet

The EZnet is a highly scalable cross-chip hierarchical mesh crossbar interconnecting the SoC hardware blocks.

### 1.2.10 Traffic Managers (TM)

The Traffic Manager, henceforth referred to as TM in this document, contains configurable, per-flow-queue shapers and a hierarchical traffic scheduler.

The NPS-400 has two TMs, which can either send queued frames through any of the network interfaces, via the TxNDMA unit, or loopback frames to the NPS PMUs.

### 1.2.11 Statistics Counters Manager (STAT)

The Statistics Counters Manager, henceforth referred to as STAT in this document, is a hardware block that accelerates the statistics counter updates and management of the counters for packet processing applications. The STAT efficiently maintains posted and on-demand statistical counters in RAM.

### 1.2.12 Ternary Content Addressable Memory (TCAM)

The Ternary Content Addressable Memory controllers, henceforth referred to as TCAM in this document, provide access to internal and externally connected TCAM units. The TCAM unit may be used standalone or as part of an integrated RAM based algorithmic TCAM expansion.

### 1.2.13 DMA

The DMA provides data transaction services to the CTOPs' threads. After activating the DMA transaction, the CTOP thread can be suspended until the transaction terminates.


The DMA deals with any alignment on source data and any alignment on destination data as well as byte resolution transaction length. A frame data structure may be spread in EMEM or in IMEM in a multi-buffer format. The DMA can also accelerate some bump-on-the-wire operations such as checksum and CRC calculations.

### 1.2.14 Cryptographic Engines

The NPS provides hardware implemented accelerations for common security cryptographic algorithms. There are 16 such cryptographic engines, one per CTOP cluster (i.e. NPC), providing superior performance for the security processing of threads running in these clusters.

The security processing can be designed as either run-to-complete or pipelined.

- *Note that cryptographic features are not supported on all models of the NPS-400 device.*

 For more detailed information about the NPS architecture refer to the *NPS-400 Architectural Specifications*.

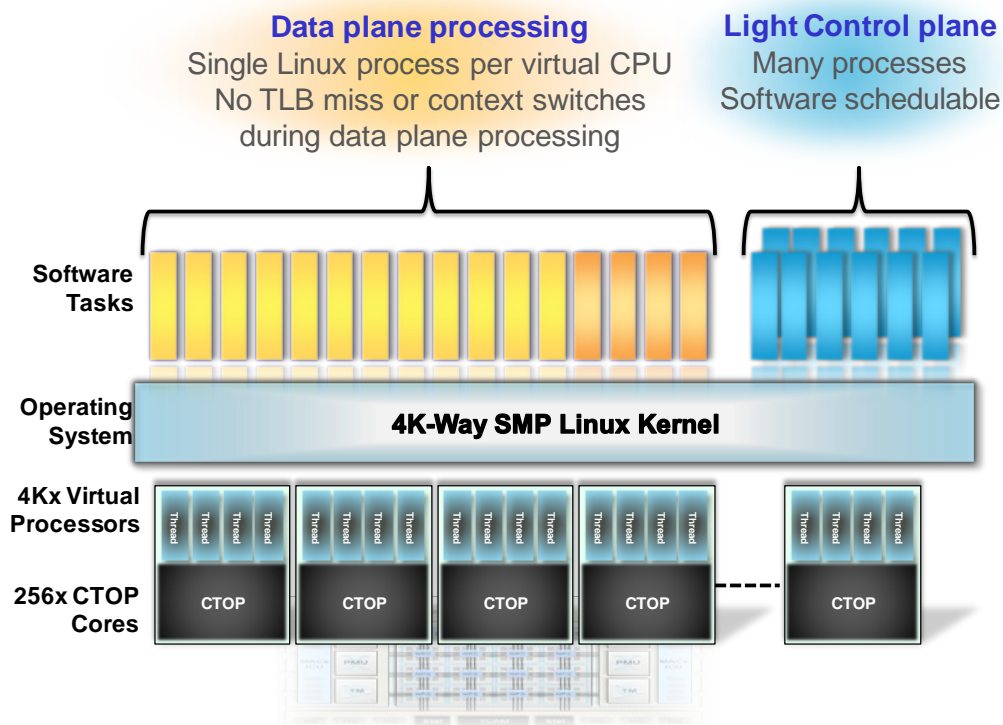
## 1.3 Zero Overhead Linux

Part of the software stack provided with NPS-400 is the Zero Overhead Linux (aka ZOL). Zero Overhead Linux is an enhanced operating mode of the Linux<sup>®</sup> based operating system provided by EZchip with the NPS-400, which makes use of the unique hardware features of the NPS to provide the ability to run a standard Linux user space process performing data plane packet processing on the NPS with performance on par with a bare metal application running without an OS.

Zero Overhead Linux provides the unique ability to enjoy the best of both worlds:

To develop packet processing applications as regular C based Linux applications while making use of all the standard tools and features provided the Linux OS for application development while still meeting the stringent performance requirements that are expected from a data plane packet processing application.

**Figure 2. NPS-400 Zero Overhead Linux**



### 1.3.1 How does one use Zero Overhead Linux?

Under Zero Overhead Linux, data plane packet processing code is written as a standard Linux user space application in C, which runs on multiple cores in parallel. One such process is assigned to each hardware thread of the NPS CTOP cores.

An EZchip-provided code library configures the operating system and hardware in such a way that only explicit end-actions (such as performing a system call or attaching to the process with a debugger) will take control away from the data plane application running on the processor.

At all other times, the data plane packet processing code has uninterrupted access to the processor, providing performance on par with a bare metal application running without an OS.



## 1.3.2 How does Zero Overhead Linux work?

Zero Overhead Linux works by mitigating the possible sources of interference, or interruptions, that are normally associated with running a user space process on an OS such as Linux, from either the hardware or the software, during execution of performance-sensitive packet processing code.

### 1.3.2.1 *The Fixed Mapping Table*

One of the obstacles that has traditionally affected the performance of packet processing code under a general purpose OS is the need for the OS to manage virtual memory, thus exposing the packet processing code to the risk of a TLB (Translation Look-aside Buffer) miss event.

The assignment of packet processing critical code and data sections to FMT slots, or pages, is done by the programmer using specialized compiler pragmas.

The packet processing code and associated data of applications running in Zero Overhead Linux are mapped from virtual to physical memory using the Fixed Mapping Table mechanism rather than the TLB mechanism. Thus, TLB misses will never occur for packet processing code and the associated data under Zero Overhead Linux.

### 1.3.2.2 *Isolating the data plane application from OS interference*

In order to provide the data plane application uninterrupted access to the processor, Zero Overhead Linux mode fine tunes the Linux OS to:

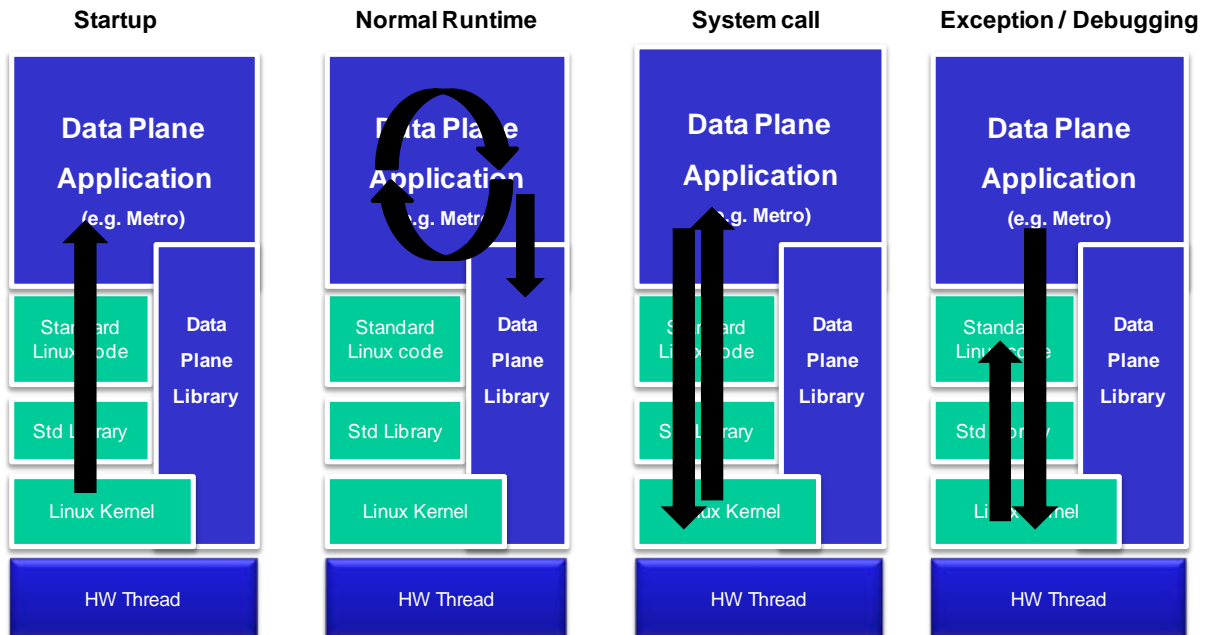
- Schedule a single data plane task per virtual CPU in Zero Overhead Linux mode.
- Turn off OS services that may interrupt the data plane application on the virtual CPU in Zero Overhead Linux mode, such as the timer interrupt or memory statistics services.
- Set up the OS such that other virtual CPUs will not have a cause to interrupt a virtual CPU running in Zero Overhead Linux mode via cross-CPU interrupts without a specific programmer action.

Thanks to the above configurations, the Linux kernel will not involuntarily interrupt the packet processing application running under Zero Overhead Linux.

### 1.3.3 Data Plane Application Life Cycle

The [Figure 3](#) below illustrates the data plane application life cycle and it is described below.

**Figure 3. Data plane application life cycle**



#### 1.3.3.1 Startup

The Linux kernel starts up and runs the Linux user space application in the normal fashion. All setup and initialization phases can be performed by the application at this phase. Any Linux system and library call can be used in this phase.

As part of this phase, performance critical packet processing code and data sections are mapped using the fixed mapping table (FMT) to memory spaces, mostly in the internal memory (IMEM).

Finally, the stack is relocated to a region mapped via the FMT to IMEM and Zero Overhead Linux mode is engaged, and control is transferred to the main packet processing event loop.

#### 1.3.3.2 Packet Processing

Once data plane mode has been engaged and we are running in the packet processing main loop (normal runtime in [Figure 3](#) above), no software or hardware interruptions will occur as long as the code execution is limited to data plane API library calls and computational code.

The data plane library API calls provide the interface to control the hardware packet processing engines and accelerators.

#### 1.3.3.3 System Call

At any point during a packet processing event loop, the programmer may call any library outside of the data plane API library including any Linux system call, such as printing to a log file or standard output via `printf()`, thus providing the ability to perform either debugging or non-performance-critical slow path exception processing.

During the execution of said library and system calls, data plane performance is not guaranteed for the calling hardware thread.

When the library or system call processing has finished, data plane mode will automatically re-engage. Care should be taken however, as residual processing may be required for some system or library calls,

such as long lived timers, which may affect data plane performance even some time after the library or system call has returned.

#### 1.3.3.4 Exceptions and Debugging

In addition to programmer initiated data plane interruptions in the form of system calls, normal Linux exception handling, such as signal delivery due to out of bound memory access or divide by zero, will interrupt the data plane application as expected, allowing the implementation of exception handling in the normal way.

In addition, break and watch points placed via the debugger will also interrupt the data plane application.

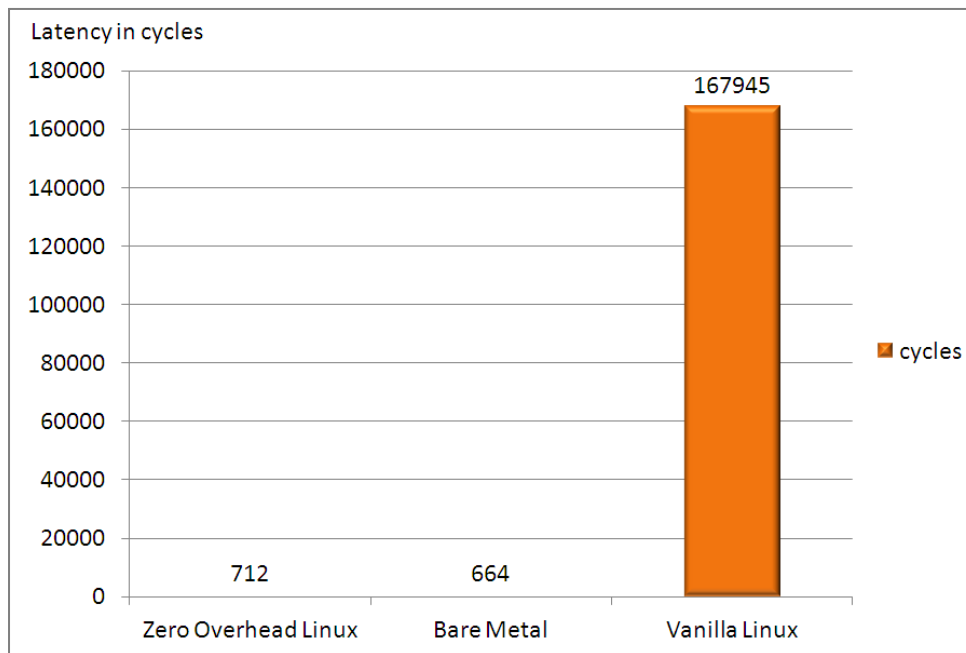
### 1.3.4 Performance Provided by Zero Overhead Linux

The performance provided by a data plane application running under Linux in Zero Overhead Linux mode is the same performance as one would expect from the same code running in a bare metal environment without an OS.

The following graph (Figure 4) depicts the measurements of the maximum number of cycles a simple load-increment-store code sequence from uncached RAM (e.g. C language “i++;” instruction) takes, in three environment configurations:

- **Bare metal** – A bare metal application running without an OS.
- **Vanilla Linux** – Linux user space application.
- **Zero Overhead Linux** – Linux user space application running in ZOL mode.

Figure 4. Zero Overhead Linux performance for 3 environment configurations (less is better)



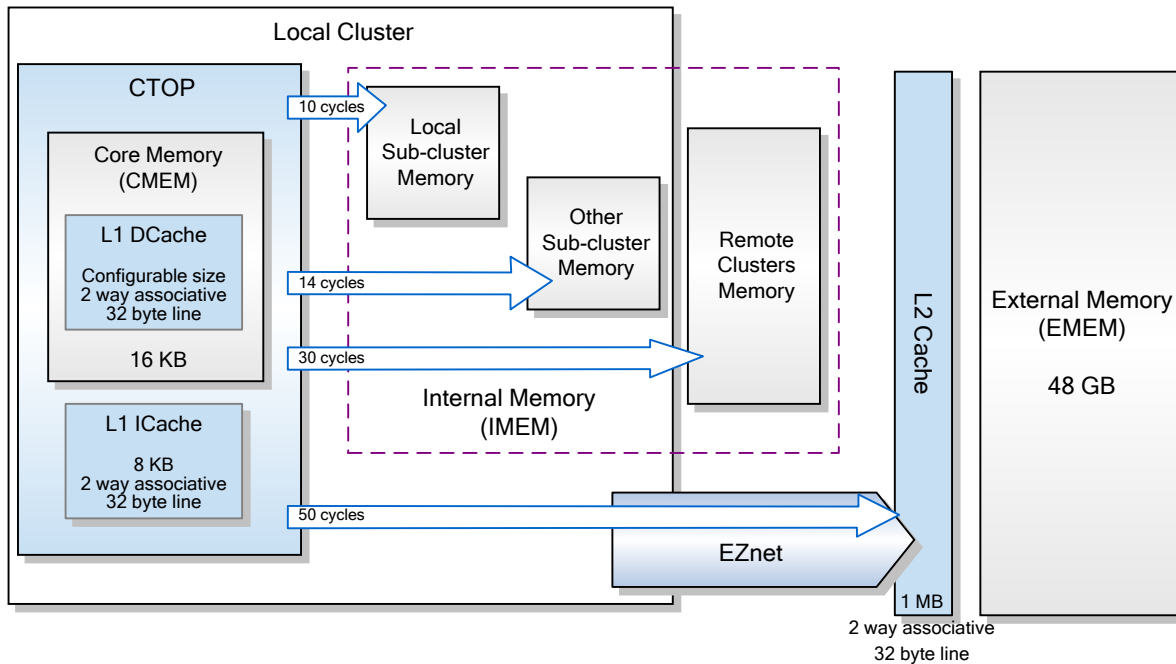
As can be seen by the above graph, the performance of code running a Linux user space application under Zero Overhead Linux is virtually identical to the same code running on bare metal.

The high number of cycles in the Vanilla Linux measurement for such a small code snippet is due to interruptions from the OS, such as a timer interrupts. These interruptions do not exist when running either in bare metal or under Linux in Zero Overhead Linux mode.

## 1.4 Memory Hierarchy

The NPS-400 allows the programmer to make use of up to 4 TB of memory storage space that spans multiple memory banks of several types spread out across the chip and SDRAM. The NPS memory hierarchy is designed to allow the programmer to make efficient and secure use of these resources in a simple fashion. The proper use of the memory by the data path code is often the key to getting the most, performance wise, of the NPS. The following sections describe the main concepts and terms of this memory hierarchy.

**Figure 5. NPS-400 memory diagram**



### 1.4.1 Memory Types

The NPS supports three different types of memories: external memory, internal memory, core's local memory. The following sections describe each of these types, their sizes, typical usage, and access latencies.

#### 1.4.1.1 External Memory (EMEM)

The external memory, henceforth referred to as EMEM, includes most of the memory available to the NPS, up to 48 GB of RAM. This memory is typically used to store the bulk of the frame data, the search tables and is where non-performance-critical application code and data (including the Linux OS) reside.

As the name implies, the external memory contains memory which is external to the NPS. It is comprised of several banks of DRAM memory. The access latency to EMEM ranges from 50 cycles, with a hit in L2 cache, to 500 cycles on SDRAM access.

The EMEM can be used to store program code and data as well frame buffers and search tables.

The EMEM is made available to software through the mapping of it to the core virtual address space via the CTOP memory management unit and by various accelerators, e.g. DMA, lookup, etc.

For maximum performance, it is recommended that access to data in EMEM in the high speed frame processing code path will be performed via one of the accelerator engines and not via load/store operations.

### 1.4.1.2 Internal Memory (IMEM)

The internal memory, henceforth referred to as IMEM, is 16 MB of SRAM which is spread throughout the chip, providing low latency access times. This memory is typically used to store performance-critical code and data, as well as small search tables and frame header data.

Each NPC cluster integrates 1MB of IMEM referred to as LMEM (local memory). The NPC's IMEM is made of two 512KB sub-clusters.

CTOP access to the IMEM bank in its local cluster uses only the cluster's internal crossbar. CTOP access to the IMEM banks in neighboring clusters is routed via the EZnet hierarchical mesh crossbar.

The access latency to the local IMEM bank ranges between 10 cycles, for accesses to the local sub-cluster bank, and up to 14 cycles, for accesses to the adjacent sub-cluster bank. The access latency to IMEM banks in remote clusters can reach up to 30 cycles.

The IMEM can be used to store performance critical program code and data, as well as frame buffers and small search structures.

The IMEM is made available to software by mapping it to the core virtual address space via the core Fixed Mapping Table mechanism, which is done automatically by the supplied EZdp library, and by various accelerators, e.g. DMA, lookup, etc.

For maximum performance, it is recommended that critical code and data will be placed in IMEM.

### 1.4.1.3 Core's Local Memory (CMEM)

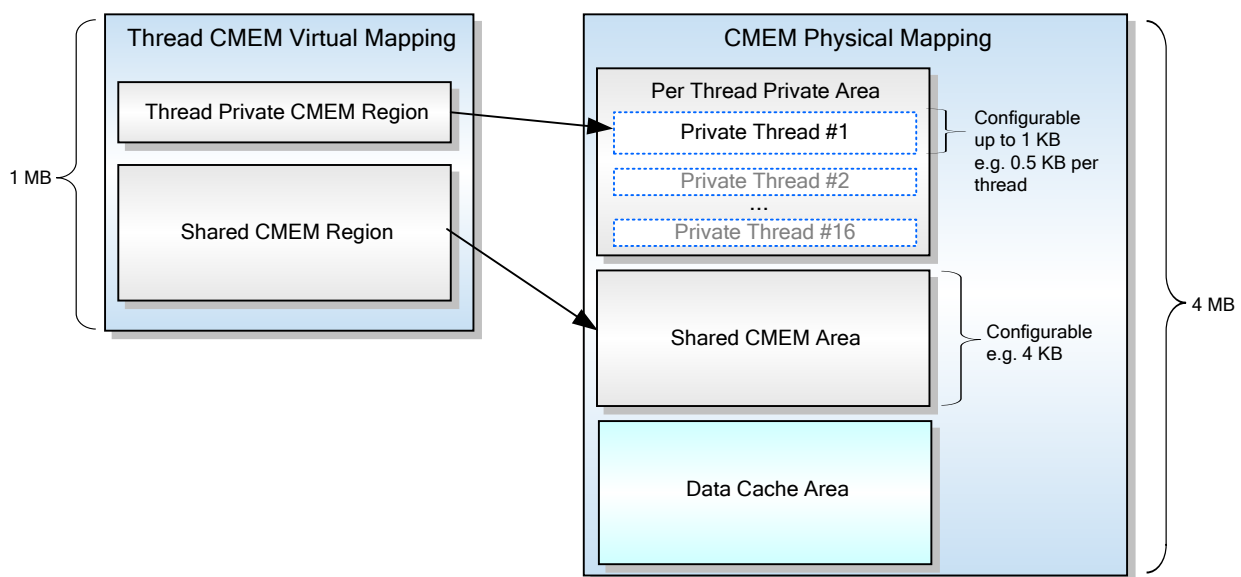
The core's local memory, henceforth referred to as CMEM, is a 16 KB bank of SRAM which is part of each CTOP core. This memory is used to store performance-critical data pertaining to the current frame being processed, such as frame headers, search keys and results, and temporary data.

Each of the CTOPs has its own CMEM bank. The CMEM is shared by 16 threads of the core. However, unlike the IMEM, the CMEM of other cores (whether they are in the local cluster or not) is not accessible to any CTOP core except itself.

Of all the available memory types the CMEM has the lowest latency at 1-2 cycles.

During runtime the CMEM is partitioned into three separate areas ([Figure 6](#)): the thread private area, core shared area and the L1 data cache storage area.

**Figure 6. CMEM virtual and physical views**



#### **1.4.1.3.1 Thread Private CMEM Area**

The thread private area of the CMEM is automatically segmented by the core hardware into a per thread private memory area. Separation and protection between the different thread segments is enforced by hardware. The software running on each hardware thread can use this area to store task private variables and data structures.

#### **1.4.1.3.2 Core Shared CMEM Area**

As the name implies, the core shared CMEM area is jointly shared and managed by the various hardware threads of the core. This area can be used as a shared memory between the tasks running on the different cores or as a shared heap.

As with any shared memory, care should be taken when multiple tasks access this memory to ensure data structure coherence in the face of pseudo concurrent accesses by using synchronization techniques, such as locks.

#### **1.4.1.3.3 L1 Data Cache Area**

Part of the CMEM memory is configured to be used as a backing store for the core L1 data cache. Once configured, this memory is used internally by the core hardware and is invisible to the programmer.

##### **1.4.1.3.3.1 L1 Instruction Cache**

The L1 instruction cache is 8 KB in size, 2-way associative with a 32 byte cache line. The NPS has an instruction cache per core which is shared between all threads in that core.

##### **1.4.1.3.3.2 L1 Data Cache**

The L1 data cache is write-back, 2-way associative with a 32 byte cache line.

The RAM used for the cache is part of the core's local memory (CMEM), and size-wise it is configurable in a power of two (0 KB, 1 KB, 2 KB, 4 KB ... 16 KB). The CMEM memory that is not used by the L1 data cache is available as core RAM for software usage.

The L1 data cache is equally split between the hardware threads of the given CTOP core.

There is no hardware cache coherency between the L1 data cache of different cores. Hence memory regions shared between different cores must be either be accessed non-cached or, alternately, the cache coherency would need to be managed in software.

## 1.4.2 Memory Space Configuration

In order to ensure top performance, the NPS allows fine grained control of the allocation and use of the NPS globally accessible memories. This control is facilitated by the concept of memory spaces.

A memory space is an area of memory in IMEM or EMEM accessible by the NPS which is managed as a single logical unit. Each memory space has a set of configurable properties associated with it which govern the usage of the memory space as a whole.

The NPS supports two types of memory spaces: internal and external.

While not all types of memory spaces support configuration of all the properties, the list of properties includes:

- The memory area size
- The type of memory of which it is comprised – IMEM or EMEM
- The underlying memory banks distribution and scrambling method to load balance the different memory banks.
- Whether access to the memory space is cache or not and the distribution of the L2 cache instance used.
- The protection scheme used to protect the information in the memory space: in-band ECC, out-of-band ECC.
- Access permissions for the NPS CTOP cores and hardware engines.

### 1.4.2.1 External Memory Spaces

External memory spaces are memory areas in EMEM which are used to define memory areas in DDR memory.

EMEM memory spaces may be configured via the control plane library API calls to select size, bank distribution and scrambling, L2 cache instance distribution, protection scheme, replications and hardware engines and core access permissions.

External memory space configuration is SoC global. Each NPC cluster may map a configurable subset of the global memory spaces.

### 1.4.2.2 Internal Memory Spaces

Internal memory spaces are memory areas in the internal SRAM banks inside the NPS network processing clusters.

Internal memory spaces may be configured via the control plane library API calls to select size, bank distribution, replications and permissions.

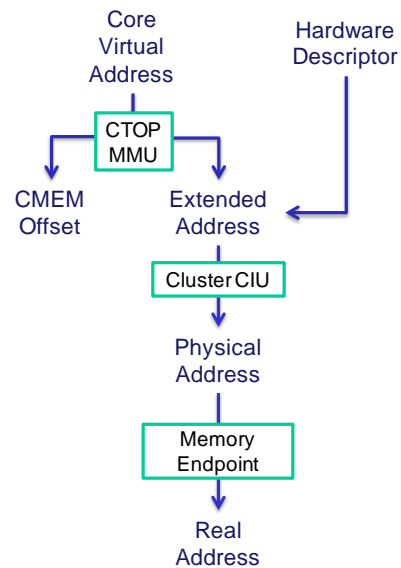
Internal memory space configuration is per cluster.

### 1.4.3 Address Translation

The NPS supports a four-level hierarchy of memory management translations:

- CTOP core virtual addresses and hardware descriptors, which are translated by the CTOP core into either an NPC cluster extended address or a CMEM offset.
- NPC cluster extended addresses, which are translated by the Cluster Interface Unit (CIU) to a physical address.
- Physical addresses, which are carried by the NPS EZnet global crossbar and are translated by the memory endpoint to a real address
- Real addresses, which are addresses on a bus, DDR memory or PCIe.

**Figure 7. Memory Translation**



#### 1.4.3.1 Core Level Virtual Memory Translation

The core level virtual to physical memory translation is responsible for translating a CTOP core virtual address into a CTOP physical address.

This translation is accomplished using two complementing mechanisms: the core translation look aside buffer, henceforth referred to as TLB, and the CTOP fixed mapping table, henceforth referred to as FMT.

#### 1.4.3.2 The Translation Look Aside Buffers

The TLB is a hardware mechanism, part of a core's memory management unit, which caches translations between virtual and extended pages. When the translation for a certain virtual page cannot be found within the TLB cache, a TLB miss event exception occurs and the operating system reprograms the TLB registers with the needed translation information.

#### 1.4.3.3 The Fixed Mapping Table

The NPS extends the traditional TLB mechanism with the Fixed Mapping Table (FMT). The Fixed Mapping Table is a dedicated virtual to extended address translation table, comprised of 16 entries, each covering an 8MB size page, through which access to latency sensitive code and data are mapped into physical pages. Each FMT slot can be configured to map virtual memory to an NPS internal memory space.

Once configured, the FMT settings controlling the address translation are kept entirely in hardware registers, thus guaranteeing a TLB miss is never needed for code and data mapped using this mechanism.



## 1.5 Multi-threading

The CTOP core supports hardware multi-threading. This feature allows up to 16 hardware threads to timeshare a single CTOP execution unit. Each hardware thread appears to the software as a separate logical core which behaves as a separate CPU with its own independent state, including registers, current privilege level, interrupts and exception levels, and so on.

At any given time, only one thread executes instructions on the core while the other threads are dormant; when being dormant, threads may advance their load, store, fetch and accelerate engine transactions, but do not execute instructions on the CPU pipeline or advance their program counters.

Threads are scheduled by the CTOP hardware in a transparent seamless fashion to the programmer. The multi-threading facility allows the CTOP to keep the core execution unit fully utilized and hide memory access and off-core accelerator processing latency. When a thread cannot make further progress while a transaction is in progress, it is put into a dormant state and another hardware threads can utilize the execution unit.

Having said that, the programmer may provide software based hints to guarantee optimal execution unit utilization.



The hardware threads discussed in this section should not be confused with the software threads, or tasks, managed by an operating system running on top of the core. Although many of the terms are the same (e.g. thread, context switch, schedule), hardware multi-threading is controlled and managed by the hardware and not the operating system. To the operating system, each of the 16 different hardware threads appears as a distinct and separate CPU, on which the OS might schedule multiple software threads in the usual manner.

## 1.5.1 Thread States

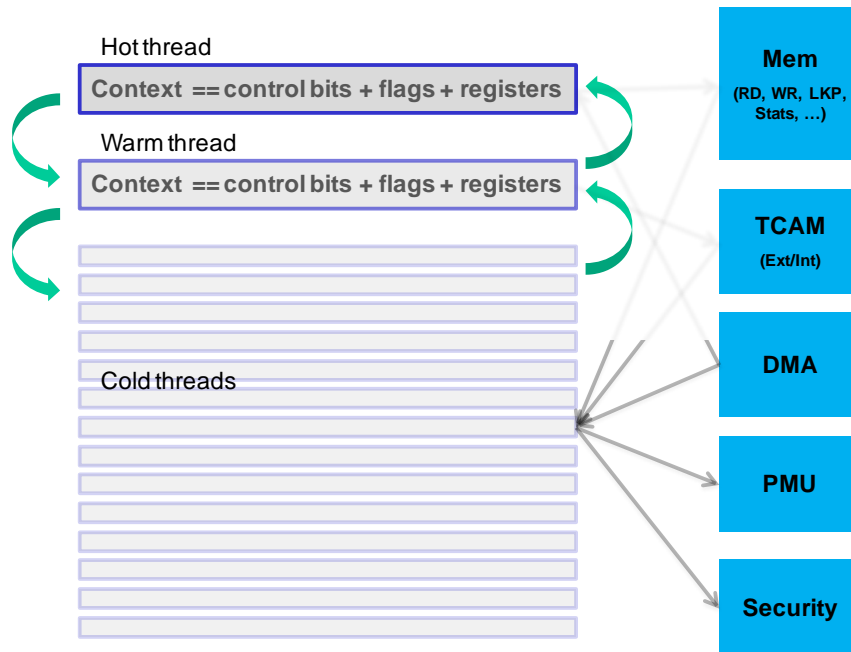
A thread can be in one of three states ([Figure 8](#)):

The **hot** thread is the current active thread utilizing the execution unit.

The **warm** thread is either the previous or the next thread that will become the active thread at the next context switch.

The **cold** threads are the rest of the hardware threads of the core, which are dormant: neither currently active nor about to become active in the next context switch.

**Figure 8. NPS-400 thread states**



## 1.5.2 Thread Scheduling

The CTOP Multi-Threading Management unit (MTM) is responsible for tracking the eligibility of the various threads to gain access to the execution unit. It is also the entity selecting the next thread to be designated as the warm thread; that is, the thread that will be context switched into the execution unit at the next scheduling event.

### 1.5.2.1 Scheduling Events

Scheduling events are events which cause the core to consider switching the warm and hot threads and in effect, to context switch a new hardware thread into the execution unit, if an eligible thread is available.

NPS supports these scheduling events:

- **Pipeline utilization watchdog event** – specific events in the execution unit processing which are inductive of an underutilized execution unit pipeline, such as an instruction cache miss or a long pipeline stall, also trigger an automatic scheduling event.
- **Implicit EZdp library scheduling**– an explicit notification from software via a dedicated CTOP ISA instruction indicating to the core that a scheduling decision is requested.
- **Explicit application scheduling events** – an explicit notification from software via a dedicated CTOP ISA instruction indicating to the core that a scheduling decision is requested.
- **Async** – programmer should explicitly call to schedule/ezdp\_sync.
- **Sync (default)** – EZdp library call to schedule/sync.

- **Thread cycle counter watchdog timer event** – a cycle counter tracks the number of cycles the hot thread has executed since the last context switch and triggers a scheduling event once a preconfigured cycle count has elapsed, in order to guarantee fairness between all eligible threads.

During a scheduling event, the CTOP core will attempt to switch the hot and warm thread contexts, in effect, bringing in a new thread to run and retiring the previous running thread. Note that a scheduling attempt may not be successful – for example if only a single thread is eligible for execution at a given time. In such a case, the current hot thread will continue to occupy the execution unit, possibly stalling the CTOP pipeline until it can safely proceed.

In addition, at every scheduling event, the CTOP core Multi-Threading Management (MTM) unit will choose the next thread to be context switched at the next scheduling event, switching the warm thread and a cold thread context if need be, thus preparing the context of the next thread to run. This process can take up to 7 cycles to complete, however, it is performed in parallel to the hot thread utilizing the execution unit, so that thread context switch time is a single cycle.

The next thread to be scheduled is chosen by the MTM according to Round Robin algorithm from all eligible threads.

Thread eligibility for scheduling is determined according to the state and nature of pending transactions each thread has, the eligibility criteria which was indicated when the thread was scheduled out and asynchronous hardware messages such as interrupts as discussed below.

### 1.5.2.2 Pending Transactions

The CTOP Multi-Threading Management unit (MTM) keeps score of all outstanding transactions performed by each of the threads; a transaction is any interaction of the core with resources outside of the core, such as memory access for either data load or store, instruction fetch or accelerator invocation such as DMA, lookups and crypto engines.

There are three types of pending transactions:

- **Write transactions** – write transactions are close ended posted transactions. These transactions are guaranteed to finish in a finite time and no CTOP instruction is dependent on their result. Memory store operations, or asynchronous accelerator operations that do not return a state in CTOP registers, are all write transactions.
- **Read transactions** – read transactions are close ended non-posted transactions. These transactions are guaranteed to finish in a finite time but some future CTOP instruction may depend on the completion of the transaction in order to proceed. Memory load operations, instruction fetching and synchronous accelerator invocations that return a result in CTOP registers are all read transactions.
- **Externally controlled transactions** – externally controlled transactions are open ended transactions whose completion is dependent on an entity outside of the CTOP core; therefore, these transactions are not guaranteed to complete in any finite time frame. Requesting a new frame to process is an externally controlled transaction.

Any thread may open up to four concurrent pending transactions of any nature at a given time. Attempting to open more pending transactions will cause the thread to stall.

### 1.5.2.3 Interrupts

In addition to the pending transactions which are described above, the MTM unit also tracks asynchronous hardware messages. As the name implies asynchronous hardware messages are hardware-based notification events which occur outside of the control of the core.

Existence of a pending interrupt is taken into account in the scheduling eligibility criteria as described in the next section.

### 1.5.2.4 Eligibility Criteria

When a scheduling event occurs and a thread is switched, eligibility criteria are used by the MTM unit to determine when the thread will be considered eligible again for scheduling.

Three possible criteria for eligibility:

- **Read transactions synchronization** – the hardware thread will be marked eligible again when all pending read transactions are completed. Interrupt handling is delayed until the thread is eligible again.
- **Read/Write transactions synchronization** – the hardware thread will be marked eligible for scheduling again when all pending read and write transactions are completed. Interrupt handling is delayed until the thread is eligible again.
- **Wait for trigger synchronization** – the hardware thread will be marked eligible when all read and write transactions are completed and either an external controlled transaction have completed or an interrupt occurred.

## 1.6 Network Interfaces

### 1.6.1 Engines and Interfaces

The NPS is equipped with multiple network interface engines that reside on the two sides of the chip.

Each network interface engine can be configured to operate as multiple network interfaces of various rates (from 10 Gbps to 400 Gbps) of Ethernet or Interlaken. A network interface is a single physical PHY.

### 1.6.2 Input and Output Channels and Queues

Network interfaces are further segmented into channels. A channel is a logical segment of a physical network interface resource that maintains independent flow control state with its peer. An Interlaken channel is an example of a network interface channel.

The input channels and the output channels of a network interface are orthogonal. It is possible for a network interface to have a single input channel and multiple output channels.

Each output channel is serviced by a four level priority queue. The priority of the frames queued for transmission is determined according to the class of service assigned to the frame by the data plane software.

### 1.6.3 Logical Channels

In addition to the output channels servicing network interfaces, the NPS is also equipped with logical output channels that provide hardware accelerated services for frame loopback, replication and discard.

### 1.6.4 Traffic Manager

NPS data plane software may request a frame to be queued directly to an output channel of a network interface with only rudimentary fairness provided by the output priority queue.

In most real life applications however, the data plane software will instead request a frame it wishes to transmit to be handled by one of the NPS traffic management hardware engines.

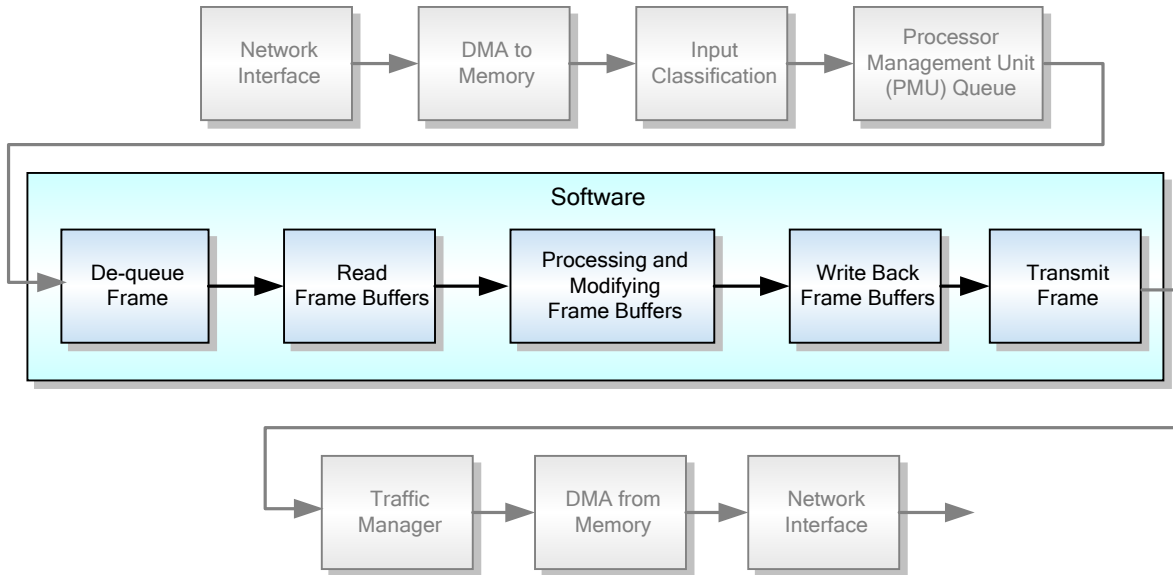
In this case, rather than designating the output channel and interface directly, the data plane software assigns the frame a TM flow ID and additional information which is then used by the traffic manager according to its configuration to select the output channel and interface using the traffic manager packet switch table.

This indirection allows control plane software to easily provision sophisticated services such as port fail over and load balancing in a coordinated fashion without needlessly complicating the data plane software.

## 1.7 Frame Flow

The following figure is an outline of the typical flow a frame takes from entering the chip from one network interface until it is transmitted via another:

**Figure 9. Typical frame flow**



The frame traffic is received on the RxIFU/MAC and is parsed and classified by the ICU.

The RxNDMA allocates frame buffers, via the BMU, in IMEM/EMEM and forwards the frame descriptor (FD) to the PMU.

The PMU allocates a Job Descriptor (JD) for frame processing and queues the job to an internally managed physical queue (PQ), chosen based on the Input Classification Unit (ICU) configuration.

A data plane application process running on a CTOP thread indicates it is ready for a new job and schedules out of the execution unit to wait for the job arrival.

The PMU dispatches a job message that schedules the thread back to processing and indicates a job descriptor identifier.

The application copies (via DMA) the Job Descriptor to the CMEM.

The application extracts the frame buffer descriptors from the Frame Descriptor embedded inside the Job Descriptor.

Using the frame buffer descriptors, the application copies (via DMA) the frame header from the frame buffer to the CMEM.

The application parses the frame header and decodes network protocols via the Protocol Decoder accelerator.

The application performs data plane frame processing sequence of structure lookups, traffic rate policers and statistic updates. The remote structure accesses are accelerated by hardware mechanisms.

The application modifies the frame header in the CMEM.

The application copies (via DMA) the modified header from the CMEM to the frame buffer.

The application updates the job descriptor with its verdict for the next stage of processing for the frame, such as transmission via a TM queue, dispatching to a different PMU queue or discarding.

The application copies (via DMA) the modified job descriptor from the CMEM to the job descriptor.

The application sends the job to the PMU for further processing.

The PMU re-establishes the network frame order by sending the frame to the next processing stage in the same order that the frame arrived in its queue – either to another queue in the PMU, to the TM for transmission or loopback.

The TM scheduled frame is queued to the output channel selected by the TM configuration of either a network interface, from which the TxNDMA reads the frame data and transmits the frame via the TxIFU/MAC, or to a logical output channel for loopback, replication or discarding.

## 2. NPS Control Plane / Data Plane Overview

TBD



## 3. First Steps

In this chapter we will guide you step by step to building your first data plane application, describing the basic steps and details needed to write a new data plane processing application. The discussed hello packet application and code are installed automatically as part of the EZide installation; readers of this chapter are encouraged to experiment with the sample code while following this chapter.

### 3.1 Basic Guidelines

By its nature this chapter of this manual touches a wide set of system capabilities affecting data plane applications. Many of these will be explained at length in the following chapters.

In the meantime the following few keywords used require specific understanding in order to understand the text:

- **CPU** – Where used the term *CPU* comes to denote a hardware thread as discussed in the [Multi-threading](#) section.
- **System Memory** – The term *system memory* refers to memory provided by the SW and used by the various HW engines.
- **Configuration** – Any mention of *configuration* refers to the NPS control plane configuration done by the host driver. For further information refer to the *EZcp Reference Manual*.

### 3.2 Hello Data Plane: frame in, frame out

We shall start with a very simple program that takes a frame from one physical port and forwards it to another. We shall go over the program in detail and explain all the different parts and then we shall evolve it over the course of this manual, as we learn about the different capabilities of the NPS and how to make use of them via the data plane API.

Here is a listing of the program:

**Figure 10. Hello data plane - data plane application skeleton example**

```
#include <stdio.h>
#include <stdlib.h>
#include <ezdp.h>

/* A job ID, allocated from core memory */
static ezdp_job_id_t job __cmem_var;

/* A job descriptor, allocated from core memory */
static struct ezdp_job_desc job_desc __cmem_var;

/* Packet processing function located in per cluster IMEM */
void packet_processing(void) __imem_1_cluster_func;
void packet_processing(void)
{
    while (true) {
        /* Receive a new job */
        ezdp_receive_job(&job, &job_desc, 0);

        /* Transmit job via the channel 1 interface of side 0 */
        ezdp_send_job_to_interface(&job, &job_desc, 0, 1, 0);
    }
}

int main(void)
{
```

```

/* Wait until control plane is up and configure the chip */
if(ezdp_sync_cp() != 0){
    return EXIT_FAILURE;
}

/* Initialize the data plane application 0 */
if(ezdp_init_global(0) != 0){
    return EXIT_FAILURE;
}

/* Initialize the data plane hardware thread and bind this process to
cpu 16 */
if(ezdp_init_local(0, 16, NULL, 0, 0) != 0) {
    return EXIT_FAILURE;
}
/* Call to packet processing function */
ezdp_run(&packet_processing, 0);

return EXIT_SUCCESS;
}

```

### 3.2.1 The Bare Necessities: a data plane application program skeleton

As you can see it is a standard Linux program written in C, with a few special library calls and macros which will be explained shortly.

Let us now go over the program and explore its structure in detail.

#### 3.2.1.1 Include files

As can be seen in the listing above ([Figure 10](#)), in addition to several standard C library include files, we also include the *ezdp.h* include file. This include file contains definitions that are required for making use of the EZdp data plane API and is the only include file that is needed to make use of the API routines.

#### 3.2.1.2 Code and data section assignments

As described in the [Zero Overhead Linux](#) section above, the NPS supports a three-level hierarchical memory system. The EZdp API defines qualifiers that can be used to allocate a given variable or function in a specific memory type. Examples of qualifier usage can be seen in our example program listing above. These qualifiers instruct the linker to place the variable or function in a specific ELF section of the binary, which are loaded to the requested memory region (via struct *ezdp\_cpu\_cfg*) at run time.

##### 3.2.1.2.1 Placing private variables in the core's private local memory

To allocate a variable in the private CMEM memory region, append the qualifier *\_\_cmem\_var* to the variable definition.

For example, here is how we allocated the *job* variable in the CMEM in our example program above:

```
static struct ezdp_job_id job __cmem_var;
```

A variable allocated in this fashion will be placed at run time in the low latency CMEM memory of the processor in the area private to the specific hardware thread, thus guaranteeing private access a close to zero cycle access latency to this information.

Because of the limited size of the CMEM, place only the most frequently accessed global data structures of your program in this region.

Yet another reason to place data in the CMEM is that some of the hardware acceleration blocks can only access the CMEM memory. For this reason, some of the parameters provided to particular EZdp API calls require an address in the CMEM. This fact is indicated by the `__cmem` prefix to the pointer parameter. Failing to use CMEM parameters where required would result in a run time error, unless when compiled in debug mode which would throw an assertion.

This is a good region to place job descriptors, frame data work buffers, search keys and result structures. We will describe each of these data structures in the sections ahead.

### 3.2.1.2.2 Placing shared variables in the core's local memory

To allocate a variable in the shared CMEM memory region, append the qualifier `__cmem_shared_var` to the variable definition.

Here is an example:

```
uint32_t shared_variable __cmem_shared_var;
```

A variable allocated in this fashion will be placed at run time in the shared section of the low latency CMEM memory of the processor, thus guaranteeing a close to zero cycle access latency to this information which is shared between all hardware threads belonging to the same core.

Placing variables which are later used as parameters provided to EZdp API calls which require an address in the CMEM in the shared CMEM section is allowed.

This is a good region to place search structure descriptors or any other data structures that can be safely shared between the different hardware threads of the same core. We will describe each search descriptor in the sections ahead.

---

**Warning!** Since the automatic hardware mechanisms may switch between the different hardware threads of the same core at any given time, the CMEM shared area must be treated as a shared memory region of your program; Concurrent update of data structures in this area should be protected using synchronization devices such as atomic operations or locks, even though only a single hardware thread is executed at each point in time.

---

### 3.2.1.2.3 Placing functions into a specific fixed mapping memory

In similar fashion, placing a code in a specific fixed mapping memory page is done by appending the `__imem_half_cluster_func`, `__imem_1_cluster_func`, `__imem_2_cluster_func`, `__imem_4_cluster_func`, `__imem_16_cluster_func`, `__imem_all_cluster_func` qualifiers to the function definition.

For example, the `frame_processing()` function from our data plane application skeleton example above ([Figure 10](#)), is defined in this fashion:

```
void packet_processing(void) __imem_1_cluster_func;
void packet_processing(void)
{
    ...
}
```

Placing a function in different code groups provides the ability to group together code based on its required performance. Group together critical fast-path code that is called very often together and slow path code such as exception handling or code for features that are rarely used.



**Tip!** Rather than hardcoding variable and function memory allocations by using the `__cmem` `__mem` macros directly it is often useful to use macros to disaggregate application policy from the specific resource used, as is shown in the following example:

```
#define __fast_path_code __mem_fmt_slot_func(3)
#define __slow_path_code __mem_fmt_slot_func(9)
```

#### 3.2.1.2.4 Placing variables into a specific fixed mapping memory

Just like code, variables may also be placed into one of the fixed mapping table pages. To do this, add the `__imem_private_var`, `__imem_half_cluster_var`, `__imem_1_cluster_var`, `__imem_2_cluster_var`, `__imem_4_cluster_var`, `__imem_16_cluster_var`, `__imem_all_cluster_var`, `__emem_var` qualifiers to the variable definition.


#### 3.2.1.3 Initialization

At the start of the data plane program, it is necessary to configure the CTOP core, load the variables and functions which were mapped to the CMEM and FMT sections, and bind the process to one dedicated processor. This is done by a call to the EZdp API function `ezdp_init_local()`.

```
uint32_t ezdp_init_local( uint32_t app_id, int32_t cpu_id, EZDP_MEM_CTOR_FUNC
mem_ctor_func, uintptr_t mem_ctor_data, uint32_t flags);
```

This function will set the ZOL system state in such a way so as to guarantee that your packet processing function will be run for its duration with minimal non-user-initiated interruption.

The parameters to the `ezdp_init_local()` call denote the application to which this processor belongs `app_id`, the processor to which the process will be bound `cpu_id`, pointer to memory constructor and its data `mem_ctor_func` and `mem_ctor_data` and `flags` which control the behavior of the init.

 For additional information about the `ezdp_init_local()` function and flags refer to the *EZdp Reference Manual*.



The configuration of all hardware threads on the same CTOP core must be identical.

Before calling `ezdp_init_local()` per process it is necessary to invoke the EZdp API function `ezdp_init_global()`. This function should be called once per data-plane executable and is used to allocate shared memory. This memory is used to synchronize the initialization of each process. In addition, before starting the initialization process there is the need to make sure that the control plane is up and configure the chip, so it is ready for run. This is done using the `ezdp_sync_cp()` EZdp API function.

Our simplistic “Hello data plane” program starts a single data plane process on a single processor. To expand the application to make use of multiple processors and cores, simply spawn a single Linux process for each processor and in each one call `ezdp_init_local()` with the requested processor ID, as the below example demonstrates:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <stdint.h>
#include <stdbool.h>
#include <signal.h>
#include <sys/prctl.h>
#include <ezdp.h>
```

```

#define CLUSTER_ID 1
#define NUM_CORES 16
#define NUM_THREADS 16

/* Packet processing code goes here*/
...

int main(int argc, char * argv[])
{
    uint8_t core;
    uint8_t thread;
    int32_t cpu;

    /* Wait until control plane is up and configure the chip */
    if(ezdp_sync_cp() != 0){
        return EXIT_FAILURE;
    }

    /* Initialize the data plane application 0 */
    If (ezdp_init_global(0) != 0){
        return EXIT_FAILURE;
    }

    for(core = 0; core < NUM_CORES; ++core) {
        for(thread = 0; thread < NUM_THREADS; ++thread) {
            cpu = ezdp_calc_cpu_id(CLUSTER_ID, core, thread);
            pid_t pid = fork();
            switch(pid) {

                case -1:
                    /* Error in fork, abort */
                    abort();
                    /* Not reached */

                case 0:
                    /* I'm in the parent - break to continue */
                    break;

                default:
                    /* I'm in the child - set things up and run */

                    /* Request to be terminated with my parent */
                    prctl(PR_SET_PDEATHSIG, SIGTERM);
                    ezdp_init_local(0, cpu, NULL, 0, 0);

                    /* All CPU specific initializations,
                     like search structure descriptors, go here */

                    ezdp_run(packet_processing, 0 );
                    return EXIT_SUCCESS;
                    /* Not reached */

            }
        }
    }

    /* Wait for all children */
    while ((waitpid(0, NULL, 0) != -1) || (errno == EINTR));

    return EXIT_SUCCESS;
}

```

### 3.2.1.4 Running frame processing code

We are now ready to start our frame processing code. To do so, call the `ezdp_run()` EZdp API function, passing with it a callback function with your main frame processing loop. The function you pass here will typically be allocated from the highest priority memory section (i.e. the one with the best performance), as it contains the code that is being run for every frame that is being processed by the application.

Before running your provided callback function, `ezdp_run()` will swap to use an alternative stack, which was allocated and mapped on one of the FTM pages.



Your callback function will be called using an alternate stack, which will be allocated from one of the FMT pages, rather than the stack which `main()` is using.

## 3.2.2 Working with Jobs

The NPS boasts a multi-core architecture that supports processing of many concurrent tasks on multiple processors in the system. Each such task, called a *job*, represents a single quantum of work to be performed by the NPS.

The NPS supports the following job types:

**Interface job** – a job containing a frame received from one of the NPS external network interfaces. This is the most common job type.

**Loopback job** – a job containing a frame which was created in one of the NPS internal hardware engines, such as one replica of a multicast frame.

**Confirmation job** – a job containing a confirmation message of the transmission of a single frame, such as those used in the IEEE 1588 network time synchronization protocol.

**Timer job** – a job signifying that a hardware timer has elapsed.

**User Information job** – a job containing a programmer-defined software message.

In order to track and schedule the different jobs being processed in the system at any given time, the NPS is equipped with a hardware block called the Processor Management Unit, or PMU. The PMU is responsible for orchestrating the assignment of jobs for processing to cores or other hardware processing blocks throughout the system, in a fair and prioritized manner.

### 3.2.2.1 The job ID

Each job in the system has a globally unique identifier called the Job Identifier, or job ID. A job ID is the token by which one references a single job.

It is important to note that a job ID is just a handle: an opaque number with no inherent meaning which is used to address a specific job in the system. Some of the operations done on a job by the programmer may change, or reassign, the job ID of a specific job mid processing implicitly; however, no programmer action is ever required to modify the job ID explicitly.

The data type `struct ezdp_job_id` is used for holding a job ID.

### 3.2.2.2 The job descriptor

Each job in the system also has a job descriptor (JD), in addition to the job ID.

A job descriptor is a data structure, maintained cooperatively by the Processor Management Unit and the frame processing software, which tracks the state of every job that is being processed by the NPS.

The data type `ezdp_job_desc` describes a job descriptor. Here are the relevant parts of the job descriptor type definitions as taken from `ezdp_job_defs.h`.

```

struct ezdp_job_desc {
    struct ezdp_frame_desc frame_desc;
    union {
        struct ezdp_job_rx_info rx_info;
        struct ezdp_job_tx_info tx_info;
    } __packed;
};

```

The job descriptor is divided into three parts: the [frame descriptor](#), [receive-side job information](#) and [transmit-side job information](#).

For efficiency reasons, the use of the receive information and transmit information are mutually exclusive as they are defined as fields in a C language union. Therefore care must be taken not to set the destination information until after the last read of the source information.

Let us go over the various parts of the job descriptor in more detail.

### 3.2.2.2.1 The job embedded frame descriptor

The frame descriptor is a data structure that holds information about the frame which is associated with the job, if any.

For jobs of type *interface* and *loopback*, the frame descriptor describes the frame to be processed.

For *confirmation* type jobs the frame descriptor describes the frame for which transition confirmation was received.

*Timer* type jobs are received with a NULL frame, since they are not associated with any specific frame.

Finally, the content of the frame descriptor structure for *user information* type jobs is user defined and is determined by the software that created the job.

As the frame descriptor structure is tightly coupled with the way frame data is arranged in the NPS memory, the full details about the frame descriptor structure will be provided in [The frame descriptor](#) section ahead.

#### 3.2.2.2.1.1 The frame logical ID

At this point in the course of our guide, we shall concern ourselves with just a single field of the frame descriptor structure – the *logical\_id* field.

```

struct ezdp_frame_desc {
    ...
    uint8_t logical_id;
    ...
};

```

The *logical\_id* field encodes the information of the kind of processing required by the software for this particular job.

The logical ID is assigned to each frame at its point of origin:

- Frames received through one of the NPS network interface get their logical ID assigned according to the configuration of the specific network interface.
- Frames generated by the timer mechanism have their logical ID determined by the configuration of the specific timer that generated them.
- Transmit confirmation and loopback frames have their logical ID set by the software when requesting these features.
- Finally, software may set the logical ID of frames processed by software.

It is important to note that there is no predetermined meaning to a specific logical ID and it is up to the system architect to assign application and system specific meaning to these IDs.



One common usage example includes assigning a different logical ID to each network interface to denote the origin port, another separate logical ID to each timer and a common logical ID to all loopback frames to denote multicast processing is needed. A variant of this scheme has all the network interfaces that belong to a single Link Aggregation Group assigned the same logical ID since they require the same treatment.

The single rule governing the assignment of logical IDs is that the software should be able to determine based on the logical ID what is the processing that should be applied to the specific frame, including if needed, separating the interface, timer, loopback, confirmation and software jobs apart.

#### 3.2.2.2.2 Receive-side job information structure

The receive-side job information structure provides additional information about the frame, the resources it uses and the conditions of the system as a whole, such as Ethernet and IP checksums, ordinal number of the frame in the processing queue, various congestion level notifications and so on.

We will describe the structure and the associated uses later on in the [Job Information](#) section ahead.

#### 3.2.2.2.3 Transmit-side job information structure

The transmit-side job information holds parameters which you may set prior to sending the frame to the Traffic Manager for processing that control the QoS enforced by the traffic manager on the frame, transmission statistics aggregation and transmission interface selection.

Information in this structure is only relevant for frames transmitted via the traffic manager mechanism and has no effect when other mechanisms of frame transmission are being used.

We will describe the structure and the traffic manager as a whole in the [Traffic Management](#) section ahead.

### 3.2.2.3 Getting a new job

The first operation in processing frames is requesting a new job from the Processor Management Unit.

#### 3.2.2.3.1 Receiving a job and a job descriptor

The simplest way to receive a new job is to call the `ezdp_receive_job()` API function:

```
ezdp_receive_job(struct ezdp_job_id * __cmem jobh_ptr, struct ezdp_job_desc * __cmem jd_ptr)
```

The `ezdp_receive_job()` function requests a new job assignment from the PMU, stores the job ID of the new job at the CMEM address provided by the `jobh_ptr` parameter and then copies the job descriptor from its location in IMEM into the CMEM at the address provided by the `jd_ptr` parameter.

Once the function returns, the job ID and the job descriptor are available in the CMEM at the supplied addresses and the frame processing can begin.

The `ezdp_receive_job()` is a compound high level function that combines several lower level operations that are commonly done together. These operations are:

- Requesting a job ID.
- Waiting for the job ID request to be fulfilled.
- Loading the job descriptor into the supplied buffer.

The EZdp API also provides low level functions that allow the programmer to fine tune his program by calling each of these operations separately.



### 3.2.2.3.1.1 Requesting a job ID

*ezdp\_request\_job\_id( struct ezdp\_job\_id \* \_\_cmem jobh\_ptr )*

The *ezdp\_request\_job\_id()* performs an asynchronous request for a new job from the PMU, providing the pointer to the CMEM where the job ID will be written to when it is available.

---

**Warning!** This function is asynchronous and returns immediately after the request has been sent. The buffer pointed to by the *job\_id\_ptr* parameter should not be used until the request is fulfilled. You must use the *ezdp\_wait\_for\_job\_id()* function to wait for the request fulfillment.

---

### 3.2.2.3.1.2 Waiting for a job ID request to be fulfilled

*ezdp\_wait\_for\_job\_id( void )*

The *ezdp\_wait\_for\_job\_id()* synchronizes with the job ID request from the PMU. The function is guaranteed to only return when a new job has been allocated from the PMU for the thread for processing and the job ID written to buffer whose address was provided in a previous call to *ezdp\_request\_job\_id()*.

### 3.2.2.3.1.3 Loading a job descriptor

*ezdp\_load\_job(struct ezdp\_job\_id \* \_\_cmem jobh\_ptr, struct ezdp\_jd \* \_\_cmem jd\_ptr)*

The *ezdp\_load\_job()* loads the job descriptor for the job identified by the job ID in the CMEM at address *jobh\_ptr* and copies it into the CMEM buffer pointed to by the *jd\_ptr* argument.

---

**Warning!** This call merely creates a local copy of the job descriptor in the CMEM. Any changes made to the job descriptor are only visible to your program and do not affect the system as a whole until the local copy of the job descriptor is copied back into the master copy.

---

## 3.2.2.4 Finalizing a job

After receiving a job and a job descriptor, we have all the information required to process the frame. We can use the information in the job descriptor and the frame data itself to determine what should be done with the frame, such as for example, through which network interface to transmit it.

For now, let us forgo the details of how to access the frame payload and modify it and focus our attention on what can be done with the frame once it is processed. The missing details will be provided in the next section, [Frame Headers and Payload](#).

There are three possible ways to end the processing of a frame: transmitting the frame, dropping the frame, or holding the frame.

### 3.2.2.4.1 Transmitting a frame

To transmit the frame through one of the system's network interfaces, one needs to send the frame to one of the transmit channels of one of the network interfaces, where it will be queued for transmission according to the frame's designated class of service and the channel's flow control status.

In most real world applications, the frame will not be sent to the channel directly. Instead the frame will be handed off to the Traffic Manager which provides virtual queuing, QoS enforcement and channel selection service (for example for implementing a Link Aggregation Group with a fast fail-over). We will discuss the Traffic Manager in depth in the [Traffic Management](#) section ahead.

For now however, we will focus on a simplistic application that requires no advanced traffic management services and wishes to assign the frame to a transmit channel of a network interface on its own accord.

This is done by calling the *ezdp\_send\_job\_to\_interface()* API function.

### 3.2.2.4.1.1 Storing a job descriptor and transmitting a frame

*ezdp\_send\_job\_to\_interface(struct ezdp\_job\_id \* \_\_cmem jobh\_ptr, struct ezdp\_job\_desc \* \_\_cmem job\_desc\_ptr, uint32\_t side, uint32\_t output\_channel, uint32\_t flags)*

The *jobh\_ptr* and *job\_desc\_ptr* arguments are pointers to CMEM buffers containing the job ID and job descriptor respectively, as received from a previous call to *ezdp\_receive\_job()* or underlying lower level APIs.

The *side* and *output\_channel* arguments together choose an output channel to be used for transmission. The frame will be queued for transmission via the network interface and the channel it was configured to use.

The argument *flags* can be used to control network order restoration behavior of the PMU for the frame: a value of zero requests the default behavior of transmitting the frame in the exact order it was received, while a value of EZDP\_ALLOW\_REORDER allows a more permissive behavior. For further discussion of frame ordering, see the [Order restoration](#) section ahead.

In a similar fashion as to what we have seen previously with the job receive API, the *ezdp\_send\_job\_to\_interface()* call is a high level compound function that performs two separate tasks:

- Synchronizing our local copy of the job descriptor in the CMEM with the master copy in IMEM.
- Notifying the Processor Management Unit that we wish to send the frame to the requested channel to be queued there for transmission.

Here too the EZdp API provides low level functions that allow the programmer to fine tune his program by performing each of these operations separately.

#### 3.2.2.4.1.1.1 Storing a job descriptor

*ezdp\_store\_job(struct ezdp\_job\_id \* \_\_cmem jobh\_ptr, struct ezdp\_job\_desc \* \_\_cmem jd\_ptr)*

The *ezdp\_store\_job()* function stores the job descriptor in the CMEM buffer pointed to by the *jobh\_ptr* argument, into the designated global location for that job descriptor in IMEM, based on the job ID indicated by the *job\_id\_ptr* argument.

This action, in effect, synchronizes the state of the master copy of the job descriptor in IMEM with the local copy of the job descriptor in the CMEM which was previously obtained by a matching call to *ezdp\_load\_job()*, updating the master copy with any changes that were made on the local copy and makes them globally visible in the system.

---

**Warning!** Without this call, any changes made to the local copy of the job descriptor will not be visible to the hardware engines of the system, such as the Processor Management Unit and the Traffic Manager.

---

#### 3.2.2.4.1.1.2 Returning a job to the Processor Management Unit

*ezdp\_send\_job\_id\_to\_interface( struct ezdp\_job\_id \* \_\_cmem jobh\_ptr, uint32\_t side, uint32\_t output\_channel, uint32\_t flags)*

The *ezdp\_send\_job\_id\_to\_interface()* function returns the job specified by the *jobh\_ptr* argument to the custody of the Processor Management Unit, indicating that the processing of the frame associated with this job has been completed and it should be sent to the specified *output\_channel* of *side* for queuing for transmission.

The argument *flags* can be used to control network order restoration behavior of the PMU for the frame: a value of zero requests the default behavior of transmitting the frame in the exact order it was received, while a value of EZDP\_ALLOW\_REORDER allows a more permissive behavior. For further discussion of frame ordering, see the [Order restoration](#) section ahead.

### 3.2.2.4.2 Dropping a frame

Transmitting a frame is the most common outcome of frame processing but it is not the only possible one. Sometimes we may wish to drop a frame instead. This is done via the `ezdp_discard_job()` API call:

**`ezdp_discard_job(struct ezdp_job_id * __cmem job_id_ptr, struct ezdp_job_desc * __cmem jd_ptr)`**

The `ezdp_discard_job()` function finalizes a job, freeing all associated resources, including the frame and the buffers it occupies. Calling `ezdp_discard_job()` indicates that the job processing of this frame has ended and that the frame is to be dropped permanently, releasing all its associated resources.

`ezdp_discard_job()` is actually a wrapper function with much of its functionality being done in an accelerated way using dedicated HW engines.

### 3.2.2.4.3 Advanced options

In addition to sending the frame via a channel associated with one of the NPS network interfaces or dropping the frame, the NPS supports the following additional job finalization options:

- Sending the frame to the Traffic Manager for virtual queuing, QoS and output channel selection services. We will discuss the Traffic Manager in depth in the [Traffic Management](#) section ahead.
- Forwarding the frame to a different PMU queue for further software processing in a pipeline fashion. Details will be provided in the [Controlling the Jobs Pipeline](#) section ahead.
- Sending the frame to the loopback interface, optionally requesting the frame to be replicated to implement efficient multicast. For discussion of this feature see [Frame Loopback](#) section.
- Sending the frame container and request the PMU to expose it, in order to implement lightweight multicast. For discussion of this feature see [Lightweight Multicast](#) section.
- Sending the frame and requesting that a transmit confirmation notification be generated. For more details see the section [Precision Time Protocol IEEE 1588](#) ahead.

## 3.3 Frame Headers and Payload

### 3.3.1 The Frame

A frame is a single Ethernet wire transmission that has been received by the NPS or is transmitted by it. The frame data excludes the Ethernet frame preamble, start of frame delimiter and inter-frame gap and may or may not include the 4 byte frame CRC, depending on the network port configuration. All other frame data, from the MAC destination to the frame payload, is kept in the system memory.

#### 3.3.1.1 Frame buffers

The frame data is kept in system memory as a series of 256 byte buffers; each buffer contains a portion of the frame data. Buffers may reside in either EMEM or IMEM.

Data in a frame buffer is protected by an ECC scheme which is stored outside the buffer itself and managed automatically by the hardware.

##### 3.3.1.1.1 Buffer management

The Buffer Management Unit hardware engine, or BMU, keeps tracks of all buffers in the system. The BMU maintains a set of buffer pools, according to type and configurable budget groups and uses these pools to service buffer allocation and de-allocation requests. These requests are generated by both software and hardware. For example by the network DMA hardware engine which transfers frames between the system memory and network interfaces.

The BMU maintains buffers in two pools: the internal buffer pool, whose buffers reside in IMEM, and the external buffer pool, whose buffers reside in EMEM.

The default allocation policy for frame data buffers is to allocate the first buffers of each frame from IMEM and the rest of the frame buffers, if any, from EMEM; the rationale behind this policy being that the first frame buffer usually holds the frame headers which are accessed more often than the frame payload. This policy may be modified, on a port by port basis, through the system configuration.

#### 3.3.1.2 Buffer descriptors

A buffer descriptor, or BD, is a data structure that describes a single frame buffer.

A buffer descriptor acts as a pointer to a buffer in memory; however, since the CTOP is a 32 bit core and the NPS supports more than 4Gb of RAM, it is not a simple pointer that may be accessed directly by load and store. Instead, special Direct Memory Access, or DMA, copy operations must be used in order to copy information to and from a frame buffer.

In addition, a buffer descriptor encodes additional metadata about the use of the frame buffer; whether it is used to hold frame data or metadata.

Here are the essential parts of the definition of the buffer descriptor as taken from the *ezdp\_frame\_defs.h* file (some non-user serviceable fields have been omitted for clarity):

```
enum ezdp_buffer_mem_type {
    EZDP_EXT_MEM,
    EZDP_INT_MEM
};

enum ezdp_buffer_type {
    EZDP_METADATA_BUF,
    EZDP_FRAME_BUF
};

typedef uint32_t ezdp_buffer_desc_t;
```

```

struct ezdp_buffer_desc {
    union {
        ezdp_buffer_desc_t raw_data;
        struct {
            unsigned valid_data_buf;
            enum ezdp_buffer_mem_type mem_type;
            unsigned id;
        };
    };
};

```

Let's go over the fields composing the buffer descriptor data structure:

**valid\_data\_buf** – the *valid\_data\_buf* specifies whether the buffer contains data or not.



A frame with invalid data buffers cannot be sent to the TM, an interface nor discarded.


**Memory type** – the *mem\_type* field specifies whether the corresponding buffer is located in IMEM or EMEM.

**ID** – the *id* field is an index used by the hardware engines to locate the corresponding buffer in memory.

### 3.3.1.2.1 Linked Buffer Descriptors

As we have seen, each buffer descriptor functions as a pointer to a single 256 byte buffer. Most frames however, are composed of more than a single buffer. Therefore, an efficient representation is needed to describe a series of buffers. The Linked Buffers Descriptor, or LBD for short, is this representation.

The linked buffer descriptor is a memory structure (i.e. array) that holds a series of buffer descriptors and ancillary data and represented by **struct** *ezdp\_linked\_buffers\_desc\_line*.

 For more details about system structure of the linking of NPS buffer descriptor see the [Working with linked buffer descriptors](#) section.

### 3.3.1.3 The frame descriptor

As we have described in the previous section, each frame is assigned a job that is being tracked by the system. Part of the job descriptor structure is the frame descriptor structure which is a data structure in the system memory.

Here is the definition of the frame descriptor data structure:

```

enum ezdp_frame_type {
    EZDP_NULL_FRAME,
    EZDP_EXT_FRAME,
    EZDP_STD_FRAME
};

struct ezdp_frame_desc {
    uint8_t ecc;
    enum ezdp_frame_type type;
    unsigned transmit_confirmation_flag;
    unsigned timestamp_flag;
    unsigned transmit_keep_buf_flag;
    unsigned gross_checksum_flag;
    ...
    unsigned class_of_service;
    unsigned buf_budget_id;
    uint16_t frame_length;
    uint8_t data_buf_count;
    uint8_t header_offset;
    struct ezdp_buffer_desc buf_desc;
    uint8_t free_bytes;
    uint8_t logical_id;
};

```

```
enum ezdp_frame_multicast_control    multicast_control;
unsigned                             job_budget_id;
};
```

Let us go over the main fields of frame descriptor structure:

- **Type** – the *type* field describes the frame type, as described in detail in the [Frame Types](#) section.
  - **EZDP\_STD\_FRAME** – the standard frame type is used to represent standard Ethernet frames and holds up to 7 buffer descriptors in an optimized way to reduce memory accesses to the first buffer.
  - **EZDP\_EXT\_FRAME** – the extended frame type is used to represent frames that require more than 7 frame buffers, such as Ethernet Jumbo frames.
  - **The NULL frame** – as the name implies, frames of the NULL frame type have no frame data buffers at all. This kind of frame is useful when a job without associated frame is passed to SW (for example timer job generated by HW).
- **Frame Length** – the *frame\_length* field contains the total length in bytes of the frame data starting from the layer 2 frame headers. The frame length excludes all other data not received or transmitted even if it is present in the frame data buffers, such as the frame context bytes, and other bytes preceding the header offset field position.
- **Data Buffers Count** – the *data\_buf\_count* field indicates the number of data buffers that hold frame data. This count does not include buffers containing non-frame data, such as extended linked buffers descriptor buffers or buffers that contain software contexts.
- **Buffer Descriptor** – the *buf\_desc* field contains a buffer descriptor for the first, and possibly only, buffer of the frame. Depending on the frame type as described in the *type* field, this buffer may contain frame data, additional buffer descriptors that contain frame data themselves, or both.
- **Header Offset** – the *header\_offset* field provides the offset from the beginning of the first frame data buffer where frame data starts. Note that small and large linked buffers descriptors and any software context information which is not part of the frame data and are not to be transmitted by the MACs are not included in the offset. The offset may be used to add additional frame headers, where required, without the need to copy the original of the frame.
- **Logical ID** – the *logical\_id* field is a software logical context to designate the type of processing the frame should undergo. Logical IDs were described in [The frame logical ID](#) section above.

### 3.3.2 Working with Frame Buffers

Let us recap what we have learned thus far: each job descriptor structure provided by the PMU to a thread for processing has an embedded frame descriptor in it; this frame descriptor, in turn, contains a buffer descriptor that describes one or more buffers in the system memory which contain the frame data.

Accessing and modifying the frame headers and payload is done by copying in one or more of the frame data buffers to a local buffer in CMEM memory, processing it as desired, and then assuming it was modified copying the data back to its place in the frame buffer and updating any applicable information in the frame descriptor (such as frame length or header offset) which is effected by the changes in frame data.

The copy operation is performed by a specialized DMA engine which performs the needed data transfer in the most efficient manner without consuming core processing cycles. The handle to each buffer is the frame data buffer descriptor (*ezdp\_buffer\_desc\_t*) which was described in previous sections.

### 3.3.2.1 Accessing and modifying frame data

#### 3.3.2.1.1 Getting a descriptor of the frame's first data buffer

As explained above, our first order of business in inspecting and modifying frame data is to obtain the buffer descriptor to the buffer which holds the data.

Under most circumstances, the first buffer descriptor we will require is the buffer descriptor of the first frame buffer that holds frame data, as that is where the frame headers are typically located.

To get the buffer descriptor of the first frame data buffer, use the `ezdp_get_first_buf()` API call:

```
int32_t ezdp_get_first_buf(struct ezdp_frame_desc *fd, struct ezdp_buffer_desc *buf_desc, struct
ezdp_linked_buffers_desc_line __cmem* lbd_line, ezdp_frame_buf_iterator_state_t *iter_st)
```

The `ezdp_get_first_buf()` function copies into `buf_desc` the buffer descriptor for the first data buffer frame `fd` and returns the length of the data in the buffer, starting from the header offset until the last byte.

The function requires two additional parameters: `lbd_line`, which is a pointer to a buffer where the system may load a frame linked buffer descriptor line, and `iter_st`, which is a pointer to a buffer where the function may store internal state needed should the caller would later wish to retrieve additional frame buffers.

Here is a usage example:

```
static struct ezdp_buffer_desc bd __cmem_var;
static struct ezdp_linked_buffers_desc_line bd_line __cmem_var;
static ezdp_frame_buf_iterator_state_t iter __cmem_var;

register int32_t len;
register int32_t offset;

while(true) {
    /* Receive job */
    ...
    offset = job_desc.frame_desc.header_offset;
    len = ezdp_get_first_buf(&job_desc.frame_desc, &bd, &bd_line, &iter);
    ...
}
```

#### 3.3.2.1.2 Loading the frame data into local core memory

Now that we have both a buffer descriptor and the desired length of data to copy, we can proceed to load the frame data into the local core memory for processing.

This is done using the `ezdp_load_frame_data()` API call:

```
void ezdp_load_frame_data(void __cmem *dst_ptr, struct ezdp_buffer_desc __cmem *src_bd_ptr,
uint32_t src_bd_offset, uint32_t size, uint32_t flags)
```

The `ezdp_load_frame_data()` call invokes a DMA engine to copy the content of a data buffer in IMEM or EMEM into a buffer in the CMEM.

The `dst_ptr` argument provides the pointer to the CMEM buffer to which the data will be copied, whereas the `src_bd_ptr` argument provides a pointer to a buffer descriptor structure in the CMEM that points to the source data buffer descriptor.

The `src_bd_offset` and `size` arguments provide the offset from the beginning of the data buffer from which to copy the data and the length of data to copy.

The `flags` parameter is currently unused and should be set to zero.

---

**Warning!** Do not use this function to load a linked buffer descriptor. Copying these requires the use of a different API which will be described in the [Working with linked buffer descriptors](#) section ahead.

---



Here is a typical usage example. Note that the frame data is copied at an offset from the start of the array, to leave room for adding additional headers later on.

```
static struct ezdp_buffer_desc bd __cmem_var;
static struct ezdp_linked_buffers_desc_line1 bd_line __cmem_var;
static ezdp_frame_buf_iterator_state_t iter __cmem_var;
static uint8_t data[EZDP_BUFFER_DATA_SIZE] __cmem_var;

register int32_t len;
register int32_t offset;

while(true) {
    /* Receive job */
    ...
    offset = frame_desc.job_desc.frame_desc.header_offset;
    len = ezdp_get_first_buf(&job_desc.frame_desc, &bd, &bd_line, &iter);
    ezdp_load_frame_data(&data[offset], &bd, offset, len, 0);
    ...
}
```

### 3.3.2.1.3 Querying and modifying frame data

Once a copy of the frame data resides in a char array on our local core memory, we may parse and modify the content of the data as we see fit.

Here is an example of adding an 802.1Q VLAN tag header to a frame that was lacking one:

```
struct eth_macs {
    uint8_t dest_mac[6];
    uint8_t src_mac[6];
} __packed;

struct vlan_hdr {
    uint16_t tpid;
    uint16_t priority : 3;
    uint8_t cfi : 1;
    uint16_t tag : 12;
} __packed;

/* Receive job, get BD and length, load frame data... */

uint8_t * p = &data[offset] - sizeof(struct vlan_hdr);
struct vlan_hdr * vlan;

ezdp_mem_copy(p, &data[offset], sizeof(struct eth_macs));

vlan = (struct vlan_hdr *) (p + sizeof(struct eth_macs));

vlan->tpid = 0x8100;
vlan->priority = 0;
vlan->cfi = 1;
vlan->tag = 42;
```

### 3.3.2.1.4 Adjusting the frame metadata

Since we have modified the frame data by adding an 802.1Q VLAN header, we also need to adjust the frame metadata.

Specifically, since we have added a header to the beginning of the frame, we need to adjust both the total the frame descriptor's *frame\_length* field and *header\_offset* fields.

```
job_desc.frame_desc.header_offset -= sizeof(struct vlan_hdr);
job_desc.frame_desc.frame_length += sizeof(struct vlan_hdr);
```



Note that the modified values will not be visible to the system until after we have copied back our local copy of the job descriptor back to a globally visible memory as described in the [Finalizing a job](#) section above.

### 3.3.2.1.5 Synching frame data to global memory

Now that we have modified our local copy of the frame headers, we need to copy our modified frame back to the globally visible memory in either IMEM or EMEM.

This requires us to copy back any changes we have made in the frame data itself as well as make any needed adjustments in the frame and job metadata.

This is done using the `store_frame_data()` API call:

```
void ezdp_store_frame_data(struct ezdp_buffer __cmem * dst_bd_ptr, uint32_t dst_bd_offset, void __cmem * src_ptr, uint32_t size, uint32_t flags)
```

The `ezdp_store_frame_data()` call invokes a DMA engine to copy the content of buffer in the CMEM into a data buffer in IMEM or EMEM.

The `dst_bd_ptr` argument provides a pointer to a buffer descriptor structure in the CMEM that points to the destination data buffer into which the content of the buffer pointed to by the `src_ptr` argument will be copied.

The `dst_bd_offset` and `size` arguments provide the offset from the beginning of the data buffer into which to copy the data and the length of data to copy.

The `store_frame_data()` API accepts an additional parameter `flags`, which allows us to optimize the hardware handling of ECC protection of the frame data:

The hardware that is performing the copy also takes care of calculating the ECC checksum protecting the data in the frame buffer. ECC checksum of frame buffers are handled in 128 byte lines. In the default case, the hardware must read any lines that are touched by the store operation, merge the changed bytes, recalculate the ECC checksum and write back the data and ECC checksum.

As a special case, if you know the data you are storing in the frame buffer in a single store operation comprises all the data of value in all of the 128 byte sized lines you are writing, the flag `EZDP_MEMORY_FLAG_OVERWRITE` may be provided in the `flags` argument to indicate to the hardware that it can ignore any data previously preexisting in any ECC line that contains the data being written. This allows the hardware to use a more permissive algorithm which only calculates the ECC checksum on the new data and writing it into memory, without the need to read back and merge the old data. This can provide significant performance and power optimization, where applicable. Care must be taken however, as use of this flag can cause inadvertent data corruption if used wrongly.

In case of doubt, pass zero as the `flags` argument.

---

**Warning!** Do not use this function to store a linked buffer descriptor. Copying these requires the use of a different API which will be described in the [Working with linked buffer descriptors](#) section ahead.

---

Here is an example where we copy back the modified header we obtained and changed in the previous sections back into a globally visible memory.

```
len += sizeof(struct vlan_hdr);

ezdp_store_frame_data(&bd, job_desc.frame_desc.header_offset, \
    (void*)data, len, EZDP_MEMORY_FLAG_OVERWRITE);
```

Note that since we are replacing the content of the entire frame buffer with our local copy, we pass the `EZDP_MEMORY_FLAG_OVERWRITE` flag allowing the system to use the more efficient variant of ECC checksum calculation as explained above.

Once we have copied over our modified frame data to system memory, we may finalize the job as described in the [Finalizing a job](#) section above.

### 3.3.2.1.6 Tying all the details together

The following code extract is an example of frame header modification. The example retrieves a single frame, copies the frame's first buffer into the local CMEM memory, adds a VLAN header to the frame, copies back the modified header to the system's global memory and transmits the frame.

Please note that many details and error checking have been omitted in this simple example for the purpose of simplicity. Having said that, it is still a good example to assist in understanding how frame buffers can be retrieved, modified and copied back to the global system memory.

```

struct eth_macs {
    uint8_t dest_mac[6];
    uint8_t src_mac[6];
} __packed;

struct vlan_hdr {
    uint16_t tpid;
    uint16_t priority : 3;
    uint8_t cfi : 1;
    uint16_t tag : 12;
} __packed;

static struct ezdp_buffer_desc bd __cmem_var;
static struct ezdp_linked_buffers_desc_line lbd_line __cmem_var;
static ezdp_frame_buf_iterator_state_t iter __cmem_var;
static uint8_t data[EZDP_BUFFER_DATA_SIZE] __cmem_var;

void packet_processing(void) __mem_fmt_slot_func(3);
void packet_processing(void)
{
    register uint32_t len;
    register int32_t offset;
    register uint8_t * p;
    register struct vlan_hdr * vlan;

    while (true) {

        ezdp_receive_job(&job, &job_desc);

        offset = job_desc.frame_desc.header_offset;
        len = ezdp_get_first_buf(&job_desc.frame_desc, &bd, &lbd_line, &iter);
        ezdp_load_frame_data(&data[offset], &bd, offset, len, 0);

        /* Add a VLAN header, tag 42, priority 0 */

        p = &data[offset] - sizeof(struct vlan_hdr);

        ezdp_mem_copy(p, &data[offset], sizeof(struct eth_macs));

        vlan = (struct vlan_hdr *) (p + sizeof(struct eth_macs));

        vlan->tpid = 0x8100;
        vlan->priority = 0;
        vlan->cfi = 1;
        vlan->tag = 42;

        /* Adjust frame meta data */
        job_desc.frame_desc.header_offset -= sizeof(struct vlan_hdr);
        job_desc.frame_desc.frame_length += sizeof(struct vlan_hdr);
        len += sizeof(struct vlan_hdr);

        /* Store modified buffer back to global memory */
    }
}

```

```

ezdp_store_frame_data(&bd, job_desc.frame_desc.header_offset, \
                    (void*)data, len, EZDP_MEMORY_FLAG_OVERWRITE);

/* Transmit job via the channel 1 interface of side 0 */
ezdp_send_job_to_interface(&job, &job_desc, 0, 1, 0);

} /* while (true) */

```

### 3.3.2.2 Accessing the frame payload

Many network applications are satisfied with only examining the first frame data buffer, as this is typically where frame headers are located. Other applications however, require acting upon each buffer of the frame, for example in order to encrypt or decrypt a frame in a VPN application or to search the frame payload in a deep packet inspection type of application.

The *ezdp\_get\_first\_buf()* API call we have seen before is actually part of a set of two functions that acts together as an iterator over a frame data buffer descriptors.

After the iterator state has been initialized by the call to *ezdp\_get\_first\_buf()*, descriptors of the rest of the frame data buffers may be obtained by repeated calls to the *ezdp\_get\_next\_buf()* API call:

***int32\_t ezdp\_get\_next\_buf(struct ezdp\_frame\_desc \*fd, struct ezdp\_buffer\_desc \*buf\_desc, struct ezdp\_linked\_buffers\_desc\_line \_\_cmem\* lbd\_line, ezdp\_frame\_buf\_iterator\_state\_t \*iter\_st)***

The calling convention of *ezdp\_get\_next\_buf()* is identical to *ezdp\_get\_first\_buf()*, with the addition that the function may return a value of -1 to indicate that no more frame data buffers are available.

Here is an example of iterating over all frame buffers and printing the frame data to standard output:

```

for(offset = frame_desc.job_desc.frame_desc.header_offset,
    len = ezdp_get_first_buf(&job_desc.frame_desc, &bd, &lbd_line, &iter);
    len != -1;
    offset = 0, len = ezdp_get_next_buf(&job_desc.frame_desc, &bd, &lbd_line, &iter) ) {

    ezdp_load_frame_data((&data[offset], &bd, offset, len, 0);

    for(int32_t i = 0; i < len; i++) {
        printf("%02x", data[offset + len]);
    }
}

```

### 3.3.2.3 Allocating and releasing data buffers

The buffers with frame data are automatically allocated by the HW (i.e. by the network DMA control of the BMU) without the need for SW intervention.

In other cases the software may still need to explicitly allocate, modify or release frame buffers from the Buffer Management Unit. The following sub-sections describe the relevant buffer management APIs.

#### 3.3.2.3.1 Buffer pools

As described in the [Buffer descriptors](#) section, the Buffer Management Unit manages frame buffers in two pools: a pool of IMEM buffers and a pool of EMEM buffers. Allocation requests must specify the pool from which to allocate using the appropriate value of the *ezdp\_buffer\_mem\_type* enum:

- ***EZDP\_EXT\_MEM*** – allocate from EMEM.
- ***EZDP\_INT\_MEM*** – allocate from IMEM.

### 3.3.2.3.2 Allocating a single buffer

*ezdp\_buffer\_desc\_t* **ezdp\_alloc\_buf**( *enum ezdp\_buffer\_mem\_type* pool\_type, *uint32\_t* budget\_id)

The *ezdp\_alloc\_buf*() call allocates a single frame buffer from the Buffer Management Unit and charges it to the specified budget.

For efficiency reasons, the call returns a variable *ezdp\_buffer\_desc\_t*, which is a representation of a buffer descriptor structure that fits inside a single register. To check if the allocation was successful, use the *ezdp\_buf\_alloc\_failed*() API call:

*bool* **ezdp\_buf\_alloc\_failed**(*ezdp\_buffer\_desc\_t* ret)

Here is an example to illustrate the allocation flow:

```
static struct ezdp_job_desc job_desc __cmem_var;
...

struct ezdp_buffer_desc bd;
uint32_t budget = job_desc.frame_desc.buf_budget_id;

bd.raw_data = ezdp_alloc_buf(EZDP_EXT_MEM, budget);

if(unlikely(ezdp_buf_alloc_failed(bd))) {
    /* Allocation failed! */
}
```

### 3.3.2.3.3 Releasing a single buffer

*void* **ezdp\_free\_buf**( *uint32\_t* budget\_id, *struct ezdp\_buffer\_desc \_\_cmem* \* bd\_ptr )

The *ezdp\_free\_buf*() call frees an allocated single frame buffer, returns it to the Buffer Management Unit and credits the specified budget accordingly.

The budget ID specified should be the same one used when the frame buffer was allocated – either the explicit budget ID specified when the buffer was allocated via a programmatic allocation request, or in the case of a frame buffer that was automatically allocated by the network DMA, the budget ID that is specified in the frame descriptor *budget\_id* field.

### 3.3.2.3.4 Allocating and releasing multiple buffers

*uint32\_t* **ezdp\_alloc\_multi\_buf**(*enum ezdp\_buffer\_mem\_type* pool\_type, *uint32\_t* budget\_id, *uint32\_t* num\_of\_bufs, *struct ezdp\_buffer\_desc \_\_cmem* \* bds\_ptr)

*void* **ezdp\_free\_multi\_buf**( *uint32\_t* budget\_id, *uint32\_t* num\_of\_bufs, *struct ezdp\_buffer\_desc \_\_cmem* \* bds\_ptr)

The *ezdp\_alloc\_multi\_buf*() and *ezdp\_free\_multi\_buf*() calls are equivalent to their single buffer counterparts with the exception that these functions work on multiple buffers at once.

If you wish to allocate or free multiple buffers at once it is more efficient to use a single call to these functions than multiple calls to their single buffer brethren.

## 3.4 Parsing Frame Headers

The process of frame header parsing is an integral part of packet processing. The purpose of this process is three fold:

- **Validation** – validate that the information supplied in the headers is correct and the frame is valid. For example, make sure that Ethernet destination and source MAC addresses are not identical.
- **Classification** – classify the frames into regular frames and those requiring special treatment. For example, an STP control frame requires different treatment than a subscriber frame carrying an IP packet.
- **Decoding** – extract actionable information from the headers which will assist in further frame processing. For example, provide the offset into the frame header where the layer 3 header begins and the type of the layer 3 protocol, regardless of whether the frame's MAC protocol is Ethernet with QinQ VLAN tags or PPPoE.

The NPS is equipped with several mechanisms to assist in these tasks for both standard protocols whose header structure is known as well as proprietary protocols.

### 3.4.1 Protocol Decoders

The NPS protocol decoders are hardware engines which perform protocol classification, validation and decoding on a set of known protocol families.

The following protocols are supported:

- MAC (Media Access Control), including Ethernet (IEEE 802.3), VLAN (IEEE 802.1) and PPPoE
- IPv4
- IPv6
- MPLS
- TCP

#### 3.4.1.1 Principle of operation

A pointer is provided to the location in CMEM where the frame headers of the specific type are located.

A second pointer is provided to a buffer in CMEM where the data structure with the results will be written to.

The protocol header decoder is invoked, and it parses the headers and fills in the result data structure.

Optionally, the first 32 bits of the result data structure are also returned in a core register as the return value of the invoking C function; this allows some decisions, such as dropping of invalid frames, to be taken without incurring the cost of performing a load of the result information from memory.

Since all the protocol decoders work in the same fashion, we shall use the MAC protocol decoder as an example to illustrate how the protocol decoders are best used.

#### 3.4.1.2 An example: the MAC protocol decoder

The MAC protocol decoder handles common OSI layer two, media access control protocols, such as IEEE 802.3 and IEEE 802.1 wired Ethernet, VLAN, network management and the PPPoE protocol.

The `ezdp_decode_mac()` API function is used to invoke this protocol decoder:

```
ezdp_decode_mac_retval_t ezdp_decode_mac( uint8_t __cmem *frame_ptr, uint32_t size, struct
ezdp_decode_mac_result __cmem *decode_result)
```

The `frame_ptr` parameter is used to supply a pointer to core memory where the MAC frame header starts.

The *size* parameter is the size in bytes of the buffer pointed to by the *frame\_ptr* parameter which the MAC protocol decoder is allowed to scan.

The *decode\_result* parameter is used to supply a pointer to C structure of the type *struct ezdp\_decode\_mac\_result* in the core memory where the protocol decoder will write the result of the operation.

The return type of the *ezdp\_decode\_mac()* function is a C structure of the type *struct ezdp\_decode\_mac\_retval* which is a digest of the decode operation results and contains the same information of the first four bytes of the *struct ezdp\_decode\_mac\_result*.

Since a CTOP register is four bytes in size, accessing information provided in this digest requires no memory accesses and is therefore more efficient and preferable where applicable.

Here are the major portions of the definitions of the result data structures:

```
typedef uint16_t ezdp_decode_mac_control_t;

struct ezdp_decode_mac_control {
    union {
        ezdp_decode_mac_control_t raw_data;

        struct {
            ...
            unsigned smac_equals_dmac;
            unsigned user_config3;
            unsigned user_config2;
            unsigned user_config1;
            unsigned user_config0;
            unsigned ipv6_multicast;
            unsigned ipv4_multicast;
            unsigned vrrp_mac;
            unsigned mac_control_other;
            unsigned mac_control_lsb_2x;
            unsigned mac_control_lsb_1x;
            unsigned mac_control_lsb_0x;
            unsigned my_mac;
        } __packed;
    };
};

typedef uint16_t ezdp_decode_mac_protocol_type_t;

struct ezdp_decode_mac_protocol_type {
    union {
        ezdp_decode_mac_protocol_type_t raw_data;

        struct {
            ...
            unsigned user_config_vlan2;
            unsigned pppoe_discovery;
            unsigned pppoe_session;
            unsigned user_config3;
            unsigned user_config2;
            unsigned user_config1;
            unsigned user_config0;
            unsigned length;
            unsigned ipv6;
            unsigned mpls_multicast;
            unsigned mpls_unicast;
            unsigned arp;
            unsigned user_config_vlan1;
            unsigned user_config_vlan0;
            unsigned ipv4;
        } __packed;
    };
};

typedef uint32_t ezdp_decode_mac_retval_t;

struct ezdp_decode_mac_retval {
    union {
        ezdp_decode_mac_retval_t raw_data;
```

```

        struct {
...
            unsigned number_of_tags;
...
            unsigned ipv6_in_pppoe;
            unsigned ipv4_in_pppoe;
            struct ezdp_decode_mac_errors error_codes;
            struct ezdp_decode_mac_control control;
        };
    };
};

struct ezdp_decode_mac_result {
...
    unsigned number_of_tags;
...
    unsigned ipv6_in_pppoe;
    unsigned ipv4_in_pppoe;
    struct ezdp_decode_mac_errors error_codes;
    struct ezdp_decode_mac_control control;
    struct ezdp_decode_mac_protocol_type tag2_protocol_type;
    struct ezdp_decode_mac_protocol_type tag1_protocol_type;
    struct ezdp_decode_mac_protocol_type last_tag_protocol_type;
    struct ezdp_decode_mac_protocol_type tag3_protocol_type;
...
    uint8_t layer2_size;
    uint16_t da_sa_hash;
...
};

```

The validity of the frame is reported by the *error\_codes* field. The *error\_codes.decode\_error* flag sub-field indicates if any protocol violation has been detected while parsing the frame, while the other sub-fields report which specific error, if any, was detected.

The class of the frame (regular or requiring special treatment) is indicated by the *control* field. If any of the flags of this field are set, it indicates the frame requires special treatment as specified by the specific flag. These flags may set for specific special purpose destination MAC addresses, such as those designating VRRP or IP multicast ranges or as user defined MAC address ranges which are configured to the protocol decoder via the NPS control plane API.

The rest of the result data structure provides useful information for the next stages of processing, such as a hash of the source and destination MAC addresses, the number of VLAN tags, if any, the layer 3 protocol reported by the last MAC header and the offset from the beginning of the frame data where the last MAC header ends.

Here is a code snippet of a typical usage example of the MAC protocol decoder inside a packet processing event loop:

```

static struct ezdp_decode_mac_result decode_result __cmem_var;

...

while(true) {

    register struct ezdp_decode_mac_retval ret;
    register uint8_t * p = &data[HEADER_OFFSET];

    /*
     * Receive a job and load frame first buffer
     */
    ...

    ret.raw_data = ezdp_decode_mac(p, len, &decode_result);

    if(ret.error_codes.decode_error) {
        print_warning("Invalid Ethernet frame - frame discarded \n");
        ezdp_discard_job(&job, &job_desc);
        continue;
    }
}

```

```
if(ret.control.my_mac)
    /* The dest Ethernet MAC address is our MAC, send to control plane */
    ezdp_send_job_to_interface(&job, &job_desc, 0, CONTROL_PORT, 0);
    continue;
}


if (ret.number_of_tags) {

    /* The frame has number_of_tags VLAN headers */
    ...

}

/* Skip over the L2 header using the parser provided size */
p += decode_result.layer2_size;

if(decode_result.last_tag_protocol_type.ipv4 ||
decode_result.last_tag_protocol_type.ipv6) {
    // Handle IP packet
    ...
}
...
};
```

 For a full list of the protocol header decoder APIs, including what information is provided by each of them, refer to the “Frame Data Decoding (ezdp\_decode.h)” section in the *EZdp Reference Manual*.



### 3.5 Counter Acceleration

Networking solutions commonly use statistic counters. In the NPS these are accelerated by hardware. Often, an application will maintain several counters for packets, bytes or other items in various forms per business entity, be it a port, a subscriber or a network flow.

The statistics acceleration engine in the NPS provides the service of maintaining millions of logical counters and exposes an interface to efficiently maintain them in a coherent manner with minimum CPU cycles and memory accesses.

The NPS offers two modes of operation for the statistics based on the expected usage of the counters: on-demand and posted. The **on-demand** mode is intended for counters that are expected to be both written to and read by the data plane packet-processing code and needs to be coherent. The **posted** mode provides a “fire and forget” optimized mode of operation for updating counters and is intended for counters that are updated by the data plane application and read (collect statistics) only by the control plane application at lower rates.

The statistic counters block in NPS uses another type of memory and has its own MSID.

Several MSIDs can be configured for on-demand and posted counters.

Accessing the counter in the data plane is done using struct `ezdp_sum_addr` which contains the counter memory id and counter id.

Here are the essential parts of the definition of the summarized address descriptor as taken from the `ezdp_memory_defs.h` file (some non-user serviceable fields have been omitted for clarity):

```
enum ezdp_mem_space_type
{
    EZDP_INTERNAL_MS      =    0x0,
    EZDP_EXTERNAL_MS      =    0x1
};

typedef uint32_t ezdp_sum_addr_t;

struct ezdp_sum_addr{
    union{
        ezdp_sum_addr_t      raw_data;
        struct{
            enum ezdp_mem_space_type    mem_type ;
            unsigned                    msid;
            unsigned                    element_index;
        };
    };
};
```

Let's go over the fields composing summarized address data structure:

- **Memory type** – the *mem\_type* field specifies whether the corresponding buffer is located in IMEM or EMEM.
  - All statistic counters are located in EMEM thus it should be hardcoded set to 0x1.
- **MSID** – the *msid* field specifies memory system id of the statistic counter memory you are trying to access. The MSID is defined by the control plane
- **Element\_index**– the *element\_index* field is an index of the counter to access.

### 3.5.1 On-demand Counters

The following forms of on-demand counters are supported:

- **Single 59 bit counters** which track a single event each. For example, per-subscriber packet drop counters.
- **Dual 96 bit counters** which track two related events; each is updated in an atomic manner. The 96 bits are flexibly partitioned to two counters (byte counter, event counter). For example, a per network interface packet count and byte count counter.
- **Bitwise 64 bit counters** that can be accessed in a resolution of 1, 2, 4, 8 and 16 bits and can be used to implement optimized semaphores or smaller counters.
- **Token bucket counters** which track the traffic color and used for traffic shaping.
- **Hierarchical token bucket counters** which allow implementation of advanced metering of traffic using multiple token buckets with a set of pre-configured relationships between them. For example, coupled policer.
- **Watchdog counters** that are designed to monitor keep-alive events based on configurable profiles and notify when the number of events exceeds its min/max threshold. For example, used to implement OAM Continuity check functionality.

The sections below describe basic counter operations using single and dual counters. Further descriptions of advanced counter operations like message queue, shadow and advanced counter types such as token bucket and watchdog can be found in the [Advanced Statistic Counter Features](#) section.

#### 3.5.1.1 Single counters

The table below summarizes the APIs for single counters:

API Name	Description
<code>ezdp_write_single_ctr</code>	Write single counter with the value specified
<code>ezdp_read_single_ctr</code>	Read single counter value.
<code>ezdp_inc_single_ctr</code> <code>ezdp_read_and_inc_single_ctr</code>	Increment single counter by the value specified and optionally read previous counter value.
<code>ezdp_dec_single_ctr</code> <code>ezdp_read_and_dec_single_ctr</code>	Decrement single counter by the value specified and optionally read previous counter value.
<code>ezdp_reset_single_ctr</code> <code>ezdp_read_and_reset_single_ctr</code>	Reset single counter to zero and optionally read previous counter value
<code>ezdp_xchg_single_ctr</code>	Write single counter with the value specified and read previous counter value
<code>ezdp_cond_dec_single_ctr</code> <code>ezdp_read_and_cond_dec_single_ctr</code>	Conditionally decrement single counter by the value specified and optionally read previous counter value
<code>ezdp_prefetch_single_ctr</code>	Pre-fetch single counter into the local cache


In addition write and read configuration is also allowed from data plane using `ezdp_write_single_ctr_cfg` and `ezdp_read_single_ctr_cfg` API routines.

**`void ezdp_inc_single_ctr(ezdp_sum_addr_t cnt_addr, uint16_t inc_value)`**

`ezdp_inc_single_ctr` increment counter pointed by `cnt_addr` with 16 bit unsigned `inc_value`.

**uint32\_t ezdp\_read\_and\_inc\_single\_ctr(ezdp\_sum\_addr\_t cnt\_addr, uint16\_t inc\_value, uint64\_t \_\_cmem \*orig\_value, bool \*overflow)**

*ezdp\_read\_and\_inc\_single\_ctr* is like *ezdp\_inc\_single\_ctr* function but in addition reads and returns the previous counter *orig\_value* value and indication if counter overflowed in the *overflow* field.

 All APIs are implemented in the similar way, for more details refer to the “Counter Operations (ezdp\_counter.h)” section in the *EZdp Reference Manual*.

### 3.5.1.2 Dual counters

The table below summarizes the APIs for dual counters:


API Name	Description
<i>ezdp_read_dual_ctr</i>	Read dual counter values (byte and event)
<i>ezdp_inc_dual_ctr</i> <i>ezdp_read_and_inc_dual_ctr</i>	Increment dual counter with the values specified (byte and event) and optional read previous counter values
<i>ezdp_dec_dual_ctr</i> <i>ezdp_read_and_dec_dual_ctr</i>	Decrement dual counter's byte value by the value specified and event value by 1, and optionally read previous counter values
<i>ezdp_reset_dual_ctr</i> <i>ezdp_read_and_reset_dual_ctr</i>	Reset dual counter values (byte and event) to zero and optional read previous counter values
<i>ezdp_prefetch_dual_ctr</i>	Prefetch dual counter into the local cache


In addition write and read configuration is also allowed from data plane using *ezdp\_write\_dual\_ctr\_cfg* and *ezdp\_read\_dual\_ctr\_cfg* API routines.

### 3.5.2 Posted Counters

Posted counters are 64 bit “fire and forget” counters which are designed to be written to by the data plane at a high rate and read at a much lower rate by the control plane. Posted counters support either single or dual counter atomic operations.

- **Single counter operations** allow tracking a single event each. For example, per-subscriber packet drop counters.
- **Dual counter operations** allow tracking two related events; each are updated in an atomic manner. For example, a per network interface packet count and byte count counter. Dual counter operations can only be performed on even counter elements, and will affect both the element and its successor.

 Not all dual counter operations are atomic, for more details refer to the “Counter Operations (ezdp\_counter.h)” section in the *EZdp Reference Manual*.

 For the full details on statistics counters API, refer to the “Counter Operations (ezdp\_counter.h)” section in the *EZdp Reference Manual*.

Below is description of basic posted counter operation, a description of the advanced operation and features can be found in the [Advanced Statistic Counter Features](#) section.

### 3.5.2.1 Initializing the counter

Counters may be initialized by either the control plane or data plane. Typically the initialization will be done once from the control plane. This is done by *ezdp\_write\_posted\_ctr* and *ezdp\_dual\_write\_posted\_ctr* APIs.

```
void ezdp_write_posted_ctr(ezdp_sum_addr_t cnt_addr, uint64_t value)
```

*ezdp\_write\_posted\_ctr* writes value to the counter defined in *cnt\_addr*.

```
void ezdp_dual_write_posted_ctr(ezdp_sum_addr_t cnt_addr, uint64_t cnt_value1, uint64_t cnt_value2)
```

*ezdp\_dual\_write\_posted\_ctr* writes *cnt\_value1* to the counter defined in *cnt\_addr* and *cnt\_value2* to sequential counter (*cnt\_addr.element\_index+1*).



*ezdp\_dual\_write\_posted\_ctr* is not atomic and is implemented as two writes to two successive counters.

### 3.5.2.2 Increment the counter

Counters are incremented through a call to the *ezdp\_add\_posted\_ctr* or *ezdp\_dual\_add\_posted\_ctr* APIs.

```
void ezdp_add_posted_ctr(ezdp_sum_addr_t cnt_addr, int32_t value)
```

*ezdp\_add\_posted\_ctr* adds signed 32 bit *value* to the counter defined in *cnt\_addr*.

```
void ezdp_dual_add_posted_ctr(ezdp_sum_addr_t cnt_addr, int16_t cnt_value1, int16_t cnt_value2)
```


*ezdp\_dual\_add\_posted\_ctr* atomic adds signed 16 bit *cnt\_value1* to the counter defined in *cnt\_addr* and *cnt\_value2* to the sequential counter (*cnt\_addr.element\_index+1*).

## 3.6 Hardware-assisted Search Structures

Yet another form of hardware acceleration provided by NPS is accelerated lookup and update. This acceleration is supported by a combination of highly optimized memory structures and proprietary CTOP instructions which minimize the number of CTOP cycles and memory accesses per lookup.

The NPS supports the following search structures: direct table, hash, longest prefix match and several types of TCAM tables. While the high level API of direct tables and hash structures is described below, longest prefix match, TCAM tables and low level APIs are described in the [Advanced Search Features](#) section.

Direct table and hash search structures are optimized and HW accelerated and designed to provide high rate lockless implementation of lookup operations assuming low rate cross thread synchronization mechanism in support of update operations (add, modify, delete, etc.).

 For the full details on search structures APIs, please refer to section “Search Structure Operations (ezdp\_search.h)” section in the *EZdp Reference Manual* and “Search Structures Group” section in *EZcp Reference Manual*.

### 3.6.1 Search Memory

Special search memory is defined in the NPS. This memory type has several copies, or replicas, in the system memory for better memory load distribution and redundancy and in-band explicit protection of the memory data. The memory is protected with parity bits, and not ECC, which allows for memory error detection only, and not fix. High-level APIs handle such memory errors by either trying the lookup again, in which case another result copy will be read and user will get the valid result, or just return with an error. In addition to the memory error reported to the application, an exception is raised to EZcp (control plane) which will fix this memory error by writing the correct data again.

This memory is optimized for massively read data that gets duplicated for performance reasons, and at the same time requires relatively rare updates (e.g. search tables).

The search memory is divided into chunks of 32 byte data or special 16-byte data chunk. Each chunk is protected by 3 bits of parity. Based on this, the search entry can be 16 bytes or multiples of 32 bytes and it is the user’s responsibility to reserve these bits, for each chunk, when defined users search entry. In addition, 1 bit in the first chunk should be reserved for SW purposes.

Here is an example for a definition of a direct table search structure for storing LAG port mappings:

```
struct lag_info {
    unsigned    : EZDP_LOOKUP_PARITY_BITS_SIZE;    // 3 bits
    unsigned    : EZDP_LOOKUP_RESERVED_BITS_SIZE; // 1 bit
    unsigned num_ports : 4;
    uint8_t ports[15];
};
```

The size of the structure is 16 bytes. The first 4 bits are reserved for internal HW and SW use and other bits are user defined.

Another example for a definition of a direct table search structure with a size of 64 bytes:

```
struct table_entry {
    unsigned first_chunk_parity : EZDP_LOOKUP_PARITY_BITS_SIZE;
    unsigned first_chunk_sw     : EZDP_LOOKUP_RESERVED_BITS_SIZE;
    unsigned reserved1 : 28;
    uint32_t data1_1;
    uint32_t data1_2;
    uint32_t data1_3;
    uint32_t data1_4;
    uint32_t data1_5;
```

```

uint32_t data1_6;
uint32_t data1_7;

unsigned second_chunk_parity      : EZDP_LOOKUP_PARITY_BITS_SIZE;
unsigned reserved2 : 29;
uint32_t data2_1;
uint32_t data2_2;
uint32_t data2_3;
uint32_t data2_4;
uint32_t data2_5;
uint32_t data2_6;
uint32_t data2_7;
};

```

As can be seen in this example above, the size of this structure is 64 bytes. You can also see that 3 bits of the parity are reserved for each 32 byte of chunk and for the first chunk an additional bit is reserved for SW use (match indication).

### 3.6.2 Direct Table Example

Direct tables are the simplest form of search structure, comprising a table of a fixed size of entries in memory. A table's structure is defined in the control plane while the table's entries may be modified from both the control plane and data plane.

The following is an example of a Link Aggregation Group port mapping table illustrating the search table API usage:

#### 3.6.2.1 Key and result structures

Direct table keys are indexes to the table. The structure of the results is software defined, with the exception of the reserved bits for HW (used to store parity bits for memory error detection) and a bit reserved for SW (to indicate if entry is occupied or not).

Here is an example for a definition of a direct table search structure for storing LAG port mappings:

```

struct lag_info {
    unsigned      : EZDP_LOOKUP_PARITY_BITS_SIZE;
    unsigned      : EZDP_LOOKUP_RESERVED_BITS_SIZE;
    unsigned num_ports : 4;
    uint8_t  ports[15];
};

```

#### 3.6.2.2 Table descriptor initialization

Since tables are defined and their resources managed by the control plane, an initialization call is required for the data plane to retrieve a search descriptor that will provide the needed information to access the table. This is done via the `ezdp_init_table_struct_desc()` API call:

```

uint32_t ezdp_init_table_struct_desc(uint32_t struct_id, ezdp_table_struct_desc_t * table_struct_desc,
char __cmem *work_area_ptr, uint32_t work_area_size)

```

The `ezdp_init_table_struct_desc()` call initializes the `table_struct_desc` structure with the information required to access the table search structure whose numeric ID is `struct_id`. The `struct_id` is the number of search structure set by the control plane. The function uses the CMEM buffer pointed to by `work_area_ptr` as a temporary work area, which may be reused when the function returns.

The return value indicates success (0) or failure (EINVAL).

Since the control and data plane are two separate applications it is recommended to validate that control and data plane definitions are similar, for this purpose use the `ezdp_validate_table_struct_desc` API.

**uint32\_t** ezdp\_validate\_table\_struct\_desc(**ezdp\_table\_struct\_desc\_t** \*table\_struct\_desc, **uint32\_t** entry\_size)

The *ezdp\_validate\_table\_struct\_desc()* call validate that *entry\_size* defined in data plane matches the *entry\_size* defined in the control plane.

The return value indicates success (0) or failure (EINVAL).

Here is an example of initialization of a direct table search structure descriptor:

```
#define LAG_TABLE_STRUCT_ID 0

static struct ezdp_table_struct_desc lag_table __ccm_var;

static char work_area[EZDP_TABLE_HIGH_LEVEL_WORK_AREA_SIZE] __ccm_var;

int main(void)
{
    uint32_t retval;
    . . .

    retval = ezdp_init_table_struct_desc(LAG_TABLE_STRUCT_ID, \
                                         &lag_table, &work_area, sizeof(work_area));
    if(retval != 0)
    {
        printf("init of table struct (%d) failed. \
              Error Code %d. Error String %s \n",
              LAG_TABLE_STRUCT_ID, retval, ezdp_get_err_msg());
        exit(1);
    }

    retval = ezdp_validate_table_struct_desc (&lag_table, sizeof(struct lag_info));
    if(retval != 0)
    {
        printf("validation of table struct (%d) failed. \
              Error Code %d. Error String %s \n", \
              LAG_TABLE_STRUCT_ID, retval, ezdp_get_err_msg());
        exit(1);
    }

    ezdp_run(&packet_processing, 0);

    return EXIT_SUCCESS;
}
```

### 3.6.2.3 Looking up a table entry

To perform a lookup in the table, use the *ezdp\_lookup\_table\_entry()* API call:

**uint32\_t** ezdp\_lookup\_table\_entry(**ezdp\_table\_struct\_desc\_t** \*struct\_desc, **uint32\_t** key, **void** \_\_ccmem \*entry\_ptr, **uint32\_t** entry\_ptr\_size, **uint32\_t** flags)

The *ezdp\_lookup\_table\_entry()* queries the entry at index *key* of the direct table whose base address and properties are given in the *struct\_desc* parameter and will write the entry into the CMEM buffer pointed by *entry\_ptr* of size *entry\_ptr\_size*.

The function uses the CMEM buffer pointed to by *work\_area\_ptr* as a temporary work area, which may be reused when the function returns.

The *struct\_desc* is table search structure descriptor which was initialized by a call to *ezdp\_init\_table\_struct\_desc* API as described above in the [Table descriptor initialization](#) section.

The *flags* field is currently not in used and should be set to 0.



The return value indicates success (0), entry not found (ENOENT) or memory error (EIO). The memory error is returned when multiple attempts to read a table entry failed with memory error indication from HW.

Here is an example of a function implementing a lookup into the LAG table:

```
static struct lag_info info __ccm_var;

static struct ezdp_table_struct_desc lag_table __ccm_var;

struct lag_info * lookup_lag( uint8_t lag_id ) __fast_path_code;
struct lag_info * lookup_lag( uint8_t lag_id )
{
    uint32_t retval;

    retval = ezdp_lookup_table_entry(&lag_table, lag_id, &info, sizeof(info), 0);

    return (retval ==0 ? &info : NULL);
}
```

### 3.6.2.4 Updating the table

To add entry to the table, use the `ezdp_add_table_entry()` API call:

**`uint32_t ezdp_add_table_entry( ezdp_table_struct_desc_t *struct_desc, uint32_t key, void __ccmem *entry_ptr, uint32_t entry_ptr_size, uint32_t flags, char __ccmem *work_area_ptr, uint32_t work_area_size )`**

The `ezdp_add_table_entry()` function writes the value pointed by `entry_ptr` from the CMEM to the direct table entry at index `key` of the direct table whose base address and properties are given in the `struct_desc` parameter. All replicas will be updated.

The `struct_desc` is table search struct descriptor which was initialized by call to `ezdp_init_table_struct_desc` API and described above in the [Table descriptor initialization](#) section.

The argument `flags` can be used to control the behavior of the add operation and has two options `EZDP_OPPORTUNISTIC` and `EZDP_UNCONDITIONAL`.

`EZDP_UNCONDITIONAL` value indicates that no check if the ‘entry already exists’ is performed. In such a case, no lookup (to find if entry already exists) is executed and locking this entry is not required.

`EZDP_OPPORTUNISTIC` value indicates that the add operation can be opportunistic, meaning that the operation would be aborted when a memory error is detected or entry is already locked.

The return value indicates success (0), entry already exist error (EEXIST) or memory error (EIO). The memory error is returned when multiple attempts to read a table entry failed with memory error indication from HW.

The following example implements the addition and removal of LAG table entries:

```
static char work_area[EZDP_TABLE_HIGH_LEVEL_WORK_AREA_SIZE] __ccm_var;
static struct lag_info info __ccm_var;

static struct ezdp_table_struct_desc lag_table __ccm_var;

void add_lag(uint8_t lag_id, uint8_t num_ports, uint8_t * ports) __slow_path_code;
void add_lag(uint8_t lag_id, uint8_t num_ports, uint8_t * ports)
{
    info.num_ports = num_ports;
    ezdp_mem_copy(info.ports, ports, sizeof(uint8_t) * num_ports);
    ezdp_add_table_entry(&lag_table, lag_id, &info, sizeof(info), 0, \
                        &work_area, sizeof(work_area));

    return;
}
```



```

void del_lag(uint8_t lag_id) __slow_path_code;
void del_lag(uint8_t lag_id)
{
    ezdp_delete_table_entry(&lag_table, lag_id, 0, &work_area, sizeof(work_area));
    return;
}

```

Modifying, updating or deleting a table's entries are done in a similar fashion to the operations presented above.

### 3.6.3 Hash Example

Hash structures are very similar to direct tables, with the exception that the table key is hashed, the hash is used as an index to the structure and key stored in the entry.

The NPS handles hash collisions and entry chaining automatically and efficiently. When a hash structure is defined via the control plane API, it is possible to state whether the specific hash structure should be optimized for a sparse hash where collisions are rare (called "single cycle" hash) or for a populated hash where many collisions are expected (called "multi cycle" hash). Both forms fully support hash collisions and the difference is merely in optimization for the most common case.

The following is an example of an Ethernet switch MAC learning hash table illustrating the hash structure API usage.

#### 3.6.3.1 Key and result structures

Hash structure keys and results follow the same rules as tables. However to address hash collisions and entry chaining, a hash entry is bigger than the hash result, with the additional room is used to store internal metadata needed for the hash chaining.

The macro `EZDP_PAD_HASH_ENTRY` is provided to pad the result for additional metadata and to match the entry size to valid size. Padding size is calculated based on the size of the hash key and result.

Here is an example of the structure definitions needed for our MAC table example:

```

struct mac_key {
    struct eth_mac mac;
    uint16_t vrf;
};
struct mac_info {
    unsigned : EZDP_LOOKUP_PARITY_BITS_SIZE;
    unsigned : EZDP_LOOKUP_RESERVED_BITS_SIZE;
    unsigned reserved: 3;
    unsigned in_use : 1;
    uint8_t port;
    uint16_t vlan;
};
struct mac_entry {
    struct mac_info result;
    EZDP_PAD_HASH_ENTRY(sizeof(struct mac_info), sizeof(struct mac_key));
};

```

#### 3.6.3.2 Hash descriptor initialization

Hash structure descriptors are initialized in the same fashion as in direct tables:

```

#define MAC_TABLE_STRUCT_ID 1

static char work_area[EZDP_HASH_HIGH_LEVEL_WORK_AREA_SIZE(sizeof(struct
mac_info), sizeof(struct mac_key))] __ccm_var;

static struct ezdp_hash_struct_desc mac_hash __ccm_var;

```

```

int main(void)
{
    uint32_t retval;
    ...
    retval = ezdp_init_hash_struct_desc(MAC_TABLE_STRUCT_ID, &mac_hash, \
                                       &work_area, sizeof(work_area));
    if(retval != 0)
    {
        printf("init of hash struct (%d) failed. \
               Error Code %d. Error String %s \n",\
               MAC_TABLE_STRUCT_ID, retval, ezdp_get_err_msg());
        exit(1);
    }

    retval = ezdp_validate_hash_struct_desc (&mac_hash, true,\
                                             sizeof(struct mac_key), sizeof(struct mac_info), sizeof(struct mac_entry));
    if(retval != 0)
    {
        printf("validation of hash struct (%d) failed. \
               Error Code %d. Error String %s \n",\
               MAC_TABLE_STRUCT_ID, retval, ezdp_get_err_msg());
        exit(1);
    }

    ...
    ezdp_run(&packet_processing, 0);

    return EXIT_SUCCESS;
}

```

### 3.6.3.3 Lookup a hash entry

To perform a lookup in the hash structure, use the `ezdp_lookup_hash_entry()` API call:

```

uint32_t ezdp_lookup_hash_entry( ezdp_hash_struct_desc_t *struct_desc, void __cmem *key_ptr,
uint32_t key_ptr_size, void **result_ptr, uint32_t *result_ptr_size, uint32_t flags, char __cmem
*work_area_ptr, uint32_t work_area_size)

```

The `ezdp_lookup_hash_entry()` will find the entry with matched `key_ptr` in the hash table whose base address and properties are given in the `struct_desc` parameter. The function uses the CMEM buffer pointed to by `work_area_ptr` as a temporary work area and writes the entry into it. The function returns the pointer to the result which is located in `work_area_ptr` and the size of the `result_ptr_size`. If developers want to reuse the `work_area_ptr` they must copy the result from `result_ptr` to another place when the function returns.

The `struct_desc` is table search struct descriptor which was initialized by the call to `ezdp_init_table_struct_desc` API and described above in the [Table descriptor initialization](#) section.

The `flags` field is currently not in used and should be set to 0.

The return value indicates success (0), entry not found (ENOENT) or memory error (EIO). The memory error is returned when multiple attempts to read a table entry failed with memory error indication from HW.

Here is an example of a function implementing a lookup into the LAG table:

```

static char work_area[EZDP_HASH_HIGH_LEVEL_WORK_AREA_SIZE(sizeof(struct
mac_info), sizeof(struct mac_key))] __ccm_var;

static struct ezdp_hash_struct_desc mac_hash __ccm_var;

static struct mac_info info __ccm_var;
static struct mac_key key __ccm_var;

struct mac_info * lookup_mac( uint16_t vrf, struct eth_mac * mac )
{

```

```

uint32_t      retval;
void          *result_ptr;
uint32_t      result_ptr_size,

key.vrf = vrf;
ezdp_mem_copy(key.mac.byte, mac, sizeof(struct eth_mac));

retval = ezdp_lookup_hash_entry(&mac_table, &key, sizeof(key), \
                                &result_ptr, &result_ptr_size, 0, &work_area, sizeof(work_area));

if(retval != 0) {
    return NULL;
}
else{
    ezdp_mem_copy(&mac_info, result_ptr, result_ptr_size);
    return &mac_info;
}
}

```

### 3.6.3.4 Updating the hash

To add an entry to the hash, use the `ezdp_add_hash_entry()` API call:

```

uint32_t ezdp_add_hash_entry(ezdp_hash_struct_desc_t *struct_desc, void __cmem *key_ptr,
uint32_t key_ptr_size, void __cmem *result_ptr, uint32_t result_ptr_size, uint32_t flags, char
__cmem *work_area_ptr, uint32_t work_area_size)

```

The `ezdp_add_hash_entry()` will write the result pointed by `result_ptr` from the CMEM at index `key` hash whose base address and properties are given in the `struct_desc` parameter. All replicas will be updated.

The `struct_desc` is table search struct descriptor which was initialized by call to `ezdp_init_table_struct_desc` API and described above in the [Table descriptor initialization](#) section.

The argument `flags` can be used to control the behavior of the add operation and has one option `EZDP_OPPORTUNISTIC`. `EZDP_OPPORTUNISTIC` value indicates that we allow the add operation to be opportunistic meaning giving up when a memory error is detected or the entry is already locked.

The function's return value may indicate either success (0), entry already exists error (EEXIST), hash is full (ENOMEM), or memory error (EIO). The memory error is returned when multiple attempts to reading a hash entry failed with HW memory error.

Modifying, updating or deleting a hash's entries are done in a similar fashion to the operation presented above.

Following example implementing the addition and removal of LAG table entries:

```

static char work_area[EZDP_HASH_HIGH_LEVEL_WORK_AREA_SIZE(sizeof(struct
mac_info), sizeof(struct mac_key))] __ccm_var;

static struct ezdp_hash_struct_desc mac_hash __ccm_var;

static struct mac_info info __ccm_var;
static struct mac_key key __ccm_var;

void add_mac(uint16_t vrf, struct eth_mac * mac, uint8_t port, uint16_t vlan)
{
    uint32_t      retval;

    key.vrf = vrf;
    ezdp_mem_copy(key.mac.byte, mac, sizeof(struct eth_mac));

    info.port = port;
    info.vlan = valn;
    info.in_use = 1;
}

```

```
        ezdp_add_hash_entry(&mac_table, &key, sizeof(key), \
                           &info, sizeof(info), 0, &work_area, sizeof(work_area));

        return;
    }

void del_mac(uint16_t vrf, struct eth_mac * mac) __slow_path_code;
{
    key.vrf = vrf;
    ezdp_mem_copy(key.mac.byte, mac, sizeof(struct eth_mac));

    ezdp_delete_hash_entry(&mac_table, &key, sizeof(key), \
                          0, &work_area, sizeof(work_area));
    return;
}
```

## 4. Advanced Features

### 4.1 DMA and Memory Operations

#### 4.1.1 Memory Subsystem

As described in the [Memory Hierarchy](#) section, the NPS supports EMEM and hierarchical IMEM. These memories can be configured and used for different purposes. The system (i.e. NPS hardware) support for these memory contexts is advanced and highly capable. The sections below describe the main memory properties, how to configure them and how to use them properly.

##### 4.1.1.1 Memory Protection

NPS supports three types of memory protection schemes: in-band ECC, out-of-band ECC, or no protection. All the schemes are supported in EMEM, while IMEM is always protected with the out-of-band ECC scheme.

If no protection is selected, the data will not be protected with ECC. Users should be aware of data corruptions.

ECC protection solves one bit error (single error) and detects two or more bit errors (double error). When an ECC error is detected, an interrupt is sent to the DP application and reported to the CP.

This is done by calling the following functions:

*EZcorChannel\_GetSERInfo()*

*EZcorChannel\_FixSER()*

##### 4.1.1.1.1 In-band ECC Scheme

In the in-band scheme, the data is arranged in 16B chunks where the least significant byte is reserved for ECC purposes (15B of data and 1B of ECC). This has two important implications. First, any operation on such a memory space must be with memory size that is a multiple of 16B. Second, the user must reserve the least significant byte of each 16B for ECC purposes.

The in-band scheme is efficient since the protection bits are inside the data and, therefore, less memory accesses are needed. However, the need to reserve a byte every 16B might harm data continuity.

Furthermore, as described above, using the in-band scheme limits the operations to 16B aligned data of a 16B multiple size.

The example bellow demonstrates a 32 byte in-band ECC protected 32 byte structure:

```
struct in_band_ecc_example {
    unsigned    /* reserved for ECC */ : 8;
    unsigned    data0                  : 24;
    uint32_t    data1;
    uint32_t    data2;
    uint32_t    data3;
    unsigned    /* reserved for ECC */ : 8;
    uint8_t     arr[15];
}
```

FD and LBD structures (see [The frame descriptor](#) and [Linked Buffer Descriptors](#) sections) are defined with one least significant byte reserved, thus can be stored in memories with all protection schemes (i.e. also in memory protected with in-band ECC).

#### 4.1.1.1.2 Out-of-band ECC Scheme

On the other hand in the out-of-band scheme, the protection ECC is maintained outside the data. Each 128B is protected by 16B outside the 128B of data. Frame data stored in BD is stored with the out-of-band scheme for data continuity.

The out-of-band scheme is less efficient than the in-band scheme since more memory accesses are needed to update the ECC which is stored outside the data. However, there is no need to reserve bytes inside the data (supporting continuity). Also, access to the memory is not limited to alignment or size.

#### 4.1.1.2 Memory Allocation

##### 4.1.1.2.1 IMEM

NPS has 16MB internal memory which can be used for storage of:

- Code
- Search databases
- Frame buffers
- Job descriptors
- Variables.

Allocating memory for frame buffers and job descriptors is accomplished by the following CP function:

***EZapiChannel\_ConfigCmd\_SetGeneralParams(..., uint32\_t uiInternalFrameBuffers, uint32\_t uiJobBuffers, ...)***

*uiInternalFrameBuffers* indicates the number of 256B frame buffers and *uiJobBuffers* indicates the number of 32B job descriptors, both allocated in IMEM.

Allocation of memory for code, search databases and DP variables is accomplished by the following CP function:

***EZapiChannel\_ConfigCmd\_SetIntMemSpaceParams*** is used to configure a memory space in IMEM. The main parameters are type and size. The type must be one of the following predefined types. (Each of these types is assigned with a unique ID (MSID), as defined in *ezdp\_memory\_defs.h*):

- EZapiChannel\_IntMemSpaceType\_HALF\_CLUSTER\_CODE  
EZapiChannel\_IntMemSpaceType\_HALF\_CLUSTER\_DATA  
EZapiChannel\_IntMemSpaceType\_HALF\_CLUSTER\_SEARCH
- EZapiChannel\_IntMemSpaceType\_1\_CLUSTER\_CODE  
EZapiChannel\_IntMemSpaceType\_1\_CLUSTER\_DATA  
EZapiChannel\_IntMemSpaceType\_1\_CLUSTER\_SEARCH
- EZapiChannel\_IntMemSpaceType\_2\_CLUSTER\_CODE  
EZapiChannel\_IntMemSpaceType\_2\_CLUSTER\_DATA  
EZapiChannel\_IntMemSpaceType\_2\_CLUSTER\_SEARCH
- EZapiChannel\_IntMemSpaceType\_4\_CLUSTER\_CODE  
EZapiChannel\_IntMemSpaceType\_4\_CLUSTER\_DATA  
EZapiChannel\_IntMemSpaceType\_4\_CLUSTER\_SEARCH
- EZapiChannel\_IntMemSpaceType\_16\_CLUSTER\_CODE  
EZapiChannel\_IntMemSpaceType\_16\_CLUSTER\_DATA  
EZapiChannel\_IntMemSpaceType\_16\_CLUSTER\_SEARCH
- EZapiChannel\_IntMemSpaceType\_ALL\_CLUSTER\_CODE  
EZapiChannel\_IntMemSpaceType\_ALL\_CLUSTER\_DATA  
EZapiChannel\_IntMemSpaceType\_ALL\_CLUSTER\_SEARCH

The type of memory space indicates two important things, namely the type of data that the memory space will contain (code, data or search structures) and the place in the memory hierarchy which affects the latency to the memory space and the IMEM usage. Configuring a memory space closer to the cores enhances performance, but increases usage of IMEM and restricts appearance of the memory to fewer threads. For example, if we configure a memory space in 4\_CLUSTER then only the threads belonging to the same 4 clusters will share this memory space and the configured size will be multiplied by 4 (for all 4 clusters in the chip).

For example, with **EZapiChannel\_ConfigCmd\_SetIntMemSpaceParams** function it might be defined that the per cluster data memory space have 128KB (2MB from IMEM are used):

```
EZapiChannel_ConfigCmd_SetIntMemSpaceParams(index=0, enable=true,
type=1_CLUSTER_DATA, size=128K, ... )
```

After allocating memory, it may be used according to its type:

- Memory allocated for code can be used by locating the function in the relevant FMT slot using the `__mem_fmt_slot_func(n)` qualifier (see the [Code and data section assignments](#) section).
- How to use the allocated memory for a search database (see the [Advanced Search Features](#) section).
- Frame buffers are referred by BDs (described in the [Buffer descriptors](#) section).
- Job descriptors are referred by JIDs (described in [The job ID](#) section).
- Memory allocated for data can be used by locating the variables in the relevant FMT slot using the `__mem_fmt_slot_var(n)` qualifiers (see the [Code and data section assignments](#) section) and accessed through virtual addressing or by accelerators addressing (see the [Referring to Data Allocated Memory](#) section).

#### 4.1.1.2.1 EMEM

EMEM is used for storage of:

- Search databases
- Frame buffers
- Variables.

Allocating memory for frame buffers is accomplished with the same CP function as IMEM allocation:

```
EZapiChannel_ConfigCmd_SetGeneralParams(..., uint32_t uiExternalFrameBuffers, ...)
```

*uiExternalFrameBuffers* indicates the number of 256B frame buffers allocated in EMEM.

Allocation of memory for search databases and variables is accomplished by the following CP function:

**EZapiChannel\_ConfigCmd\_SetExtMemSpaceParams** is used to configure a memory space in EMEM. The main parameters are the size, ECC protection scheme and MSID. The type of memory space should be GENERAL (SEARCH will be discussed in the [Advanced Search Features](#) section), ECC protection scheme is one of the three types described above (none, in-band or out-of-band) and MSID is a user defined ID for the memory space (0..31).

For example, with the **EZapiChannel\_ConfigCmd\_SetExtMemSpaceParams** function it might be defined that MSID 0 on EMEM has in-band protection scheme and size of 128MB:

```
EZapiChannel_ConfigCmd_SetExtMemSpaceParams(index=0, enable=true, type=GENERAL,
size=128M, ecc_type=IN_BAND, ..., msid=0 )
```

Just like IMEM, after allocating memory, it may be used according to its type.

In addition to the memory allocated by the user using the APIs above, 4G of EMEM is also allocated for general purposes (Linux and/or non DP variable), and known as default memory. By default, code

or/and variables are located in default memory (e.g. EMEM), unless special qualifier is provided, as described above.

## 4.1.2 Referring to Data Allocated Memory

Referring to memory may be accomplished by simple virtual addressing or with HW accelerators.

### 4.1.2.1 Virtual Addressing

Virtual addressing is the most convenient and straightforward way to reference a memory. However it is important to understand the limits of such addressing, which does not make use of the DMA accelerators provided by NPS.

It is important to understand that referencing a memory space in IMEM or EMEM with virtual addressing in combination with DMA accelerators (see below) is not straightforward (the translation between virtual address to HW accelerator address is not always known and possible).

### 4.1.2.2 HW Accelerator Addressing

One more forms of accelerated operation supported in the NPS is direct memory access. This acceleration technology allows the programmer to initiate memory operations asynchronously, outside the regular 32 bit memory space reserved for virtual memory, using dedicated hardware blocks without incurring CTOP cycles for the operation.

These memory operations range from a simple offloading of memory copy via DMA to sophisticated atomic operations.

All the accelerated memory operations make use of specialized hardware descriptors, which provide extended memory addressing.

Four types of these memory descriptors are available:

- **Extended address** – 64 bit address format which is comprised of a memory space type, memory space ID and a byte offset in the space. This is the most generic form but provides the least performance.
- **Summarized address** – 32 bit address which is comprised of a memory space type, a memory space ID and an index of an entry inside the memory address. The size of the entry is provided by the operation performed using the address.
- **Buffer Descriptor** – 32 bit address which is comprised of a memory space type and an index. BDs are used for referring to frame buffers in IMEM/EMEM.
- **Job ID** – 32 bit address which is comprised of an index, which is used for referring to JDs in IMEM.

Here is an abbreviated listing of the definitions of these memory descriptors:

```
enum ezdp_mem_space_type {
    EZDP_INTERNAL_MS,
    EZDP_EXTERNAL_MS
};

#define EZDP_EXT_ADDR_ADDRESS_MSB_SIZE 4

struct ezdp_ext_addr {
    enum ezdp_mem_space_type mem_type;
    unsigned msid;
    ...
    unsigned address_msb : EZDP_EXT_ADDR_ADDRESS_MSB_SIZE;
    uint32_t address;
};
```



This descriptor contains the memory type (IMEM/EMEM) and MSID. Notice that the MSID field size is 5 bits in the extended address descriptor as opposed to only 4 bits in the summarized address descriptor (see below). The extended address descriptor also contains a 36 bit (32+4) address. This address is calculated from the beginning of the memory space referred to by the MSID and memory type.

Here is an abbreviated listing of the definitions of the summarized memory descriptor:

```
#define EZDP_SUM_ADDR_ELEMENT_INDEX_SIZE 27

typedef uint32_t ezdp_sum_addr_t;

struct ezdp_sum_addr {
    union {
        ezdp_sum_addr_t raw_data;
        struct {
            enum ezdp_mem_space_type mem_type;
            unsigned msid;
            unsigned element_index : EZDP_SUM_ADDR_ELEMENT_INDEX_SIZE;
        };
    };
};
```

The summarized address descriptor contains the memory type (IMEM/EMEM) and MSID just like the extended address (except the MSID field is 4 bits). It also contains a 27 bit index which is not an address. This difference must be understood when using this descriptor. It might be useful to think of the summarized address as an index of an array. Each element of the array has a size of entry\_size. Therefore, increasing the index in the summarized address by one implies referring to the next element in the array.

It is important to emphasize that both descriptors describe the same memory, only in different ways. The summarized address is just a more compact at the expense of the range of memory that may be addressed.

The following API may be used to convert one address to another.

The general scheme is: (extended address) = (summarized index) \* (entry size).

**void\_t ezdp\_sum\_addr\_to\_ext\_addr(struct ezdp\_ext\_addr \*ext\_addr, ezdp\_sum\_addr\_t sum\_addr, uint32\_t entry\_size)**

*ezdp\_sum\_addr\_to\_ext\_addr* converts the summarized address to the equivalent extended address. The resulting extended address is calculated in the *ext\_addr* out argument. As described above, any use of a summarized address must state the entry size of the address.

**ezdp\_sum\_addr\_t\_t ezdp\_ext\_addr\_to\_sum\_addr(struct ezdp\_ext\_addr \*ext\_addr, uint32\_t entry\_size)**

*ezdp\_ext\_addr\_to\_sum\_addr* converts the extended address to the equivalent summarized address according to the provided entry size. The resulting summarized address is returned. As described above, any use of a summarized address must state the entry size of the address.

### 4.1.3 Memory Operations

NPS supports memory operations that allow memory access in any size (up to 1K) and any alignment. These memory operations range from a simple offloading of memory copy via DMA to sophisticated atomic operations.

The DP API is generally divided into three types of DMA operations:

- Load – copy data from IMEM/EMEM to CMEM.
- Store – copy data from CMEM to IMEM/EMEM
- Copy – copy data from IMEM/EMEM to IMEM/EMEM

Some operations are further optimized for specific data sizes (16B and 32B) and are recommended for enhanced performance. These optimized operations have size and alignment limitations that the user should be aware of. 16B operations are limited to sizes that are a multiple of 16B bytes and in 16B alignment. 32B operations are limited to sizes that are a multiple of 32B bytes and in 32B alignment.

Here are some of DMA operations. Note the limitations of each operation.

**`uint32_t ezdp_copy_data_by_ext_addr(struct ezdp_ext_addr __cmem *dst_ptr, struct ezdp_ext_addr __cmem *src_ptr, uint32_t size, uint32_t flags)`**

`ezdp_copy_data_by_ext_addr` invokes a DMA to copy data of the provided size from the address pointed by `src_ptr` to the address pointed by `dst_ptr`. These pointers must be 8B aligned in CMEM. The data will be copied in atomic chunks of 128B (for EMEM) or 32B (for IMEM) and a 16-bit checksum of the data will be returned.

When using this operation with a memory space with the out-of-band ECC protection scheme, it is possible to launch the operation with the `EZDP_MEMORY_FLAG_OVERWRITE` flag. In general, if you know the data you are storing in one operation comprises all the data of value in all of the 128B sized lines you are writing, the flag `EZDP_MEMORY_FLAG_OVERWRITE` indicates to the hardware that it can ignore any data previously pre-existing in any ECC line that contains the data being written. This allows the hardware to use a more permissive algorithm which only calculates the ECC checksum on the new data and writing it into memory, without the need to read back and merge the old data. This can provide significant performance and power optimization, where applicable. Care must be taken however, as use of this flag can cause inadvertent data corruption if used incorrectly.

**`uint32_t ezdp_load_data_from_ext_addr(void __cmem *dst_ptr, struct ezdp_ext_addr __cmem *src_ptr, uint32_t size, uint32_t flags)`**

`ezdp_load_data_from_ext_addr` invokes a DMA to copy data of the provided size from the address pointed by `src_ptr` to the address pointed by `dst_ptr` on CMEM. The source address must be 16B aligned. The data will be copied and the first 4B will be returned.

**`uint32_t ezdp_load_16_byte_data_from_ext_addr(void __cmem *dst_ptr, struct ezdp_ext_addr __cmem *src_ptr, uint32_t flags)`**


**`uint32_t ezdp_load_32_byte_data_from_ext_addr(void __cmem *dst_ptr, struct ezdp_ext_addr __cmem *src_ptr, uint32_t flags)`**

`ezdp_load_16/32_byte_from_ext_addr` is an optimized operation that invokes a DMA to copy 16B/32B from the address pointed by `src_ptr` to the address pointed by `dst_ptr` on CMEM. The source address must be 16B aligned. The data will be copied and the first 4B will be returned.

**`uint32_t ezdp_load_16_byte_data_from_sum_addr(ezdp_sum_addr_t sum_addr, uint32_t entry_size, uint32_t offset, uint8_t __cmem *ptr, uint32_t flags)`**

**`uint32_t ezdp_load_32_byte_data_from_sum_addr(ezdp_sum_addr_t sum_addr, uint32_t entry_size, uint32_t offset, uint8_t __cmem *ptr, uint32_t flags)`**

`ezdp_load_16/32_byte_from_sum_addr` is an optimized operation that invokes a DMA to copy 16B/32B from the address pointed by `sum_addr` to the address pointed by `ptr` on CMEM. As described above, any use of a summarized address must state the entry size of the address. The data will be copied from the requested offset in the entry and the first 4B will be returned.

 All APIs are implemented in the similar way, for more details refer to the “DMA Operations (ezdp\_dma.h)” section in the *EZdp Reference Manual*.

## 4.1.4 Atomic Operations

NPS supports a wide variety of atomic operations that are activated through special instruction set extensions to the base architecture. During these operations a thread can simultaneously read a location and write to it in the same bus operation. This prevents any other thread from writing or reading this memory until the operation is complete.

These operations may serve to implement semaphores, counting semaphores, etc. For example, here is an implementation of a simple lock that resides on MSID0 in EMEM:

```
#define UNLOCKED 0
#define LOCKED 1

/* Defining an 8-bit lock on address 0x0 on MSID0 on EMEM. The lock address is
located on shared CMEM and therefore shared by all threads in a core */

struct ezdp_ext_addr lock __shared_cmem_var;
lock.mem_type = EZDP_EXTERNAL_MS;
lock.msid = 0x0;
lock.address_msb = 0x0;
lock.address = 0x0;

/* initializing the lock - should be done at initialization of the system */
ezdp_atomic_write8_ext_addr(&lock, UNLOCKED);

...

/* try to lock (compare value in lock to UNLOCKED, if succeed store LOCKED in the
lock). */
if(ezdp_atomic_cmpxchg32_ext_addr(UNLOCKED, &lock, LOCKED) == UNLOCKED)
{
    // lock acquired */
}
else
{
    // lock is already taken by another thread.
}

/* release lock by simple write */
ezdp_atomic_write32_ext_addr(&lock, UNLOCKED);
```



Atomic operations are not allowed for the in-band EEC protection scheme.

All atomic operations are listed in the *EZdp Reference Manual*. The table below summarizes the some basic main APIs for atomic operations:

API Name	Description
ezdp_atomic_read32_ext_addr ezdp_atomic_read32_sum_addr	Atomically read a 32 bit value from an extended or summarized address.
ezdp_atomic_write32_ext_addr ezdp_atomic_write32_sum_addr	Atomically write a 32 bit value to an extended or summarized address.
ezdp_atomic_xchg32_ext_addr ezdp_atomic_xchg32_sum_addr	Atomically exchange a 32 bit value in an extended or summarized address.
ezdp_atomic_cmpxchg32_ext_addr ezdp_atomic_cmpxchg32_sum_addr	Atomically compare and exchange a 32 bit value in an extended or summarized address.

API Name	Description
ezdp_atomic_cmpxchg32_ext_addr ezdp_atomic_cmpxchg32_sum_addr	Atomically compare and exchange a 32 bit value in an extended or summarized address.
ezdp_atomic_read_and_inc32_ext_addr ezdp_atomic_read_and_inc32_sum_addr	Atomically read and increment a 32 bit value in an extended or summarized address.

***uint32\_t ezdp\_atomic\_read32\_ext\_addr(struct ezdp\_ext\_addr \*addr)***

*ezdp\_atomic\_read32\_ext\_addr* reads the 32-bit value stored at the extended address pointed by *addr*.

***uint32\_t ezdp\_atomic\_read32\_sum\_addr(ezdp\_sum\_addr\_t addr)***

*ezdp\_atomic\_read32\_sum\_addr* is similar to *ezdp\_atomic\_read32\_ext\_addr* except the address of the required 32 bits is a summarized address.



All APIs are implemented in the similar way, for more details refer to the “Atomic Operations (ezdp\_atomic.h)” section in the *EZdp Reference Manual*.

## 4.2 Controlling the Jobs Pipeline

The application running on the NPS clusters handles and processes jobs. For a high level discussion of jobs and their use see [Working with Jobs](#) in the previous chapter.

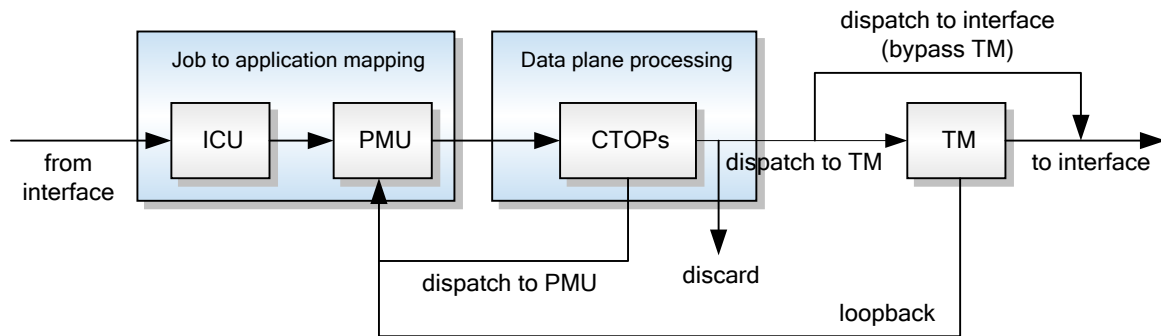
When a frame is received by a DP application it is designated as part of a *job*, which is a single quantum of work that needs to be performed by one of the NPS CTOPs. Handling of jobs in a DP application is accomplished with a dedicated API (see the [Handling Jobs in Data Plane Applications](#) section).

All frames entering the NPS SoC are mapped to one of the NPS clusters. The mapping of the frame to cluster is done by ICU classification and PMU schedule configuration (see the [Mapping Job to Application](#) section).

Handling the job is up to the application, but the result of any job processing must be one of the following:

- Discard job, i.e. job is not needed and resources are to be discarded.
- Dispatch back to PMU, i.e. job needs further processing, perhaps by another application.
- Dispatch to an interface, i.e. job is transmitted to an output port. (This may be done through TM or directly to the output interface.)

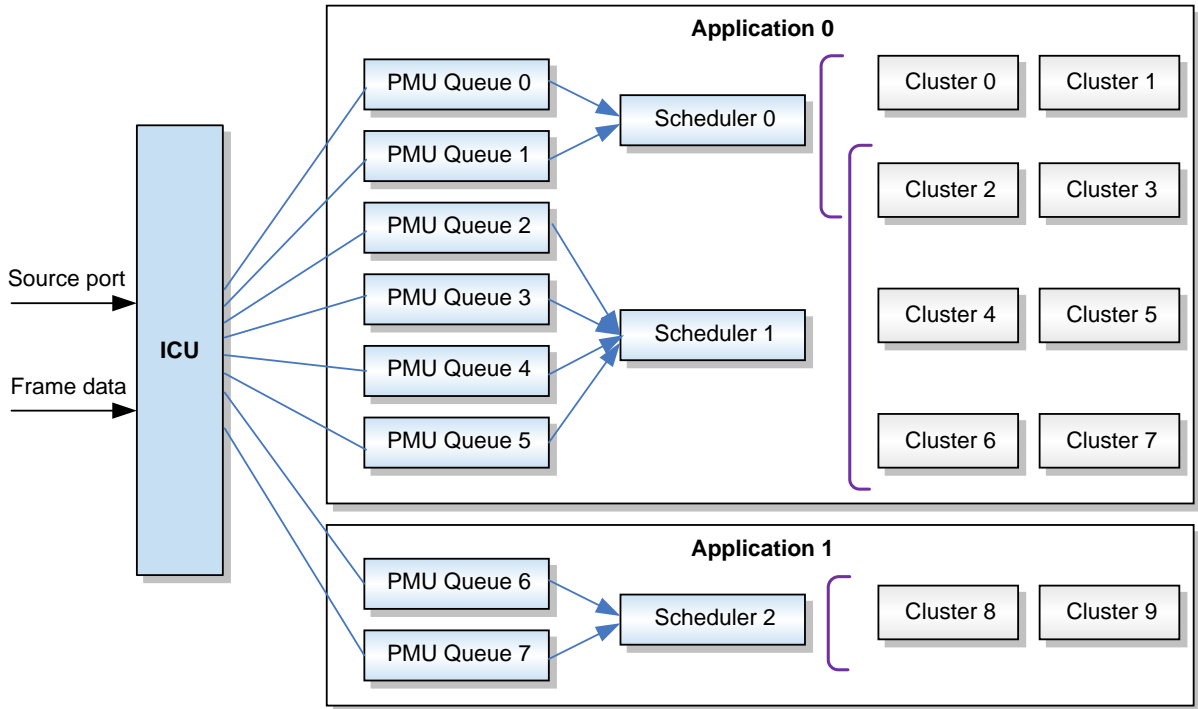
**Figure 11. Handling the job by the application**



## 4.2.1 Mapping Job to Application

Mapping of jobs to an application is accomplished by classifying each incoming frame by the Input Classification Unit (ICU) to one of the 256 Processor Management Unit (PMU) queues. The PMU then schedules the jobs to the different clusters while maintaining ordering between the frames processed in parallel. Applications running on the cluster handle these jobs.

**Figure 12. Mapping of jobs to an application**



### ICU and Classification Profiles

The ICU classification is accomplished based on the source network interface and predefined fields in the frame headers. The ICU profile defines which frame data should be used for frame classification. The output of ICU process is CoS and a hash value.

The example below defines CoS as VLAN priority (bits 113-115) and hash value is 4 LSB of VLAN tag (bits 124-127), using **EZapiICU\_ConfigCmd\_SetExternalClassificationProfile** API:

```
EZapiICU_ConfigCmd_SetExternalClassificationProfile(side=0, profile=0,
hi_gig_enable=false, cos_offset=113, cos_num_bits=2, hash_offset=124,
hash_num_bits=4)
```

After defining a classification profile there is a need to assign it to an interface. This is done by the **EZapiICU\_ConfigCmd\_SetEthParams** function. For example, assigning the above profile to interface 0 is performed by:

```
EZapiICU_ConfigCmd_SetEthParams(side=0, if_engine=0, eth_if_type=10GE, if_number=0,
..., external_classification=true, external_classification_profile=0, ...)
```

Both CoS and hash values are later used in order to select the PMU queue and an additional CoS value is written in frame descriptor (see [The frame descriptor](#) section).

### Mapping Jobs to PMU Queues

The PMU queue selection is based on per CoS configuration and ICU output hash value.

**cos\_params\_0\_start\_pmu\_queue** defines the start PMU queue for CoS 0 and

**cos\_params\_0\_num\_pmu\_queues** defines the number of PMU queues to use per CoS. Both

configuration values together with the ICU output hash value determine the PMU queue in the following way: **cos\_params\_0\_start\_pmu\_queue** + ICU output hash value % **cos\_params\_0\_num\_pmu\_queues**. Configuring the start PMU queue and number of PMU queues per CoS is done by a call to **EZapiChannel\_ConfigCmd\_SetEthRXChannelParams** API.

For example define 16 different PMU queue per each CoS.

```
EZapiChannel_ConfigCmd_SetEthRXChannelParams(side=0, if_engine=0, eth_if_type=10GE,
if_number=0, rx_channel=0, logical_id=0, ..., cos_params_0_start_pmu_queue=0,
cos_params_0_num_pmu_queues=16, cos_params_1_start_pmu_queue=16,
cos_params_1_num_pmu_queues=16,
cos_params_2_start_pmu_queue=32, cos_params_2_num_pmu_queues=16,
cos_params_3_start_pmu_queue=48, cos_params_3_num_pmu_queues=16, ...)
```



Logical ID, CoS and PMU queue (side and queue) are written to the job descriptor in (see the [Job Information](#) section).

### Mapping PMU Queues to Clusters

Mapping the jobs of the PMU queue to the cluster is done using the PMU job scheduler. Each of the 256 PMU queues is assigned, via configuration, to one of several PMU job schedulers. This is done with the following function:

**EZapiChannel\_ConfigCmd\_SetPMUQueueParams(side, queue, priority, weight, scheduler, enable\_sequence\_number, job\_limit\_profile)**

Each queue is given a *priority* from 0 to 3 and a *weight* from 1 to 256 by which the *scheduler* may give different priorities to different queues. *enable\_sequence\_number* and *job\_limit\_profile* are discussed later ([Sequence numbering](#) and [Outstanding job limit](#) sections).

Each PMU job scheduler in turn is associated, via configuration, with a group of NPS clusters, which will receive jobs from the specified PMU scheduler. Different schedulers may serve different or overlapping groups of CTOPs (as depicted in the figure above).

**EZapiChannel\_ConfigCmd\_SetPMUSchedulerParams(side, scheduler, enable, application, cluster\_bitmap, commit\_jobs, max\_jobs)**

Each scheduler serves an *application* that runs on a group of NPS clusters, determined by *cluster\_bitmap*. In order to avoid starvation between two schedulers when both of them assign jobs to the same cluster *max\_jobs* and *commit\_jobs* should be configured. The *commit\_jobs* threshold is the number of jobs that the scheduler is guaranteed to be able to assign for CTOP processing. The maximum jobs threshold is the upper limit of the jobs this scheduler may assign. The delta between maximal and committed jobs is not guaranteed.



The sum of all committed jobs threshold MUST be at most the number of threads which handle it (number of cluster the scheduler serve \* 16 (cores) \* number of threads per core).

For example, voice frames have very high priority but to prevent starvation of other frame types, voice jobs might be scheduled by a separate scheduler that is mapped to dedicated clusters.

A CTOP may receive jobs from any queue associated with any job scheduler which is a part of its CTOP application group. Within each job scheduler, queues are scheduled according to their priority and weight in a weighted round robin fashion.

Using ICU classification, PMU queue mapping and scheduling to different NPS clusters allows breaking port ordering domains to application order domains. More complex classification can be done by SW for more info see the [Pipeline vs. Run to Completion](#) section.

#### 4.2.1.1 Order restoration

The PMU schedules jobs to CTOP threads in parallel, without taking into account the incoming frame order. It is permissible for a late frame to be processed prior to an earlier frame. In a similar fashion, due to the parallel nature of processing, it is permissible for a frame to finish processing on a CTOP prior to processing being finished for an earlier frame.



For stateful features where processing order is important stateful dispatcher mechanism such as the one developed by the [Stateful Flow Table \(SFT\)](#) can be used, for further information see the SFT specification.

However, when a CTOP thread has indicated to the PMU that it has finished the processing of a job, the PMU will by default suspend the further processing of a job until such time as all jobs preceding it in the same PMU queue have finished processing and have advanced to the next processing stage, such as transmission via TM queue or dispatching to a different PMU queue as the case may be.

This in effect restores the order of frames so that frame progress via each stage of processing, including transmission, is done according to the order in the PMU queue. Thus, each PMU queue functions also as an “ordering domain” for the frames associated in the jobs which are queued in it. Dispatching a job to a different queue therefore creates a new frame order.

## 4.2.2 Handling Jobs in Data Plane Applications

### 4.2.2.1 Getting a job

The journey of a job in a DP application always starts with receiving a job from PMU.

The simplest way to receive a new job is to call the `ezdp_receive_job()` API function:

```
void ezdp_receive_job (ezdp_job_id_t __cmem * jobh_ptr, struct
                        ezdp_job_desc __cmem * jd_ptr, uint32_t flags)
```

The `ezdp_receive_job()` function requests a new job assignment from the PMU, stores the job ID of the new job at the CMEM address provided by the `jobh_ptr` parameter and then copies the job descriptor from its location in IMEM into the CMEM at the address provided by the `jd_ptr` parameter. Once the function returns, the job ID and the job descriptor are available in the CMEM at the supplied addresses and the frame processing can begin.

After a job was received, its type can be determined by examining the `logical_id` field. The `logical_id` field can serve as an identifier for the information of the kind of processing required by the software for this particular job. The logical ID might be used as a key in a table lookup to determine the service information required.

```
struct port_info
{
    uint8_t type;
    // more information about port/service
};
struct port_info    port_info    __cmem_var;

...
/* Read job from PMU */
ezdp_receive_job(&jid, &jd, 0);

/* Lookup port information. Use logical ID as a key */
```



```

uint8_t key = jd.frame_desc.logical_id;
ezdp_lookup_table_entry(PORT_INFO_TABLE_DESC, key, &port_info, sizeof(struct
port_info), 0);

/* Process job according to port type */
switch(port_info.type)
{
case EXTERNAL_INTERFACE:
    // process jobs that came from an external interface
    break;
case USER:
    // process jobs
    break;
case TIMER:
    // process timer jobs
    break;
// more cases ...
}
...

```

#### 4.2.2.2 Job Information

Once the job type is known, the receive-side job information can provide valuable information for processing.

Here is an abbreviated listing of the receive side job information data structure:

```

struct ezdp_job_rx_info {
...
    struct {
        union {
            struct ezdp_job_rx_interface_info      interface_info;
            struct ezdp_job_rx_loopback_info        loopback_info;
            struct ezdp_job_rx_confirmation_info    confirmation_info;
            struct ezdp_job_rx_timer_info           timer_info;
            struct ezdp_job_rx_user_info            user_info;
        };

        uint16_t gross_checksum;
        uint16_t seq_number;
...
        unsigned seq_number_valid;
        unsigned side;
        unsigned source_queue;
    };
...
};

```

The receive-side job information structure is divided into two sections: a section containing [job type specific information](#) and a section containing [job type agnostic information](#).

##### 4.2.2.2.1 Job type specific receive-side information

The first part of the receive side job information structure contains information which is specific to the job type. As such, each of the five types of jobs supported in the NPS provides here a different structure which contains the job type specific information. This overlay structure is represented as a C union, with one member per job type.

The NPS supports the following job types:

**Interface job** – a job containing a frame received from one of the NPS external network interfaces. This is the most common job type.

**Loopback job** – a job containing a frame which was created in one of the NPS internal hardware engines, such as one replica of a multicast frame.

**Confirmation job** – a job containing a confirmation message of the transmission of a single frame, such as those used in the IEEE 1588 network time synchronization protocol.

**Timer job** – a job signifying a hardware timer has elapsed.

**User Information job** – a job containing a programmer defined software message.

#### 4.2.2.2.1.1 Interface job receive-side information

The interface job receive-side information structure holds ancillary information provided by the NPS regarding jobs representing frames received from one of the NPS network interfaces.

The information may be used by the data plane application to gain additional information about the frame and the system state.

Here is an abbreviated definition of this structure:

```
enum ezdp_congestion_level {
    EZDP_LOW_LEVEL,
    EZDP_MEDIUM_LEVEL,
    EZDP_HIGH_LEVEL,
    EZDP_CRITICAL_LEVEL
};

struct ezdp_job_rx_interface_info {
...
    struct {
        enum ezdp_congestion_level pmu_queue_congestion_level;
        enum ezdp_congestion_level job_congestion_level;
        enum ezdp_congestion_level emem_buf_congestion_level;
        enum ezdp_congestion_level imem_buf_congestion_level;
        enum ezdp_congestion_level global_congestion_level;
        unsigned imem_buf_count;
...
        unsigned icu_succ_parsing_flag;
        unsigned truncation_flag;
        unsigned crc_ok_flag;
        unsigned crc_checked_flag;
        uint8_t timestamp_sec;
        uint32_t timestamp_nsec;
    };
...
};
```

Here is the description of the various fields of this information structure:

- **PMU Queue Congestion Level** – the congestion level of the specific PMU queue that the job is queued in at the job queuing time.
- **Job Congestion Level** – the overall congestion level of all the jobs scheduled by the system at the job queuing time.
- **External Memory Buffers Congestion Level** – the congestion level of external memory frame buffers at the job queuing time.
- **Internal Memory Buffers Congestion Level** – the congestion level of external memory frame buffers at the job queuing time.
- **Global Congestion Level** – an indication of the global overall system congestion state at the job queuing time.
- **Internal Memory Buffers Count** – the number of consecutive frame buffers, starting from the first frame buffer, which were allocated in IMEM. All other buffers, if any, were allocated in EMEM.

- **Input Classification Unit Successful Parsing Flag** – a Boolean flag indicating that the ICU unit was successful at parsing the frame.
- **Truncation Flag** – a Boolean flag indicating that the flags were truncated during reception.
- **CRC OK Flag** – a Boolean flag indicating that the frame Ethernet CRC header is correct. The flag is only valid if the *CRC Checked Flag* is set.
- **CRC Checked Flag** – a Boolean flag indicating that the frame Ethernet CRC header was checked on reception.
- **Time Stamp Seconds** – the seconds portion of a timestamp counter that can be used to calculate an IEEE 1588 precision time protocol timestamp for the frame. The timestamp is sampled by the Rx MAC when the frame is speculated to be within the IEEE 1588 length range (based on frame length) to minimize overheads. The field value is only valid when the frame descriptor *time\_stamp* flag is on.
- **Time Stamp Nanoseconds** – the nanosecond portion of the timestamp counter. The field value is only valid when the frame descriptor *time\_stamp* flag is on.

#### 4.2.2.2.1.2 User job receive-side information

The user job receive-side information structure holds user information that was stored by previous processing.

The information may be used by the data plane application to pass information about the frame and the system state in pipeline processing.

Here is an abbreviated definition of this structure:

```
struct ezdp_job_rx_user_info {
...
    struct {
        uint32_t user_data_info0;
        uint32_t user_data_info1;
    };
...
};
```

#### 4.2.2.2.2 Job type agnostic receive-side information

The general section of the receive-side job information structure which applies to any job received, has the following fields:

- **Gross Checksum** – an Internet (IP) checksum of the entire frame data, starting at the first byte of the MAC header and up to the end of the frame, excluding any bytes stripped by the MAC, if any (such as the 4 byte Ethernet CRC, if the MAC was configured to omit it). The field value is only valid if the *Checksum* flag in the frame descriptor is set. This field can be used to calculate a packet's TCP checksum by subtracting such fields, such as MAC header, which are not included in the TCP checksum.
- **Sequence Number** – the ordinal number representing the position of the job in the PMU queue, relative to other jobs in the queue. The field value is only valid if the *Sequence Number Valid Flag* field is true.
- **Sequence Number Valid Flag** – a flag indicating that the *Sequence Number* field is valid.
- **Side** – which of the two PMU instances, one for each side of the NPS, handles this job. The job will be returned, when done, to the handling PMU.
- **Source Queue** – the Processor Management Unit queue that the job belongs to. The application may change this value to indicate that the job is to be moved to a different queue of the PMU.

### 4.2.3 Transmit a Job

There are several possible ways to end the processing of a frame: transmitting the frame to an interface (through TM or directly to the interface), dropping the frame, or transferring the frame to another PMU queue.

#### 4.2.3.1 Transmitting a frame to an interface through TM

In most real world applications, the frame will not be sent to the channel directly. Instead the frame will be transmitted through the TM which provides virtual queuing, QoS enforcement and channel selection service.

Dispatching a job to TM is done with the following API:

```
void ezdp_send_job_to_tm (ezdp_job_id_t __cmem * jobh_ptr, struct
                           ezdp_job_desc __cmem * jd_ptr, uint32_t side, uint32_t flags)
```

The `ezdp_send_job_to_tm()` function dispatches a job through TM and ends the handling of the job. The information for transmission must be stored in `tx_info` of `jd_ptr` (for more information see the [Traffic Management](#) section). `side` indicate to which TM to send the packet. `EZDP_ALLOW_REORDER` flag may be used in the `flags` argument to allow jobs to be reordered.

The `jobh_ptr` should contain the address in CMEM where the job ID is stored.

#### 4.2.3.2 Transmitting a frame to an interface (TM bypass)

To transmit the frame through one of the system's network interfaces, one needs to send the frame to one of the transmit channels of one of the network interfaces, where it will be queued for transmission according to the frame's designated CoS and the channel's flow control status.

This is done by calling the `ezdp_send_job_to_interface()` API function.

```
void ezdp_send_job_to_interface (ezdp_job_id_t __cmem * jobh_ptr, struct
                                  ezdp_job_desc __cmem * jd_ptr, uint32_t side, uint32_t output_channel, uint32_t flags)
```

The `ezdp_send_job_to_interface()` function dispatches a job to a specified interface and ends the handling of the job. The `side` and `output_channel` arguments together choose an output channel to be used for transmission. The frame will be queued for transmission via the network interface the channel it was configured to use.

The `jobh_ptr` should contain the address in CMEM where the job ID is stored.

The argument `flags` can be used to control network order restoration behavior of the PMU for the frame: a value of zero requests the default behavior of transmitting the frame in the exact order it was received, while a value of `EZDP_ALLOW_REORDER` allows a more permissive behavior.

For example, transmitting the job to channel 1 on side 0 without change of ordering we can use:

```
...
ezdp_send_job_to_interface(&job, &job_desc, 0, 1, 0);
...
```

#### 4.2.3.3 Discarding a job

Some DP applications may need to discard jobs. For example, when the frame data is corrupted or when the frame payload was processed and the job is not needed anymore.

Discarding a job is accomplished with the following API:

```
void ezdp_discard_job (ezdp_job_id_t __cmem * jobh_ptr, struct ezdp_job_desc __cmem * jd_ptr)
```

The `ezdp_discard_job()` function frees all job resources (FD buffers and job ID) and ends the handling of the job.

The `jobh_ptr` should contain the address in CMEM where the job ID is stored.

The following is a usage example of the above interface:

```
...
ezdp_discard_job(&job, &job_desc);
...
```

## 4.2.4 Pipeline vs. Run to Completion

A job might be processed in one of two ways. All reference applications provided are designed in *run to completion* architecture. *Pipeline* architecture is generally viewed as less efficient. The text below introduces both.

### 4.2.4.1 Run to completion

In run-to-completion mode, the job is processed by one or several threads until the processing is complete (either by discard or transmission to an output channel). In this mode, the job will never return to the CTOPs again.

Some DP applications might need, however, to order a job in a PMU queue without waiting for completing its processing. For example, this might be needed when implementing SW classification to ordering domain, i.e. when the incoming traffic is classified to several PMU queues, but processing of job continues.

Updating a job is achieved by the following API:

**void ezdp\_update\_job\_queue (ezdp\_job\_id\_t \*\_\_cmem jobh\_ptr, struct  
ezdp\_job\_desc \_\_cmem \*jd\_ptr, uint32\_t side, uint32\_t target\_queue, uint32\_t flags)**

The requested PMU queue needs to be updated in the *source\_queue* field of the job descriptor in the CMEM, which the *jd\_ptr* parameter should contain the address of.

The `jobh_ptr` should contain the address in CMEM where the job ID is stored.

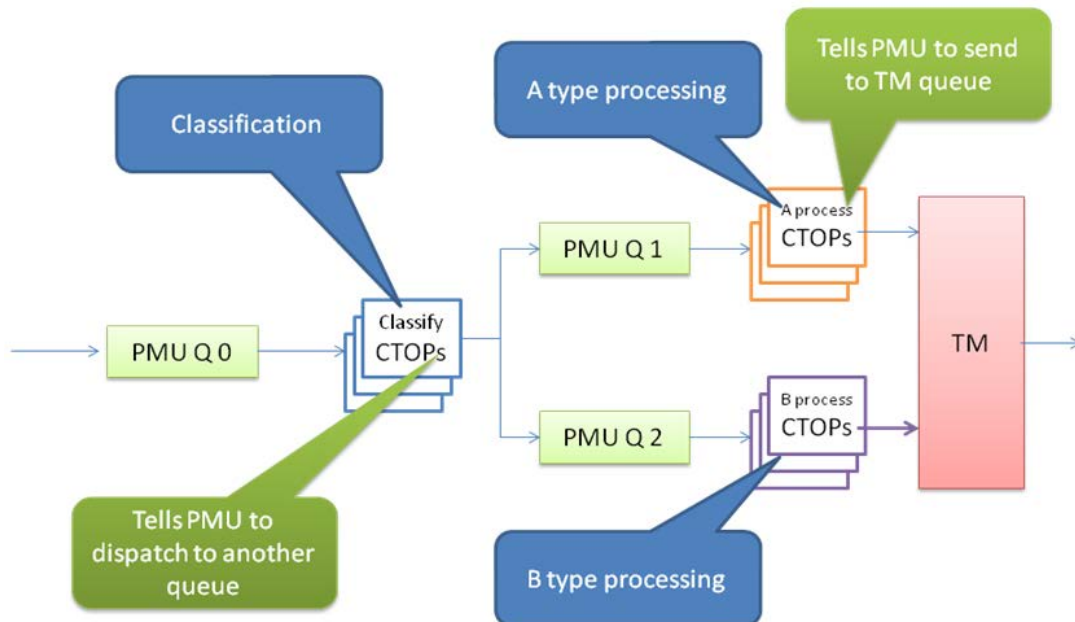


Only one update is possible.

#### 4.2.4.2 Pipeline

Another possible mode is the pipeline processing mode. In this mode, the job is processed in several SW stages where each stage (except the last) ends with dispatching the job to the next PMU queue (either by dispatching to TM loopback or by dispatching to a PMU queue). In this way a software architecture is created in which separate packet processing applications are organized in a SW pipeline with each application being one stage in the pipeline.

**Figure 13. Example of packet processing applications organized in a software pipeline**



In one such example of such an architecture, a first stage pipeline packet processing application performs classification on incoming frames and determines the right application to handle them and then forwards the frame to the appropriate application as a second stage of the software pipeline. Implementing such a scheme may be done by dispatching the job to another PMU queue, where the processing of the classification stage of the pipeline ends in a PMU dispatch.

Dispatching a job to a PMU queue is done in much the same fashion as transmitting a job to an interface, as described in the [Transmitting a frame](#) section above.

```
void ezdp_send_job_to_queue (ezdp_job_id_t __cmem * jobh_ptr, struct  
ezdp_job_desc __cmem * jd_ptr, uint32_t side, uint32_t target_queue, uint32_t flags)
```

The `ezdp_send_job_to_queue()` function dispatches a job to a specified PMU queue and ends the handling of the job.

The `jobh_ptr` should contain the address in CMEM where the job ID is stored.

For example, sending the job to queue 0 in side 1 is done by:

```
...  
ezdp_send_job_to_queue(&job, &job_desc, TARGET_SIDE, TARGET_QUEUE, 0);  
...
```

## 4.2.5 Creating a Job

Some DP applications might need to dispatch a job without having the job in the first place that was received from PMU. For example, a TCP implementation might need to send a SYN segment at the beginning of a data transfer session. In such a case, the application will have to allocate a new job, create its content (namely the SYN segment) and then dispatch it to the appropriate destination.

SW threads can allocate new empty jobs (one or more) by using either of the following APIs:

**ezdp\_job\_id\_t ezdp\_alloc\_job\_id (uint32\_t budget\_id)**

**ezdp\_job\_id\_t ezdp\_alloc\_multi\_job\_id ( ezdp\_job\_id\_t \_\_cmem \* jobhs\_ptr,  
uint32\_t budget\_id, uint32\_t num\_of\_jobs)**

The *ezdp\_alloc\_job\_id()* and *ezdp\_alloc\_multi\_job\_id()* functions allocate new job IDs.



These functions may fail when the PMU does not have free job IDs to allocate. It is mandatory to validate returned job ID with the following API:

**bool ezdp\_job\_alloc\_failed (ezdp\_job\_id\_t ret)**



Assigning the job to an ordering domain may not be done with the *update\_job\_queue* API above. This may be done only by dispatching the new job to a PMU queue in a pipeline fashion.

The PMU does not initialize the JD database and it is SW's responsibility to initialize the JD in IMEM when dispatching the job.

The following is a usage example of the above interface:

```
ezdp_job_id_t jid __cmem_var;
struct ezdp_job_desc jd __cmem_var;

#define BD_BUDGET 0    // budget for BD frame creation
#define SYN_SIDE 0
#define SYN_CHANNEL 0

jid = ezdp_alloc_job_id(NEW_JOB_BUDGET);
if(ezdp_job_alloc_failed(jid))
{
    // allocation failed
}
else
{
    jd.frame_desc.job_budget_id = NEW_JOB_BUDGET;
    create_syn_frame(&jd.frame_desc, BD_BUDGET);

    // transmit frame to interface
    ezdp_send_job_to_interface(&jid, &jd, SYN_SIDE, SYN_CHANNEL, 0);
}
...
```



It is important to understand that a newly created job does not belong to any ordering domain until it is dispatched to a queue/TM/interface.

#### 4.2.5.1 Sequence numbering

The PMU can assist in generating sequence numbers for a flow of packets. The packet sequencing allows in order flow processing of the traffic.

Each PMU queue maintains a job sequence counter that is incremented with each new job entering the queue. The incoming job is assigned with the sequence number at the time it is enqueued into the PMU queue according to its receive order. This sequence number is stored in the JD (rx\_info).



The sequence number is valid only when seq\_number\_valid flag is ON (in JD).

The sequence number allocation can be enabled for all jobs of a PMU by assigning TRUE to the *enable\_sequence\_number* flag in **EZapiChannel\_ConfigCmd\_SetPMUQueueParams** API.

#### 4.2.5.2 Outstanding job limit

Each PMU queue can limit the number of outstanding jobs. This is needed in applications that maintain a limited number of resources (like SW pool indexes) and locks are to be avoided.

This is done by assigning the PMU queue with a job limit profile with the following API:

**EZapiChannel\_ConfigCmd\_SetPMUJobLimitProfileParams (side, profile, job\_limit)**

First, a job limit profile must be defined. For example, the following command defines a job limit profile with ID 0 on side 0 which limits the outstanding jobs to 128 jobs:

**EZapiChannel\_ConfigCmd\_SetPMUJobLimitProfileParams (side=0, profile=1, job\_limit=128)**

The job limit profile may be assigned a PMU queue by setting *job\_limit\_profile* parameter in **EZapiChannel\_ConfigCmd\_SetPMUQueueParams** API to desired profile number (the profile must be defined on the same side as the PMU queue). For example:

**EZapiChannel\_ConfigCmd\_SetPMUQueueParams (... , job\_limit\_profile=1)**



## 4.3 Timer Mechanisms

Many packet processing applications require timer functionality. The NPS provides this functionality in a scalable fashion using the PMU timer job mechanism. This mechanism generates jobs just like network interface jobs and queues them into a PMU queue.

Up to 32 timers (16 timers per side) may be enabled to generate a predetermined number of timer event jobs at a predetermined rate. Timer event jobs are handled just like network interface jobs by any CTOP core based on the PMU configuration and, thus, enjoy the same scalable workload distribution mechanism, ordering, sequence numbering, outstanding job limit, etc.

Each PMU timer is programmed with a time period and the number of events to generate within each period. During each period, the PMU timer will generate the requested number of timer event jobs, as evenly distributed as possible (best effort), and will queue to them into a PMU queue as configured. Each such timer event job will carry a unique sequential identifying ID number. At the end of each period the process will begin anew.

Limitations:

- Period must be at least 20ns.
- Minimal interval between events is 10ns. Therefore the number of jobs in each period must be less than period/10 (one tenth of the period).

Configuring a timer is achieved with the following API:

*EZapiChannel\_ConfigCmd\_SetPMUTimerParams* (*side*, *timer\_id*, *enable*, *logicalID*, *budgetID*, *PMUQueue*, *numOfJobs*, *nanoSecPeriod*, *secPeriod* )

*EZapiChannel\_ConfigCmd\_SetPMUTimerParams()* is used to configure up to 16 timers per side. Each timer is identified with a *timer\_id* that will appear in the job information of the timer jobs created by that timer. Each of these timers is assigned to a *PMUQueue* and *side*. The *logicalID* will appear in the job descriptor (see [The job descriptor](#)). *nanoSecPeriod* and *secPeriod* determine the period of the timer in seconds and 10-nanoseconds granularity. The timer will generate *numOfJobs* timer jobs per period.

### 4.3.1 Handling Timer Jobs in DP Application

Timer event jobs received by CTOP cores are recognized according to the logical ID assigned via configuration (see the [Handling Jobs in Data Plane Applications](#) section ). Each timer event job includes timer event receive-side job information.

The following is an abbreviated list of the timer event job receive-side job information structure which accompanies each timer event job:

```
struct ezdp_job_rx_timer_info {
    uint8_t timer_id;
    uint32_t event_id;
};
```

The *timer\_id* field designates the timer which generated the timer event job (as was configured by CP) whereas the *event\_id* field designates the event ID within the timer period (an ID between 0 and *numOfJobs* - 1). The frame descriptor will contain a logical ID and job budget ID as determined by configuration and without frame data (all other fields of frame descriptor are zeroed and invalid).

Timer jobs, like any other job, must be discarded, moved to another PMU queue or transmitted (see [Controlling the Jobs Pipeline](#)).



Since a timer job contains no data, the job can be transmitted only if data will be added to the frame (and *frame\_type* must be changed accordingly).

In the following example a timer is configured that generates 64 event jobs every 1 millisecond and invokes a TCP tick function (event ID is sent as flow ID to handle).

Example of the control plane code which enables a timer to generate 64 timer jobs every 1 millisecond.

```
EZapiChannel_ConfigCmd_SetPMUTimerParams(side=0, timer_id= TCP_TICK, enable=true,
logicalID=TIMER, budgetID= TIMER, PMUQueue=0, numOfJobs=64, nanoSecPeriod=100000,
secPeriod=0)
```

Example of the data plane code which handle the timer job.

The timer job is recognized by *logical\_id*. If it is a timer job the code in the sample discards it and follows by calling the *handler\_timer* API with *timer\_id* and *event\_id*.

```
/* Read job from PMU */
ezdp_receive_job(&jid,&jd, 0);

/* Process job according to logicalID */
switch(jd.frame_desc.logical_id)
{
...
case TIMER:
    /* incoming job is a timer job */
    /* Free job ID */
    ezdp_disacrd_job (&jid, &jd);
    handle_timer(jd.rx_info,timer_info.timer_id,
                jd.rx_info,timer_info.event_id);
    break;
...
}

void handle_timer(uint32_t timer_id, uint32_t event_id)
{
    /* inspect timer ID */
    switch(timer_id)
    {
        ...
        case TCP_TICK:
            /* send tick to TCP with <event_id> as flow */
            tcp_send_tick(event_id);
            break;
        ...
    }
}
```

## 4.4 Advanced Frame and Frame Buffer Manipulation

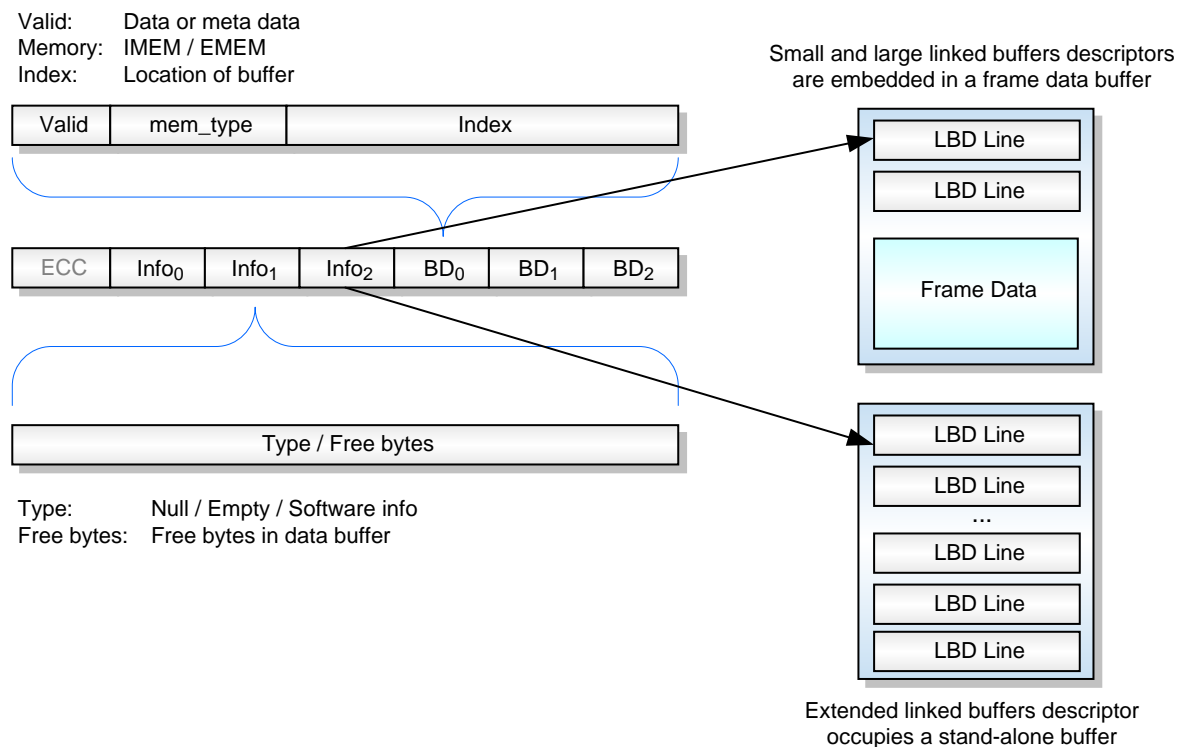
### 4.4.1 Linked Buffer Descriptors

As we have seen, each buffer descriptor functions as a pointer to a single 256 byte buffer. Most frames however, are composed of more than a single buffer. Therefore, an efficient representation is needed to describe a series of buffers. The Linked Buffers Descriptor, or LBD for short, is this representation.

The linked buffer descriptor is a memory structure (i.e. array) that holds a series of buffer descriptors and ancillary data.

The figure below illustrates the way the system structure of the linking of NPS buffer descriptors.

**Figure 14. NPS-400 buffer descriptors**



Here is an abbreviated definition of the linked buffer descriptor and related data structures:

```
#define EZDP_LINKED_BUFFER_DESC_LINE_NUMBER_OF_BUFFERS_DESC 3

struct ezdp_buffer_info {
    uint8_t free_bytes;
};

struct ezdp_linked_buffers_desc_line {
    uint8_t ecc;
    struct ezdp_buffer_info buf_info[EZDP_LINKED_BUFFER_DESC_LINE_NUMBER_OF_BUFFERS_DESC];
    struct ezdp_buffer_desc buf_desc[EZDP_LINKED_BUFFER_DESC_LINE_NUMBER_OF_BUFFERS_DESC];
} __packed;

enum ezdp_linked_buffers_desc_size {
    EZDP_SMALL_LBD = 1,
    EZDP_LARGE_LBD = 2,
    EZDP_EXTENDED_LBD = 16
};

struct ezdp_linked_buffers_desc {
    struct ezdp_linked_buffers_desc_line line[0];
};
```

```
};  
  
struct ezdp_small_linked_buffers_desc {  
    struct ezdp_linked_buffers_desc_line line[EZDP_SMALL_LBD];  
};  
  
struct ezdp_large_linked_buffers_desc {  
    struct ezdp_linked_buffers_desc_line line[EZDP_LARGE_LBD];  
};  
  
struct ezdp_ext_linked_buffers_desc {  
    struct ezdp_linked_buffers_desc_line line[EZDP_EXTENDED_LBD];  
};
```

Each linked buffer descriptor is comprised of a series of lines; each line holds 3 buffer descriptors and additional information about these buffers. The additional data provided is the amount of free bytes left in the buffer, for frame data buffers.

Linked buffer descriptors come in three sizes:

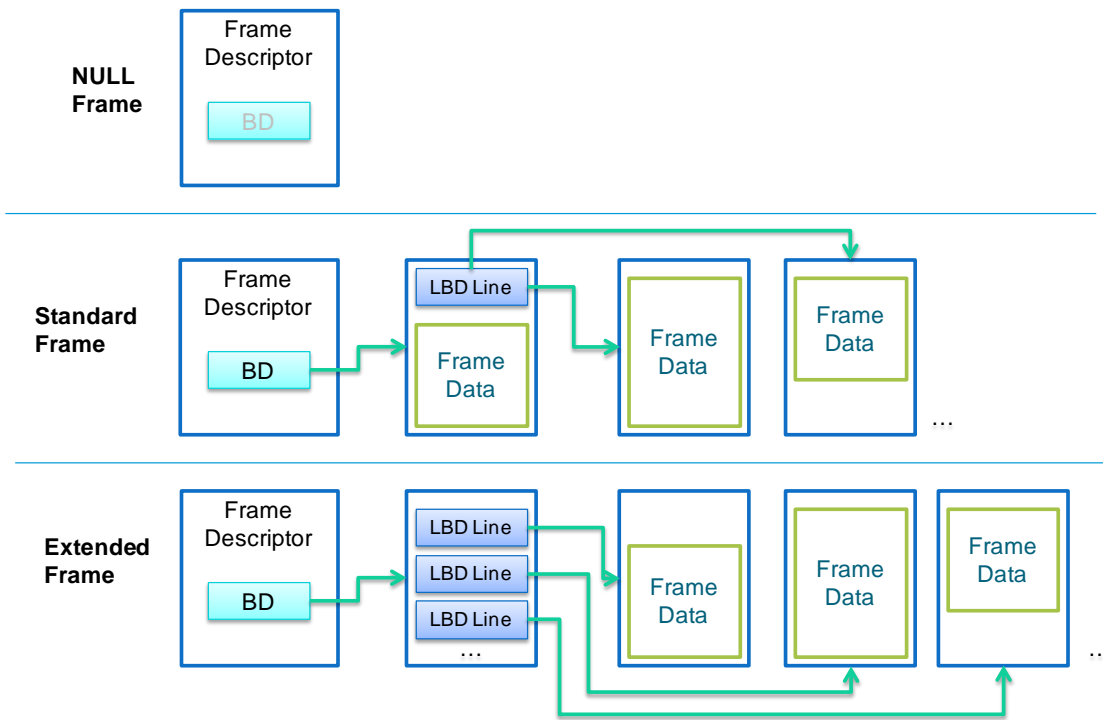
- **Small** – the small linked buffer descriptors is comprised of a single line, and therefore has room for 3 buffer descriptors. This linked buffer descriptor type resides in the first 16 bytes of a frame data buffer.
- **Large** – the large linked buffer descriptors is comprised of two lines, and therefore has room for 6 buffer descriptors. This linked buffer descriptor type resides in the first 32 bytes of a frame data buffer.
- **Extended** – the extended linked buffer descriptor is comprised of 16 lines, and therefore has room for 16 buffer descriptors. This type of linked buffer descriptor can only reside in the first buffer of an extended frame.

Linked buffer descriptors are special in that they are protected by an ECC scheme. While the protection handling is performed automatically by the hardware, care must still be taken to only copy linked buffer descriptors using by a specialized API provided for this purpose which will be described in the [Working with linked buffer descriptors](#) section.

## 4.4.2 Frame Types

The NPS supports three frame types which differ from one another in the way the frame data is arranged in system memory.

Figure 15. NPS-400 frame types



These frame types are:

- **The standard frame** – the standard frame type is used to represent standard Ethernet frames. The first buffer in this frame type is a partial linked buffer descriptor, of either short (1 line) or long (2 lines) type. The linked buffer descriptor lines in the first buffer hold buffer descriptors, or pointers, to the rest of the frame buffers (up to 6 additional buffers). The remainder of the first frame buffer contains the first bytes of the frame data.
- **The extended frame** – the extended frame type is used to represent frames that require more than 6 frame buffers, such as Ethernet Jumbo frames. The extended frame has a full linked buffer descriptor as its first buffer which points to additional buffers. These buffers in turn hold the actual frame data.
- **The NULL frame** – as the name implies, frames of the NULL frame type have no frame data buffers at all. This kind of frame is useful when allocating a new frame structure for creating a new frame, or when job without associated frame is passed to SW (for example timer job generated by HW).

### 4.4.3 The Frame Descriptor

As we have described in the previous section, each frame is assigned a job that is being tracked by the system. Part of the job descriptor structure is the frame descriptor structure which is a data structure in the system memory.

Here is the definition of the frame descriptor data structure:

```
enum ezdp_frame_type {
    EZDP_NULL_FRAME,
    EZDP_EXT_FRAME,
    EZDP_STD_FRAME
};

struct ezdp_frame_desc {
    ...
    uint8_t          ecc;
    enum ezdp_frame_type type;
    unsigned transmit_confirmation_flag;
    unsigned timestamp_flag;
    unsigned transmit_keep_buf_flag;
    unsigned gross_checksum_flag;
    ...
    unsigned class_of_service;
    unsigned buf_budget_id;
    uint16_t frame_length;
    uint8_t data_buf_count;
    uint8_t header_offset;
    struct ezdp_buffer_desc buf_desc;
    uint8_t free_bytes;
    uint8_t logical_id;
    enum ezdp_frame_multicast_control multicast_control;
    unsigned job_budget_id;
};
```

Let us go over the frame descriptor structure and its fields:

- **ECC** – the *ecc* field is a part of the memory protection scheme protecting the frame descriptor data from random corruption. The frame descriptor only uses the in-band ECC in this field when it is stored in EMEM; when the frame descriptor structure is stored in IMEM the ECC information is stored out of band by the memory subsystem hardware. In both cases, the ECC is always computed and checked by the hardware and the programmer never needs to check or modify ECC data.
- **Type** – the *type* field describes the frame type, as described by the [Frame types](#) section above.
- **Transmit Confirmation Flag** – on received frames, this flag indicates that the job is a confirmation response; on frames for transmission, this flag is used as a request to return a job confirmation when transmission is done. When the timestamp flag is on, the transmit confirmation flag is implied set and, therefore, its value is ignored.
- **Timestamp Flag** – on received frames, this flag indicates the presence of a timestamp value in the job descriptor structure's receive side information structure; on frames for transmission, this flag is used as a request for a 1588 timestamp confirmation response. When this flag is set, it implies that transmission confirmation is requested regardless of the value of the transmit confirmation flag, whose value is ignored.
- **Transmit Keep Buffers Flag** – on frames for transmission, this flag indicates a request to avoid freeing the frame buffers automatically by the network DMA hardware engine. It is, therefore, the software's responsibility to free the frame buffers when the frame has been sent. For this purpose, the flag is echoed on frame transmission confirmation responses.
- **Gross Checksum Flag** – on received frames, this flag indicates that the frame's IP checksum has been calculated; on frames for transmission, the flag is reserved and should never be set.

- **Class of service** – the *class\_of\_service* field indicates the class of service assigned to the frame – from the highest class represented as a value of 0 to the lowest class represented by a value of 3. The class of service is set on frame reception by the Input Classification Unit based on the network interface on which the frame was received and various fields in the frame headers. The specific algorithm used to compute the class of service and which fields are used are determined by the ICU configuration. The class of service determines the priority given to the frame queued for transmission via a network interface channel, as well as by the Traffic Manager when making QoS decisions regarding the frame.
- **Buffer Budget ID** – the *buf\_budget\_id* field holds the budget group ID of the buffer pool from which the frame buffers were allocated. The Buffer Management Unit which allocates buffers for frame data supports several different pools of buffers to ensure fairness in resource distribution. The budget ID identifies the pool from which the frame buffers were allocated. The budget group ID used to allocate the frame buffers from is configurable with the default value being an ID identical to the network interface port number that the frame was received from. Budgets are further discussed in the [Buffer pools](#) section ahead.
- **Frame Length** – the *frame\_length* field contains the total length in bytes of the frame data starting from the layer 2 frame headers.  
The 4 byte FCS field may be included if the MAC of the applicable network interface is configured to keep FCS bytes for received frames or to avoid adding FCS bytes on transmission. Alternatively, the 4 byte FCS may be excluded if the MAC is configured to strip FCS for received frames or to add FCS bytes for transmitted ones.  
The frame length excludes all other data not received or transmitted even if it present in the frame data buffers, such as the frame context bytes, and other bytes preceding the header offset field position.  
A value of 0x3FFF means that frame data length is greater or equal to 16383 bytes; therefore, the maximum length of a frame that can be accurately represented by an exact value is 16382 bytes (the field value being 0x3FFE).
- **Data Buffers Count** – the *data\_buf\_count* field indicates the number of data buffers that hold frame data. This count does not include buffers containing non frame data, such as extended linked buffers descriptor buffers or buffer that contain software context.
- **Buffer Descriptor** – the *buf\_desc* field contains a buffer descriptor for the first, and possibly only, buffer of the frame. Depending on the frame type as described in the *type* field, this buffer may contain frame data, additional buffer descriptors that contain frame data themselves, or both.
- **Header Offset** – the *header\_offset* field provides the offset from the beginning of the first frame data buffer where frame data starts. Note that small and large linked buffers descriptors and any software context information which is not part of the frame data and are not to be transmitted by the MACs are not included in the offset. The offset may be used to add additional frame headers, where required, without the need to copy the original of the frame.
- **Logical ID** – the *logical\_id* field is a software logical context to designate the type of processing the frame should undergo. Logical IDs were described in [The frame logical ID](#) section above.
- **Free Bytes** – the *free\_bytes* field indicates how many free bytes are left at the end of the frame header data buffer pointed by a buffer pointed to by the *buf\_desc* field. The field value is only meaningful for frames of the standard type.
- **Multicast Control** – the *multicast\_control* field describes the frame multicast handling, if applicable. A description of multicast handling and the possible values for this field are described further in the [Multicast and Replication](#) section ahead.
- **Job Budget ID** – the *job\_budget\_id* field holds the budget group ID of the buffers pool from which the job structure buffer was allocated. The Buffer Management Unit which allocates buffers for jobs supports several different pools of job buffers to ensure fairness in resource distribution.

The budget ID identifies the pool from which the job buffer was allocated. Budgets are further discussed in the [Buffer pools](#) section ahead.

#### 4.4.4 Manipulating the frame buffer structure

In the previous section we have seen how to access and modify the content of a frame using the frame data buffer iterators. The iterators provide a high level API which abstracts away many of the gory details of how a frame buffer is structured. When all we require is access to and the ability to change the frame data, this high level API is the right way to go.

Sometimes however, we need to manipulate a frame buffer structure itself, such as when we wish to add or remove buffers to an existing frame, copy frame data between buffers or copy references to frame buffers (in the form of linked buffers descriptors) between one buffer to another.

One use case which requires these abilities is the desire to move frame buffers that were originally allocated in IMEM by the network DMA units, into replacement buffers in EMEM for the purpose of freeing up valuable IMEM memory for frames which, for whatever reason, are expected to linger in the NPS for some time before being transmitted, possibly due to assignment to congested traffic manager queues.

For this reason, we shall present now the low level APIs and structures which allow fine grained control over a frame buffer structure which the frame buffer iterators are based on.



In addition to the use case mentioned above, frame buffer structure manipulation forms the basis for handling network multicast and frame replication for the purpose of port mirroring and similar services. The NPS is equipped with several features supporting these use cases, such as multicast frame buffer reference counting. These advanced features are discussed in the [Multicast and Replication](#) and [Controlling the Jobs Pipeline](#) sections ahead.

##### 4.4.4.1 Getting hold of the buffer descriptor of a standard frame

For frames of the standard type, this first buffer descriptor can be read from the *buf\_desc* field of the frame descriptor:

```
struct ezdp_buffer_desc bd;

if (job_desc.frame_desc.type == EZDP_STD_FRAME) {
    bd.raw_data = job_desc.frame_desc.buf_desc.raw_data;
} else {
    ...
}
```

For frame buffers beyond the first buffer and for extended frames, we will first need to load and parse a linked buffer descriptor in order to obtain the desired buffer descriptor. We shall explain how this is accomplished in the [Working with linked buffer descriptors](#) section ahead.

##### 4.4.4.2 Determining the frame buffer data length

Before we can proceed in copying the frame data from the buffer into our local memory we need to determine the size of the data we wish to copy.

Each frame buffer has a fixed size of 256 bytes; however the frame data may be located at an offset in relation to the buffer start and end before the buffer's end.

The API call *ezdp\_buf\_data\_len()* should be used to efficiently calculate the frame data length:

```
uint32_t ezdp_buf_data_len (uint32_t header_offset, uint32_t free_bytes, uint32_t max_length)
```



The `ezdp_buf_data_len()` calculate the length of the meaningful data in a frame buffer data.

This API call performs the following calculation:

$$\text{data\_length} = \text{max\_length} - \text{header\_offset} - \text{free\_bytes}$$

However, the calculation is done in an accelerated fashion using a specialized ISA instruction and therefore should be preferred over a manual calculation.

For the first buffer in a standard frame, this is a typical usage:

```
uint32_t len;

len = ezdp_buf_data_len(job_desc.frame_desc.header_offset, \
    job_desc.frame_desc.free_bytes, EZDP_BUFFER_DATA_SIZE);
```

#### 4.4.4.3 Working with linked buffer descriptors

The instruction that we have presented thus far allows us to load, modify and store back to system memory the first buffer of a standard type frame. In order to work in the same fashion with buffers other than the first one or with buffers that belong to an extended type frame, we need to work with linked buffer descriptors.

As previously described in the [Linked buffer descriptors](#) section above, a linked buffers descriptor is an array of buffer descriptors which form a linked list of buffers, accompanied by metadata about the buffers in the list, such as the types and data length, if applicable.

Linked buffer descriptors are used as a way to create a linked list of all the buffers that belong to a frame: the small and large linked buffer descriptors are embedded in the first frame data buffer of standard type frames and points to the rest of the frame buffers beyond the first, whereas the extended linked buffer descriptors reside in standalone metadata buffers of extended frames.

Linked buffer descriptors protection is handled in a special way: the ECC codes protecting the data content are part of the linked buffer descriptor data structure itself. While the protection is handled automatically by the hardware, care must be taken to use the following specialized API calls when copying linked buffer descriptors so that this special protection scheme can be preserved:

**`void ezdp_load_frame_lbd( void __cmem * dst_ptr, struct ezdp_buffer_desc __cmem * src_bd_ptr, uint32_t src_bd_offset, uint32_t size, uint32_t flags)`**

The `ezdp_load_lbd()` call invokes a DMA engine to copy the content of a buffer containing a linked buffer descriptor in IMEM or EMEM into a buffer in the CMEM, while taking care to correctly handle the embedded ECC information.

The `dst_ptr` argument provides the pointer to the CMEM buffer to which the data will be copied, whereas the `src_bd_ptr` argument provides a pointer to a buffer descriptor structure in the CMEM that points to the source data buffer.

The `src_bd_offset` and `size` arguments provide the offset from the beginning of the data buffer from which to copy the linked buffers descriptor and the length of linked buffers descriptor to copy.

The `flags` parameter is currently unused and should be passed as zero.

---

**Warning!** Do not use this function to load frame data. Doing so will corrupt the data loaded and can possibly result in an ECC protection fault! Use the functions described in the previous section, [Accessing and modifying frame data](#), for plain non-structured data.

---

Now that we know how to load a linked buffers descriptor, let us examine the different use cases of accessing frame buffers pointed to by linked buffers descriptors.

#### 4.4.4.4 Loading the first data buffer of an extended frame

As we described in the [Frame types](#) section above, the first buffer of an extended frame, which is pointed to by the frame descriptor, does not contain a frame data; instead, the first buffer of an extended frame contains an extended linked buffer descriptors that points to the extended frame data buffers.

Therefore, to load the first data buffer of an extended frame which typically contains the frame headers, we need to perform two steps: load the first line of the extended linked buffers descriptor in the first frame buffer, extract from that line the buffer descriptor for the first data buffer and proceed to load that buffer.

Here is a code example for this common use case:

```
static struct ezdp_link_buffer_desc_line lbd_line __cmem_var;

if ( likely(job_desc.frame_desc.type == EZDP_STD_FRAME)) {
    ...
} else {
    ezdp_load_frame_lbd((void*)&lbd_line, &job_desc.frame_desc.buf_desc, \
        0, sizeof(struct ezdp_link_buffer_desc_line), 0);

    bd.raw_data = lbd_line.buf_desc[0].raw_data;
}
```

Once we have the buffer descriptor of the first buffer that contains frame data, we can proceed to load the headers in the same way we did for standard frames in the [Loading the frame data into local core memory](#) section above.

#### 4.4.4.5 Calculating the size of the linked buffer descriptor

In the examples provided above the linked buffer descriptor was accessed one line at a time. Frame structure manipulation however can be simpler if we load all the lines of a linked buffer descriptor that contain references to frame buffers at once.

For this we need to calculate the size of the linked buffer descriptor that contains the relevant information. This is done by calling the `ezdp_lbd_len()` API call.

**`uint32_t ezdp_lbd_len ( uint32_t data_buf_count )`**

The `ezdp_lbd_len()` API call calculates the size of the relevant data in a linked buffer descriptor based on the number of buffers it references.

The function returns the size in bytes of data required to load the entire portion of the linked buffer descriptor that references the first `data_buf_count` buffers.

In a typical use case, the `data_buf_count` parameter will be derived from the frame descriptor in the following manner:

- For standard type frames, the value of the `data_buf_count` field of the frame descriptor decremented by one should be passed, since the linked buffer descriptor embedded in the first frame data buffer does not reference itself.
- For extended type frames, the value of the `data_buf_count` field of the frame descriptor can be passed as is, since the first buffer which contains the linked buffer descriptor is not a data buffer but a metadata buffer and so is not included in the count.

Here is a typical usage example for an extended type frame:

```
uint32_t len;

len = ezdp_lbd_len(job_desc.frame_desc.data_buf_count);
```

Loading the linked buffer descriptor can now be done using the `ezdp_load_frame_lbd()` API as before with the calculated length supplied as the `size` parameter of this function call.

#### 4.4.4.5.1 Iterating over all of a frame's data buffers

Iterating over all the data buffers of a frame requires us to obtain a buffer descriptor for each buffer and load it to the local memory. The frame data location within these buffers abides by the following rules:

In the first buffer that contains frame data, the data does not start at the buffer beginning, but at an offset reported by the frame descriptor *header\_offset* field. All other frame data buffers start at the beginning of the buffer.

In addition, each frame buffer may have free unused space at the end of the buffer. The amount of this free space is reported in the frame descriptor *free\_bytes* field for the first buffer of standard type frames and in the *free\_bytes* field in the *buf\_info* structure of each line of the linked buffers descriptor.

Here is a code example that demonstrates how to iterate across all the data buffers of frame, loading each of them to a buffer in the local core memory, ready for inspection or modification:

```
uint32_t bd_idx, block_idx, len, line;

uint32_t offset = job_desc.frame_desc.header_offset;
uint32_t buf_count = job_desc.frame_desc.data_buf_count;

if(job_desc.frame_desc.type == EZDP_STD_FRAME) {

    len = ezdp_buf_data_len (offset, \
                             job_desc.frame_desc.free_bytes, EZDP_BUFFER_DATA_SIZE);

    ezdp_load_frame_data((void*)&data[offset], \
                        &job_desc.frame_desc.buf_desc, offset, len, 0);

    offset = 0;
    --buf_count;

    printf("The standard frame first buffer is loaded into data[].\n");
    printf("Data starts at offset %d and is %d bytes.\n", offset, len);
}

for(bd_idx = 0; bd_idx < buf_count; ++bd_idx) {

    block_idx = ezdp_mod(bd_idx, 3, 0);

    if(block_idx == 0) {

        line = ezdp_div(bd_idx, 3, 0, 0);

        ezdp_load_frame_lbd(&lbd_line, &job_desc.frame_desc.buf_desc), \
            (line << 4), sizeof(struct ezdp_link_buffer_desc_line), 0);
    }

    len = ezdp_buf_data_len(offset, \
                            lbd_line.buf_info[block_idx].free_bytes, EZDP_BUFFER_DATA_SIZE);

    ezdp_load_frame_data((void*)&data[offset], \
                        &lbd_line.buf_desc[block_idx], offset, len, 0);

    offset = 0;

    printf("The frame's lbd %d buffer is loaded into data[]\n");
    printf("Data starts at offset %d and is %d bytes.\n", offset, len);
}
```

#### 4.4.4.5.2 Storing a linked buffer descriptor back to global memory

Once we have made modifications on a copy of a linked buffer descriptor loaded into our local memory, we need to synchronize the modified local copy with the globally visible copy of the descriptor.

This is done by calling the `ezdp_store_frame_lbd()` API call:

```
void ezdp_store_frame_lbd(struct ezdp_buffer_desc __cmem * dst_bd_ptr, uint32_t dst_bd_offset, void __cmem * src_ptr, uint32_t size, uint32_t flags)
```

The `ezdp_store_lbd()` call invokes the DMA engine which copies the content of a buffer containing a CMEM linked buffer descriptor into a data buffer in IMEM or EMEM. The ECC would be handled automatically by the DMA engine.

The `dst_bd_ptr` argument provides a pointer to a buffer descriptor structure in the CMEM that points to the destination data buffer into which the content of the buffer pointed to by the `src_ptr` argument will be copied.

The `dst_bd_offset` argument provide the offset from the beginning of the data buffer into which the linked buffer descriptor would be copied to, while the `size argument` provides the size of the linked buffer descriptor to copy.

The `flags` parameter is currently unused and should be passed as zero.

#### 4.4.4.6 Copying data between frame buffers

In the process of making frame structure manipulations, it is often useful to copy the content of one frame data buffer, in whole or in part, into another frame buffer directly, without loading the data first to a local buffer. This can be achieved by calling the `ezdp_copy_frame_data()` API call:

```
void ezdp_copy_frame_data(struct ezdp_buffer_desc __cmem * dst_bd_ptr, uint32_t dst_bd_offset, struct ezdp_buffer_desc __cmem * src_bd_ptr, uint32_t src_bd_offset, uint32_t size, uint32_t flags)
```

The `ezdp_copy_data()` call invokes a DMA engine to copy one data buffer in IMEM or EMEM into another which also resides in EMEM or IMEM,.

The `dst_bd_ptr` and `src_bd_ptr` arguments provide pointers to buffer descriptor structures in the CMEM that point to the destination and source data buffers, respectively, from and into which the data will be copied.

The `dst_bd_offset`, `src_bd_offset` and `size` arguments provide the offset from the beginning of the data buffers from and into which to copy the data and the size of data to copy.

The `flags` argument allows controlling frame data ECC checksum calculation, as explained in the [Synching frame data to global memory](#) section above. In case of doubt, pass zero as the `flags` argument.

#### 4.4.4.7 Copying linked buffer descriptors between frame buffers

Just as we can copy frame data between buffers directly, we can also copy linked buffer descriptors, or part thereof between buffers. However, due to the different ECC protection scheme used to protect linked buffer descriptors, a different API call named `ezdp_copy_frame_lbd()` must be used for this purpose:

```
void ezdp_copy_frame_lbd(struct ezdp_buffer_desc __cmem * dst_bd_ptr, uint32_t dst_bd_offset, struct ezdp_buffer_desc __cmem * src_bd_ptr, uint32_t src_bd_offset, uint32_t size, uint32_t flags)
```

The `ezdp_copy_lbd()` call invokes a DMA engine to copy one buffer in IMEM or EMEM containing a linked buffer descriptor into another such buffer.

The `dst_bd_ptr` and `src_bd_ptr` arguments provide pointers to buffer descriptor structures in the CMEM that point to the destination and source data buffers, respectively, from and into which the linked buffers descriptor will be copied.

The *dst\_bd\_offset*, *src\_bd\_offset* and *size* arguments provide respectively the offset from the beginning of the data buffers from and into which to copy the linked buffer descriptor and the length of the linked buffer descriptor to copy.

The *flags* parameter is currently unused and should be passed as zero.

#### **4.4.4.8 Cloning linked buffer descriptors between frame buffers**

In the very common case that we wish copy a linked buffers descriptor or a portion of it from the same offset within one buffer to another, a more optimized version of the above API call, named *ezdp\_clone\_frame\_lbd()*, should be used. While the difference may seem minute to the untrained eye, this alternate API call requires less cycles to perform the operation and should, thus, be preferred if applicable:

```
void ezdp_clone_frame_lbd(struct ezdp_buffer_desc __cmem * dst_bd_ptr, struct ezdp_buffer_desc  
__cmem * src_bd_ptr, uint32_t bd_offset, uint32_t size, uint32_t flags)
```

The *ezdp\_clone\_frame\_lbd()* is an optimized version of *ezdp\_copy\_frame\_lbd()* call for the common case where the offset of the linked buffer descriptor to be copied is identical in the source and destination buffers. This call should be preferred over *ezdp\_copy\_frame\_lbd()* where its use is applicable.

The *dst\_bd\_ptr* and *src\_bd\_ptr* arguments provide pointers to buffer descriptor structures in the CMEM that point to the destination and source data buffers, respectively, from and into which the data will be copied.

The *bd\_offset* and *size* arguments provide the common offset from the beginning of the data buffers from and into which to copy the data and the length of the linked buffer descriptor to copy.

The *flags* parameter is currently unused and should be passed as zero.

## 4.5 Advanced Search Features

### 4.5.1 Scanning and Aging Mechanism

The PMU timer mechanism can be used to implement any application process that requires a fixed rate of events, such as traffic generation. One use case is using PMU timers to age a search table's entries.

To use a PMU timer to age a search table's entries, code your application to process a PMU timer event job by a lookup in the table entry with the timer job *event\_id* and delete or alter it, as needed.

#### 4.5.1.1 Aging direct tables

For direct tables, no specific API is required, as loading a direct table entry is the same as performing a lookup in the table with the entry number as the key.

Deleting or modifying the entry as part of an aging process is performed with the same API calls used to delete or modify a direct search table entry.

#### 4.5.1.2 Aging hash structures

Unlike direct tables, hash structures may have multiple entries at every hash table slot. Therefore, for every hash table slot we need to scan all the entries in that slot.

This is done using the *ezdp\_scan\_hash\_slot()* API call:

```
void ezdp_scan_hash_slot( struct ezdp_hash_struct_desc *hash_desc, uint32_t slot_num,
ezdp_scan_hash_slot_callback_t callback, void* callback_user_data, void __cmem *entry_ptr,
ezdp_scan_hash_slot_working_area_t __cmem *wa_ptr)
```

The *ezdp\_scan\_hash\_slot()* call iterates over the entries in the *slot\_num* of hash referred to by *hash\_desc*.

The function requires two CMEM buffers to operate: one with enough room to store a hash entry whose address is provided in the *entry\_ptr* parameter and the other of a fixed-size temporary work area pointed to by the *wa\_ptr* parameter. Both buffers may be safely reused after the *ezdp\_scan\_hash\_slot()* function returns.

Typically, when triggered by a PMU timer event, the timer event ID can be used as *slot\_num* (*job\_desc.rx\_info.timer\_info.event\_id*) for this purpose the PMU timer number of events should be configured to  $2^{\text{Hash Size}}$ . For example, if the hash size is 10, number of events should be 1024.

For each entry in the slot, the function calls the *callback* function pointed to by *callback* with a parameter of *callback\_user\_data*. The callback function has the following definition:

```
enum ezdp_scan_hash_slot_action {
    EZDP_ACCEPT_ENTRY,
    EZDP_DELETE_ENTRY,
    EZDP_UPDATE_ENTRY
};
typedef enum ezdp_scan_hash_slot_action (*ezdp_scan_hash_slot_callback_t)
( void __cmem *result_ptr, struct ezdp_hash_struct_desc *hash_desc,
  void * user_data );
```

Each time the callback function is called, the *result\_ptr* parameter points to the buffer in CMEM which holds the result data of this hash entry.

The *hash\_desc* parameter is a pointer to the descriptor of the scanned hash structure.

The *user\_data* parameter is identical to the *callback\_user\_data* parameter provided to the *ezdp\_scan\_hash\_slot()* call.

After examining the hash entry and optionally updating the hash entry result, the callback function should return one of the following results:

- **EZDP\_ACCEPT\_ENTRY** – the hash entry should be left as is.
- **EZDP\_DELETE\_ENTRY** – the hash entry should be deleted.
- **EZDP\_UPDATE\_ENTRY** – the hash entry should be updated with the modified result.

Refreshing of the hash entry upon lookup is the responsibility of the application. For this purpose, the application must manage a refresh bit or aging counter in the result, depending on the granularity and variance required by the aging time. In addition, the application may decide to include a “static” indication bit in the result indicating static entries that should not be aged out. With this in mind, the scan callback could look something like:

```
static enum ezdp_scan_hash_slot_action aging_callback( void __cmem *result_ptr,
struct ezdp_hash_struct_desc *hash_desc, uint32_t __unused user_data )
{
    struct my_result* result;
    result = (struct my_result* )result_ptr;

    if (result->static)
        return EZDP_ACCEPT_ENTRY;

    if (result->aging-- > 0)
        return EZDP_UPDATE_ENTRY;
    else
        return EZDP_DELETE_ENTRY;
}
```

In order to support different aging times on a per-entry basis, the initial aging counter value can be stored in the entry result as well. When a lookup is performed that should cause the entry to be refreshed, the application can check if the aging counter is smaller than the initial aging counter value, and if so, refresh the entry by updating it with the initial aging counter value as the aging counter.

Note that a lookup does not require refreshing in all cases. For example, in a hash structure representing a MAC address table, typically when a lookup is performed with the source MAC address as the key, a refresh is required, but when performed with the destination MAC address, a refresh is not required.

## 4.5.2 TCAM

A common requirement from network equipment is use of search structures that support random wildcards. A typical use of this feature is implementation of an Access Control List (ACL) that could be made of various frame fields (e.g. source and destination MAC addresses, IP addresses, TCP ports). Each frame being processed must verify compliance in the database, however the rules may be comprised of only a subset of the fields, while other fields may be masked (wildcarded). For example, one rule may require all traffic originating at IP addresses “231.55.\*.\*” to be discarded, regardless of other field values. Another rule may indicate that all traffic destined to IP address “17.55.12.3” with TCP source port “50” to receive high priority. Each frame being processed must parse and build a key that contains all potential fields in order to check if one of the rules matches. Each rule is added to the TCAM database with a unique numerical rank value, so that in case more than one rule matches the lookup, the highest rank (lowest numerical value) is returned. The rank can then be used to fetch additional information about the rule from a separate table, or from the TCAM’s own associated data, in case its implementation supports it.



### 4.5.2.1 Internal TCAM

NPS-400 contains two internal TCAMs, each 1.25Mb, which can be used with various combinations of keys sizes of 10, 20, 30, 40, 50, 60, 70 and 80 bytes.

Lookup to the internal TCAM is performed with the `ezdp_lookup_int_tcam()` API. The parameters of this API include the side (one out of the two internal TCAMs), the lookup profile, the key and its size, an optional key prefix mask and a pointer to where the result will be written. The 16B result is written to CMEM, and first 4B are also returned by the command.

```
ezdp_lookup_int_tcam_retval_t ezdp_lookup_int_tcam(uint32_t side, uint32_t lookup_profile, void*
__cmem key_ptr, uint32_t key_size, uint32_t key_prefix_mask, struct ezdp_lookup_int_tcam_result*
__cmem result_ptr)
```

A single internal TCAM lookup operation can trigger up to four lookups in separate tables. Each may require a different key size, and the key used for each of those lookups can be constructed from a different subset of the main key. The lookup profile is configured by the control plane with the lookup tables. Each table is configured with the area in the TCAM it covers, the key size, the key mask used to construct the key and whether it contains associated data. Out of the four tables in a profile, only the first could be configured to return associated data. For tables that do not return associated data, the match index is returned. The data-plane application may then perform another lookup in a direct table using the match index as a key in order to obtain an application specific logical result.

The data-plane code must be aware of the internal TCAM lookup profile so that it knows how many results to expect and uses the correct result format for parsing the results. A standard TCAM lookup may contain up to four results in the format of `struct ezdp_lookup_int_tcam_standard_result`; each contains a match indication and the return index.

Standard result without associated data:

127	126-111	110-96	95	94 - 79	78-64	63	62 – 47	46-32	31	30-15	14-0
M-0 (1 bit)	Reserved (16 bits)	Index-0 (15 bits)	M-1 (1 bit)	Reserved (16 bits)	Index-1 (15 bits)	M-2 (1 bit)	Reserved (16 bits)	Index-2 (15 bits)	M-3 (1 bit)	Reserved (16 bits)	Index-3 (15 bits)

In case the first result is configured to return with associated data, the structure for the appropriated data size should be used (e.g. `ezdp_lookup_int_tcam_4B_data_result`, `ezdp_lookup_int_tcam_8B_data_result`, `ezdp_lookup_int_tcam_12B_data_result`, `ezdp_lookup_int_tcam_16B_data_result`). And the rest of the optional results could be parsed in the standard format (`ezdp_lookup_int_tcam_standard_result`) with up to a total of 16 bytes.

With 4B associated data (`ezdp_lookup_int_tcam_4B_data_result`):

127	126-96	95	94 - 79	78-64	63	62 – 47	46-32	31	30-15	14-0
M-0 (1 bit)	Data [30:0]	M-1 (1 bit)	Reserved (16 bits)	Index-1 (15 bits)	M-2 (1 bit)	Reserved (16 bits)	Index-2 (15 bits)	M-3 (1 bit)	Reserved (16 bits)	Index-3 (15 bits)

With 8B associated data (`ezdp_lookup_int_tcam_8B_data_result`):

127	126-64				63	62 - 47	46-32	31	30-15	14-0
M-0 (1 bit)	Data [62:0]				M-2 (1 bit)	Reserved (16 bits)	Index-2 (15 bits)	M-3 (1 bit)	Reserved (16 bits)	Index-3 (15 bits)



With 12B associated data (*ezdp\_lookup\_int\_tcam\_12B\_data\_result*):

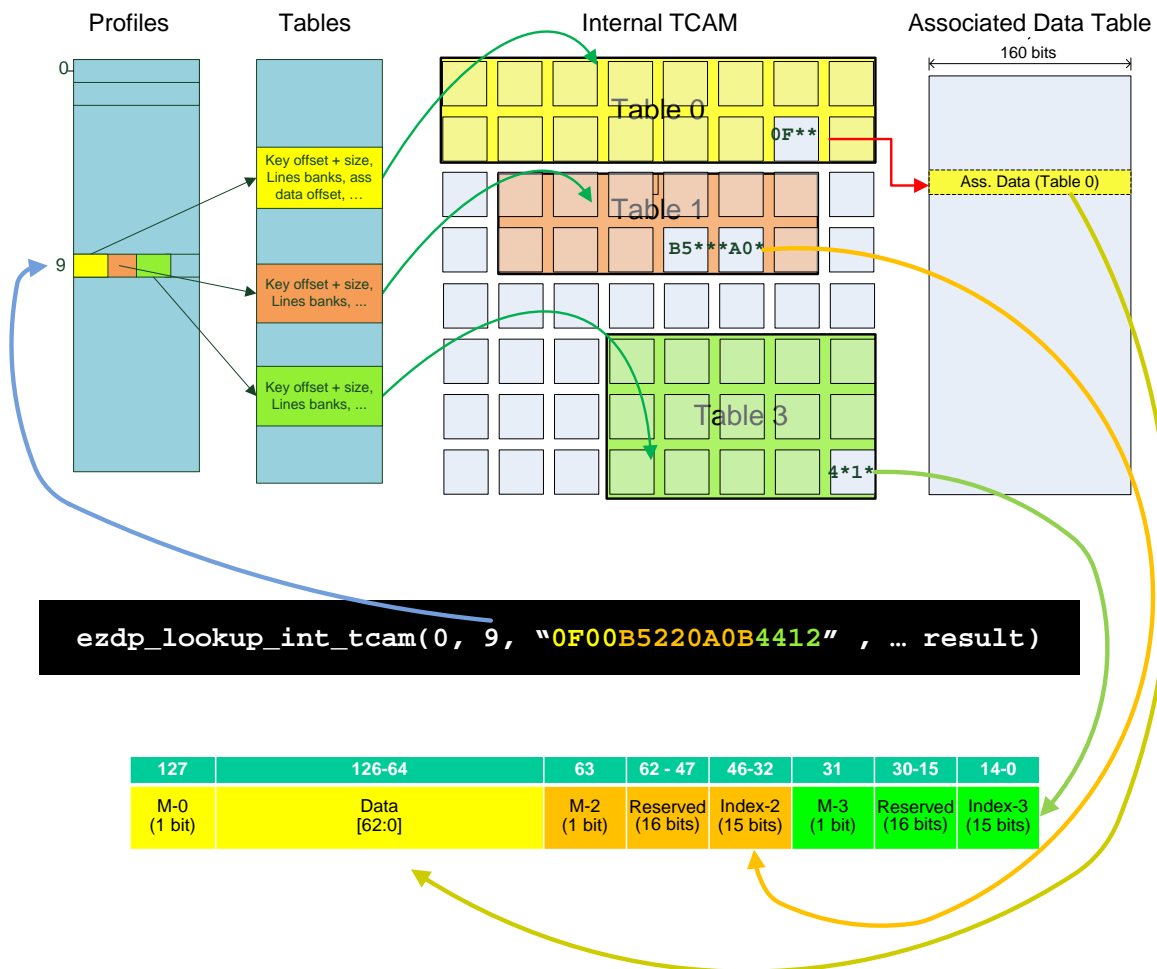
127	126-32	31	30-15	14-0
M-0 (1 bit)	Data [94:0]	M-3 (1 bit)	Reserved (16 bits)	Index-3 (15 bits)

With 16B associated data (*ezdp\_lookup\_int\_tcam\_16B\_data\_result*):

127	126-0
M-0 (1 bit)	Data [126:0]

The example below demonstrates invocation of an internal TCAM lookup to a profile that is configured to search in three separate tables, the first of which maps associated data result and the other two return an index. The key for each table is constructed from a separate part of the main key. The associated data size is 8B, allowing room for the additional two results of the other tables.

**Figure 16. Internal TCAM lookup example**



### 4.5.2.2 External TCAM

Up to two external TCAMs can be connected to the NPS-400 device, one on each side, via a standard Interlaken-LA interface. A lookup to the external TCAM is performed with the *ezdp\_lookup\_ext\_tcam()* API.

```
ezdp_lookup_ext_tcam_retval_t ezdp_lookup_ext_tcam( uint32_t side, uint32_t lookup_profile, void*
__cmem key_ptr, uint32_t key_size, char* __cmem result_ptr, uint32_t result_len)
```

In a similar manner to the internal TCAM, an external TCAM is also capable of performing lookups in up to four tables in parallel, each with a key that is constructed from a subset of the main key given to the *ezdp\_lookup\_ext\_tcam* command.

Data plane code can be aware of the lookup profile configuration and number of result elements to expect. Each result element may contain an Index, Index and Associated Data, or Associated Data only. When Associated Data exists, it can be 4, 8, 16 or 32 bytes long. The result element can be described by one of the following structures:

```
ezdp_lookup_ext_tcam_index_result_element
ezdp_lookup_ext_tcam_index_4B_data_result_element
ezdp_lookup_ext_tcam_index_8B_data_result_element
ezdp_lookup_ext_tcam_index_16B_data_result_element
ezdp_lookup_ext_tcam_index_32B_data_result_element
ezdp_lookup_ext_tcam_4B_data_result_element
ezdp_lookup_ext_tcam_8B_data_result_element
ezdp_lookup_ext_tcam_16B_data_result_element
ezdp_lookup_ext_tcam_32B_data_result_element
```

Each of these structures contains a 3-bit element type field (enum *ezdp\_ext\_tcam\_result\_element\_type*) at the same offset for all structures, which uniquely identifies the element type, and from which the application can know the element size.

The return value *ezdp\_lookup\_ext\_tcam\_retval\_t* would indicate if there was a match, multi-match, as well as if a specific error occurred, such as a time-out or device error. Error indication is also available in the first element of the result.

**Index shifting:** In many implementations of a TCAM, the returned value indicates the key location index of the 10-byte key. Therefore, for 20-byte keys, the returned value would always be a multiple of two, for 40-byte values, always a multiple of four, etc. If a direct table lookup is performed with the index as a key, it is possible to preserve space by shifting the index to the right according to the key size (1 for 20 bytes, 2 for 40 bytes, 3 for 80 bytes, etc.). The control-plane configuration of the external TCAM lookup profile allows having the HW return these indices shifted in order to preserve computing resources (*auiResultShift[4]* in *EZapiTCAM\_ExtTCAMLookupProfile*).

### 4.5.2.3 Algorithmic TCAM

NPS-400 provides a SW extension of the internal TCAM to IMEM and EMEM, which can be useful when an external TCAM is not available and the internal TCAM size is insufficient.

The lookup sequence is implemented by a combination of SW and HW features and involves an internal TCAM lookup, with associated data, and optionally additional IMEM and EMEM memory accesses. Algorithmic TCAM can also support up to eight range fields in configurable offsets of the key, each of 16 bits. In order to use an algorithmic TCAM, the structure descriptor must be initialized using the *ezdp\_init\_alg\_tcam\_struct\_desc()* API and validated with the *ezdp\_validate\_alg\_tcam\_struct\_desc()* API in a similar manner to hash table initialization.

The *ezdp\_lookup\_alg\_tcam()* API is used to implement the algorithmic TCAM lookup and returns the result index in the field indexed by *priority\_ptr*.

```
uint32_t ezdp_lookup_alg_tcam(ezdp_alg_tcam_struct_desc_t *struct_desc, uint8_t __cmem *key_ptr,
uint32_t key_size, uint32_t *priority_ptr, char __cmem *work_area_ptr, uint32_t work_area_size )
```

Algorithmic TCAM search structure management is performed entirely from the control plane. Configuration includes allocating memory, limiting number of memory accesses in IMEM and in EMEM, specifying the number of range fields and their offsets.

When populating entries, the control plane specifies the key, mask, ranges and rank. Each entry must have a unique rank number.

### 4.5.3 UltraIP


UltraIP structures can be used to implement lookups to a Longest-Prefix-Match (LPB) database, often used by IP routing tables in which each table entry rule is specified by the IP and its sub-network.

The data-plane application should initialize the descriptor to the UltraIP in a similar manner to [hash table initialization](#), e.g. initialize using `ezdp_init_ultra_ip_struct_desc()` API, and validate with `ezdp_validate_ultra_ip_struct_desc()` API.

Lookup is performed using the `ezdp_lookup_ultra_ip_entry()` API using the same arguments as with [lookup to a hash structure](#). In case of a match, the returned result size is 4 bytes which consist of one control byte and three bytes of user-defined data. In case a larger result is required, the returned result may represent an index to a direct table that can hold the actual result.

Adding and removing entries from the UltraIP structure can only be performed by the control-plane application. Each entry should have a key and mask specified where the mask must be in the form of LPM (e.g. 0xFFFFF00).

The performance which can be achieved with UltraIP structures depends on the amount of memory allocated for the structure, and on the number of entries it populates.

 For a detailed description of UltraIP, refer to the *NPS-400 Architecture Specifications* and the *EZcp Reference Manual*.



A TCAM can also be considered as an alternative for implementing an LPM lookup. In architectures where an external TCAM is available and not all four parallel lookups are utilized, it may be possible to add the LPM lookup as a parallel search and thus minimizing its added penalty.

### 4.5.4 Low level API

In general, it is always recommended to start programming the data plane using high-level APIs, and resort to using low-level APIs only where the high-level does not provide sufficient functionality or where performance is critical. Low level APIs are intended for experts who are very familiar with the way search structures operate.

For example, the `ezdp_lookup_hash_entry` high level API command implementation takes the following actions:

- Extracts the hash parameters from the structure descriptor.
- Calculates the hash function on the provided key using a low-level API such as `ezdp_prm_hash_key32`.
- Performs a lookup in the hash structure using the low-level `ezdp_prm_lookup_hash_entry` API.
- In case of error, repeats the lookup, until successful or the number of retries expire. In that case, returning a memory error indication.

To list some cases where the lookup operation may be somewhat optimized by using low-level API, reducing the number of CTOP instructions:


- Memory error checks are not required.
- The hash structure parameters are constant and known during compile time.
- The hashed key is already available.

In a similar manner, the high level API command *ezdp\_add\_hash\_entry* performs the following actions:

- Grabs a lock on the hash entry using the *ezdp\_prm\_lock\_hash\_slot* API.
- Performs a lookup to make sure that the entry does not exist. If it does, returns with an error.
- Uses the *ezdp\_prm\_add\_hash\_entry* low-level API to add the entry to the hash structure.
- Releases the lock using *ezdp\_prm\_unlock\_hash\_slot* low-level API.

Using of *ezdp\_prm\_add\_hash\_entry* on entries in the same hash slot from two threads simultaneously, or on a key that already exists in the hash table may corrupt the structure, so the high level API must take actions in order to protect from such cases.

In cases where the user's application has some other means to ensure that the entry does not already exist and that no two threads are trying to modify the same hash slot, there may be room for optimization by calling the *ezdp\_prm\_add\_hash\_entry* directly.

 All advanced low-level (primitive) operations are listed in the *EZdp Reference Manual*.

## 4.6 Precision Time Protocol IEEE 1588

Precision Time Protocol (PTP, IEEE 1588) is used in order to synchronize clocks among network devices; specifically each application of the protocol is between two devices, referred to as a master and a slave device. Implementation of the protocol requires special handling of the PTP packets not only at the master and slave devices, but also at all intermediate (transparent) devices carrying the PTP packets between the two. Intermediate devices need to update a correction field in order to compensate for the variance in propagation time induced by each device.

With a 1-step implementation of the PTP protocol, the transmitting devices are capable of storing the transmit timestamp in case of a master device, or processing time in case of an intermediate device, in the actual frame being sampled (Sync, Delay\_Req). Whereas with a 2-step implementation the transmitting devices must record the outgoing timestamp and send it in a Follow\_Up packet in case of a master device, or store in a database and retrieve it once the Follow\_Up or Delay\_Resp packets arrive, in the case of intermediate devices.

While 1-step requires use of dedicated hardware, 2-step requires bookkeeping by the data-plane application which involves processing code and memory storage. The IEEE 1588 standard also allows combining between the two in a given network, where the 1-step capable devices would update the correction field in the Sync and Delay\_Req packets, while the 1-step non-capable devices implement the required bookkeeping in order to update the correction field in the Follow\_Up and Delay\_Resp packets once they arrive.

PTP handling requires three basic capabilities, which are achieved in NPS-400 by a combination of hardware and software:

1. Sampling the receive timestamp of an incoming frame, and having the timestamp value available to the job processing the frame.
2. Sampling the transmit timestamp of an outgoing frame, and making the timestamp value available to a new job (for 2-step).
3. Sampling the transmit timestamp of an outgoing frame, and writing a calculated value based on it in the frame being transmitted. Optionally also fixing UDP checksum (for 1-step).

The following table describes which of the above functionalities is used in each processing path for different deployments:

	PTP Frame	Master	Slave	Intermediate (transparent)
2-step	Sync	2	1	1,2
	Follow_Up			
	Delay_Req	1	2	1,2
	Delay_Resp			
1-step	Sync	3	1	1,3
	Delay_Req	1	2	1,3
	Delay_Resp			

## 4.6.1 Sampling the Receive Timestamp of an Incoming Frame

The data plane processing code should classify the incoming frame. If it is found to be a PTP frame that requires sampling of the receive timestamp, the value should be available in the following fields:

```
struct ezdp_job_rx_interface_info
{
...
    uint8_t                timestamp_sec;
    /**< The timestamp seconds counter is sampled at one with the nanoseconds
        counter, providing together the standard 5B timestamp.
        Applicable only when frame descriptor time_stamp flag is on.
        The timestamp is sampled by the Rx MAC when the frame is speculated
        to be within the PTP length range (based on frame length) to minimize
        overheads. */

    uint32_t               timestamp_nsec;
    /**< The timestamp nanoseconds counter.
        Together with timestamp_sec provide the standard 5B timestamp.
        Applicable only when frame descriptor time_stamp flag is on.
        The timestamp is sampled by the Rx MAC when the frame is speculated
        to be within the PTP length range (based on frame length) to minimize
        overheads. */
...
}
```

The data-plane code should check validity of the timestamp field in the frame descriptor:

```
struct ezdp_frame_desc
{
...
    unsigned               timestamp_flag    : EZDP_FRAME_DESC_TIMESTAMP_FLAG_SIZE;
    /**< On receive path, this flag indicates presence of timestamp value in
        ezdp_job_desc.rx_info.
        For 2-step 1588 confirmation frame this flag also indicates success of
        adding 1588 timestamp. Or in other words, timestamp_flag will be off
        if adding captured 1588 timestamp failed.
        On transmit path, this flag is used as a request to add captured 1588
        timestamp to header by MAC.
        For 2-step 1588 packet, where confirmation response is required from the
        MAC, the transmit_confirmation flag should also be on. */
...
}
```

The timestamp will be available on frames that comply with the per-interface size configuration for timestamping (between *uiTimeStampMinFrameLength* and *uiTimeStampMaxFrameLength* in *EZapiIF\_EthIFParams*) and have their RX timestamping enabled (*bEnableRXTimeStamp* set to True in *EZapiIF\_EthMACParams*).

The timestamp provided by the interface is in a 5-byte format which consists of 32 bits of the nanosecond value and 8 bits of the seconds value. In most cases for handling of transparent PTP, where the delta between transmit and receive time is calculated, these should be enough and the high bits of the seconds value are not required, under the sensible assumption that the frame does not reside in the NPS for more than half the wraparound time of the seconds part (128 seconds).

If the data-plane code still needs access to the high bits of the seconds part (required for end-point processing), it can obtain them using *ezdp\_get\_real\_time\_clock()* API. It will need to check if the seconds byte has wrapped around since the receive time, and if so, decrement one from the seconds high bits.

## 4.6.2 Sampling the Transmit Timestamp of an Outgoing Frame in 2-Step Mode

For a 2-step PTP Sync or Delay Request, transparent processing requires the ability to receive a time stamped confirmation notice after the frames have been transmitted by a network interface MAC. The NPS supports this requirement by way of transmit confirmation reporting jobs.

In order to request that a frame provide a transmit confirmation report, the *transmit\_confirmation\_flag*, the *timestamp\_flag* and the *transmit\_keep\_buf\_flag* fields of the frame descriptor (*ezdp\_frame\_desc*) should be set.

Data-plane code should add a 16-byte header to the packet in the following abbreviated format:

```
struct ezdp_2step_1588_header
{
    unsigned /* reserved24_31 */ : EZDP_2STEP_1588_HEADER_RESERVED24_31_SIZE;
    unsigned /* reserved24 */ : EZDP_2STEP_1588_HEADER_RESERVED24_SIZE;
    unsigned /* reserved0_23 */ : EZDP_2STEP_1588_HEADER_RESERVED0_23_SIZE;
    unsigned /* reserved32_63 */ : EZDP_2STEP_1588_HEADER_RESERVED32_63_SIZE;
    uint8_t free_bytes;
    uint8_t header_offset;
    unsigned /* reserved76_77 */ : EZDP_2STEP_1588_HEADER_RESERVED76_77_SIZE;
    unsigned class_of_service : EZDP_2STEP_1588_HEADER_CLASS_OF_SERVICE_SIZE;
    unsigned /* reserved74_75 */ : EZDP_2STEP_1588_HEADER_RESERVED74_75_SIZE;
    unsigned buf_budget_id : EZDP_2STEP_1588_HEADER_BUF_BUDGET_ID_SIZE;
    struct ezdp_buffer_desc buf_desc;
};
```

The reserved fields in the header can be used to transfer SW information to the confirmation message, such is the RX timestamp and a session classification ID.

In addition, the frame must be sent to the interface via one of two special TM queues which are configured to limit the rate towards the network interface according to the ability of the interface to generate confirmation messages, using a dedicated credit mechanism. The TM side must be the one configured as *uiTimeStampTMSide*, and the queue must be either *uiTimeStampTMFlow0* or *uiTimeStampTMFlow1* as configured in *EZapiIF\_EthIFEPparams* of the destination interface.

Since the same special queues can be used for several interfaces, typically, the EZDP\_EXPLICIT\_PSID mode is used as *packet\_switch\_mode* in *ezdp\_job\_tx\_info* when transmitting the frame, so that the data plane code can specify the full PSID entry in the *explicit\_packet\_switch\_id* field which maps directly to the destination interface.

As a result, when the frame is transmitted by the network interface MAC, a transmit confirmation report job will be created with the transmitted frame descriptor and original frame buffers with the header that has been added. The 16-byte header is stripped by the interface and not transmitted out.

The transmit confirmation job thus can be identified by frame descriptor logical ID as configured for transmit confirmation jobs and carries a confirmation receive-side job info structure.

An abbreviated form of the structure is shown here:

```
struct ezdp_job_rx_confirmation_info {
    uint8_t timestamp_sec;
    uint32_t timestamp_nsec;
};
```

The timestamp fields of the confirmation receive-side job information contain the IEEE 1588 timestamp of the transmitted frame.

In transparent handling, data plane SW will receive the RX timestamp in a dedicated SW field, and calculate the delta with the TX timestamp from the confirmation info field. Care must be taken to handle wraparound correctly. The value should be stored in a search structure to be made available to the follow-up or delay-response frames once they arrive. The confirmation frame could then be discarded.



Once the follow-up or delay-response frame arrives, data-plane code will perform classification in order to match the corresponding session, fetch the stored value and update the correction field in the frame. Optionally it may fix the UDP checksum value to reflect the change.

A special edge case needs to be handled gracefully, in case the follow-up packet arrives to the CTOP before the transmit confirmation.



Since the original frame's buffers are not released when transmit confirmation is requested, care should be taken by software to release the frame buffers, for example by unsetting the *transmit\_confirmation\_flag*, the *timestamp\_flag* and the *transmit\_keep\_buf\_flag* fields and discarding the frame.

### 4.6.3 Sampling the Transmit Timestamp of an Outgoing Frame in 1-Step Mode

For a 1-step PTP Sync or Delay Request, transparent processing requires the ability to modify the correction field in the packet by adding the processing time (delta between TX timestamp and RX timestamp) to the initial correction field value.

It is responsibility of the data plane code to add a 1-step header to the frame with the following abbreviated format:

```
struct ezdp_1step_1588_header
{
    unsigned /* reserved28_31 */ : EZDP_1STEP_1588_HEADER_RESERVED28_31_SIZE;
    unsigned correction_odd_start : EZDP_1STEP_1588_HEADER_CORRECTION_ODD_START_SIZE;
    unsigned wrap_around_condition : EZDP_1STEP_1588_HEADER_WRAP_AROUND_CONDITION_SIZE;
    unsigned inject_checksum_flag : EZDP_1STEP_1588_HEADER_INJECT_CHECKSUM_FLAG_SIZE;
    unsigned /* reserved24 */ : EZDP_1STEP_1588_HEADER_RESERVED24_SIZE;
    unsigned /* reserved16_23 */ : EZDP_1STEP_1588_HEADER_RESERVED16_23_SIZE;
    uint16_t checksum;
    uint16_t checksum_offset;
    uint16_t correction_offset;
    uint64_t correction;
};
```

In addition, the *timestamp\_flag* in the frame descriptor (*ezdp\_frame\_desc*) should be set, while the *transmit\_confirmation\_flag* and the *transmit\_keep\_buf\_flag* fields of the frame descriptor should be unset.

When the packet is transmitted out, the out-going correction field value should be calculated as follows:

$$CF_{out} = CF_{in} + (TX_{ts} - RX_{ts})$$

The data-plane code should perform a partial calculation of the above which consists of “ $CF_{in} - RX_{ts}$ ” and place this value in the correction field in the header. The TX MAC will then complete the calculation by adding the TX timestamp value to the correction field in the header and write to the frame at the offset indicated by “correction\_offset” in the header.

**Warning!** : The RX timestamp is obtained from the HW in the Timestamp format (32 bits nanosecond + 8 bits seconds) while the correction field is specified in a 64-bit Time-Interval format which consists of a scaled-nanoseconds value. Data-plane code must convert the RX timestamp to a Time Interval format prior to performing the partial calculation. This is achieved by multiplying the 8-bit seconds value by  $10^9$ , adding the nanosecond value, and scaling by  $2^{16}$  (shifting 16 bits to the left).

Since the RTC value is sampled both on RX and TX as a 5-byte value, special handling needs to be performed in order to protect from wraparound of the seconds byte. For this purpose, the data plane code



must indicate in the 1-step header whether or not the RX RTC value is larger than half the possible value. This can be done by simply copying the MSBit of the seconds value to the *wrap\_around\_condition* flag. When this bit is set, and the HW notices on transmission that the RTC value is smaller than half the possible value, it will know it needs to add a value of 256 seconds to the calculated correction field. As long as the PTP frame does not reside in the device for a period larger than 128 seconds, the calculation is correct.

Optionally, the UDP checksum can be calculated and fixed by the transmitting MAC HW as well. If this is required, the *inject\_checksum\_flag* bit in the header should be set. In addition, data-plane code will need to prepare the partial checksum calculation in the *checksum* field. This should contain the checksum calculation of the UDP field when the correction field is set to zero. Assuming that the original checksum was correct, this can be achieved by the *ezdp\_sub\_checksum()* API with the original checksum and the original correction field value. Also the checksum field offset in the frame needs to be provided in the *checksum\_offset* field, so that the HW will write the updated checksum value at the correct offset. Finally, the *correction\_odd\_start* bit is used to indicate whether or not the correction field starts at an odd offset compared to the checksum value, which would have an effect on the polarity of the checksum calculation. In the standard IEEE 1588, this bit will most likely always be unset.

## 4.7 Traffic Management

The NPS Traffic Manager (TM) is a configurable, hardware, flow-based, bandwidth control and virtual output queuing service. The TM maintains millions of frame queues in memory and controls bandwidth constraints and head of line blocking by intelligently scheduling the queue depletions into the network interfaces or loopback one or many copies of the frame into one of the PMU queues.

The NPS is equipped with two TM engines, one on each side of the SoC. The two engines work independently of each other; each with its own distinct set of flows and each of the engines can schedule frames into network interfaces on both sides of the SoC.

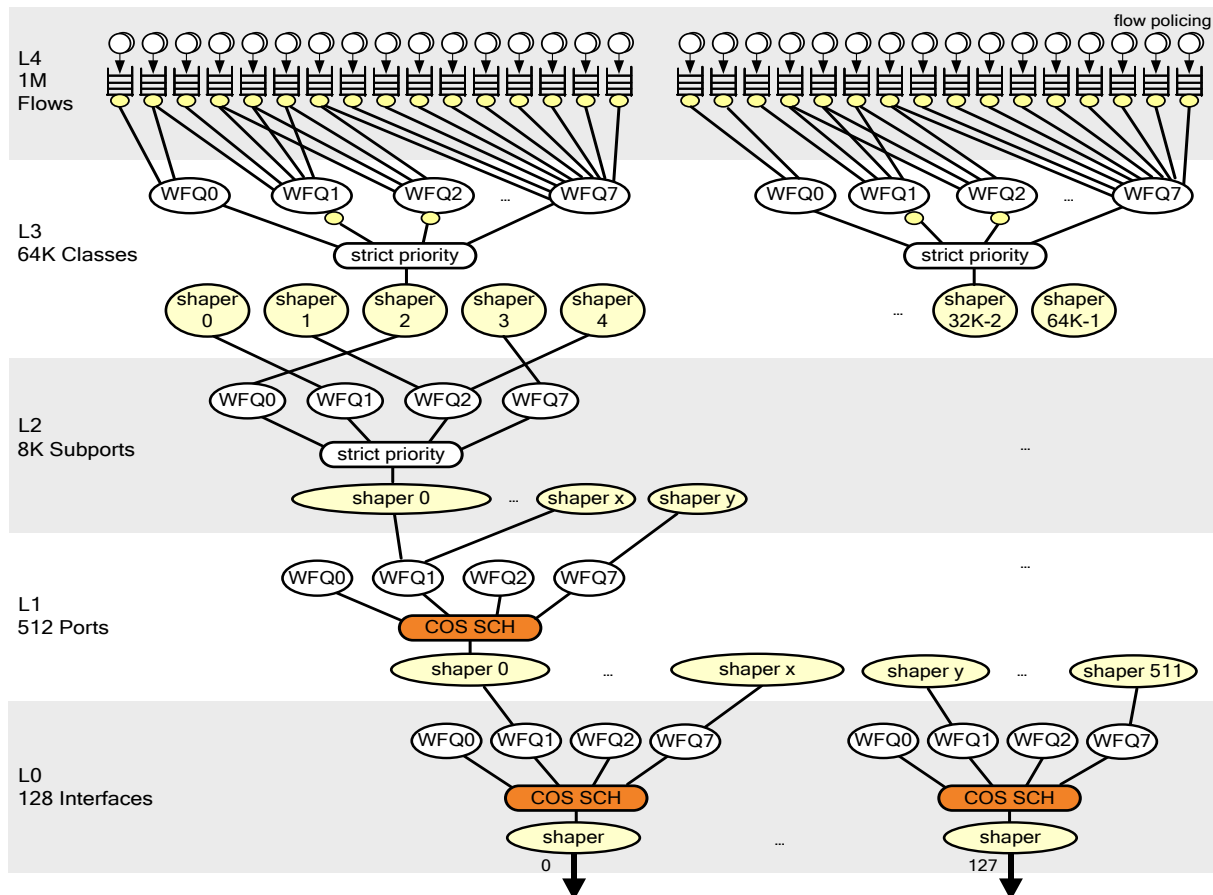
The TM can be thought of as providing two distinct but related functions: QoS scheduling and output channel selection. Normally both functions are used together. However, a bypass mode is also available in which the TM only supplies output channel selection services and performs no QoS scheduling functions.

### 4.7.1 QoS Scheduling

Each TM engine provides one million virtual out queues, called flows, and schedules their depletion into the appropriate output channels based on the frame data, the queues' congestion levels and the TM configuration.

Based on the TM configuration, frame size, metadata and the congestion level of the queues, the TM selects when to dequeue the frame and send it to its destined output channel or to discard it.

**Figure 17. NPS TM QoS scheduling**



## 4.7.2 Output Channel Selection

Once a TM has dequeued a frame, it uses information from both the frame metadata and its configuration to send it to an output channel. An internal data structure called the Packet Switch table is used by the TM to select the output channel.

Each entry in the Packet Switch table holds the ID of one output channel. Multiple entries may point to the same output channel or to different ones. The output channel might be connected to a physical network interface or be a logical output channel which may loopback the frame, optionally replicated many times, back to a designated PMU queue.

Output channels numbered from 0 to 127 belong to physical network interfaces, while those numbered from 128 to 143 are logical loopback channels.

The TM logic picks an index into the Packet Switch table and sends the frame to the output channel specified in the Packet Switch table entry at the selected index.

Three modes are available for the Packet Switch index selection: topology-based selection, fixed selection and explicit packet-switch index selection.

The following fields of the `ezdp_job_tx_info` structure are used to specify the packet-switch index:

`packet_switch_mode` specifies the mode, and Packet Switch ID `select / explicit_packet_switch_id` specifies the packet-switch index.

### 4.7.2.1 Topology-based packet-switch index selection mode

The topology-based packet-switch index selection mode uses the L1 and L0 scheduling levels configured for the specific flow to choose a base index (which is always a multiple of 8) and uses software supplied 3 bit “Packet Switch ID select” metadata to choose an offset from that base index.

This is the most commonly used selection mode. Since the selection criterion of this mode uses the QoS scheduling hierarchy, this mode is not available when the QoS scheduling function is bypassed.

### 4.7.2.2 Fixed packet-switch index selection mode

The fixed packet-switch index selection mode uses a fixed base index which is part of the TM configuration as the base index and uses the software supplied 3 bit “Packet Switch ID select” as an offset from that base index.

This mode is typically used when using the TM to loopback frames back to the NPS PMU queues or when a simplistic 8 output channel selection is desired.

### 4.7.2.3 Explicit packet-switch index selection mode

The explicit packet-switch index selection mode uses a Packet Switch table index explicitly provided by the software for the frame. Explicit mode is normally applied to the 1588 2-step related packets which use specific TM flows and are supposed to be sent to any channel.

## 4.7.3 Sending a Frame to the TM

Sending a frame to be scheduled by one of the TM engines is accomplished in two steps: filling up the TM information data structure in the frame job descriptor, and instructing the PMU to send the frame to the TM.

### 4.7.3.1 TM information

The following is an abbreviated listing of the TM information data structure and related definitions:

```
enum ezdp_tx_packet_switch_mode {
    EZDP_TOPOLOGY,
    EZDP_FIXED_BASE,
    EZDP_EXPLICIT_PSID
    ...
};

enum ezdp_tx_drop_mode {
    EZDP_CAN_DROP,
    EZDP_DONT_DROP,
    EZDP_NEVER_DROP
};

struct ezdp_job_tx_info {
    unsigned wred_color;
    unsigned packet_switch_id_select;
    union {
        uint16_t replication_count;
        uint16_t user_info;
        uint16_t explicit_packet_switch_id;
        uint16_t dest_queue;
    };
    enum ezdp_tx_drop_mode drop_mode;
    unsigned qos_bypass;
    enum ezdp_tx_packet_switch_mode packet_switch_mode;
    unsigned side;
    unsigned flow_id;
    ...
};
```

Here is a description of the fields' meanings:

- **WRED color** – the WRED color field is used to set the frame color parameter used by the TM to compute the Weighted Random Early Drop congestion avoidance algorithm verdict for the frame.
- **Packet Switch ID select** – the Packet Switch ID select field is used to set the offset from the base index computed by the TM which will be used to determine the packet switch table index used to select the output channel for sending the frame. See the section [Output Channel Selection](#).
- **Replication count** – the replication count field is used to indicate the number of frames replicas, or copies, when frame multicast is used. See the section [Multicast and Replication](#) ahead for more details.
- **User info** – the user info field is used to set a user defined field which is echoed back to software in non-replicated loopback frames, if thus configured. See the section [Frame Loopback](#) ahead for more details.
- **Explicit Packet Switch ID** – the explicit Packet Switch ID field is used to set the explicit packet switch table index which will be used when the explicit packet-switch index selection mode is selected. See the section [Output Channel Selection](#).
- **Destination queue** – the destination queue field is used to set the PMU queue into which a loop backed frame will be queued, if thus configured. See the section [Frame Loopback](#) ahead for more details.

- **Drop mode** – the drop mode field is used to set the policy the TM should adhere to when considering the frame for drop. The available modes are:
  - **Drop allowed mode** – the TM is allowed to drop the frame either to satisfy WRED congestion avoidance considerations or due to explicit control plane request to flush the queue.
  - **Do not drop mode** – the TM is only allowed to drop the frame due to an explicit control plane request to flush the queue.
  - **Never drop mode** – the TM is not allowed to drop this frame under any circumstances.
- **QoS bypass flag** – the QoS bypass Boolean flag allows to indicate that the QoS scheduling logic of the TM will not applied to this frame and only output channel selection will be performed.
- **Packet Switch mode** – the Packet Switch mode field is used to set the packet-switch table index selection mode used for output channel selection, as described in [Output Channel Selection](#) above.
- **Side** – the side field is used to select the TM engine to which the frame will be sent. Both TM engines may schedule and output frames destined to network interfaces on both sides of the SoC.
- **TM Flow ID** – the flow ID field is used to select the L4 entity to which the frame will be queued.
- **Stat\_stream\_id** – Statistic stream ID field. Relevant when TM accounting is performed according to the stream ID passed in each frame's TM header. (Refer to “TM Statistics” in the *NPS-400 Architectural Specifications*.)
- **Wred\_flow\_template\_profile / Wred\_class\_template\_profile / Wred\_flow\_scale\_profile / Wred\_class\_scale\_profile** – WRED template and scale profiles to be used at TM L3 and L4 for this frame. (For more details refer to “Hierarchical Dropping” in the *NPS-400 Architectural Specifications*.)

#### 4.7.3.2 Finalizing the job

Once the TM information structure is filled, sending the frame to the TM is done using the `ezdp_send_job_to_tm()` API call:

```
void ezdp_send_job_to_tm( ezdp_job_id_t __cmem *jobh_ptr, struct ezdp_job_desc __cmem *jd_ptr,
uint32_t side, uint32_t flags)
```

All the parameters are identical to the parameters of the `ezdp_send_job_to_interface()` API call we have described above, with the exception of the side parameter which indicates the TM engine instance to which the frame will be sent.

Here is a usage example:

```
job_desc.tx_info.flow_id = 512;
job_desc.tx_info.side = 0;
job_desc.tx_info.qos_bypass = 0;
job_desc.tx_info.packet_switch_mode = EZDP_TOPOLOGY;
job_desc.tx_info.packet_switch_id_select = 0;

ezdp_send_job_to_tm(&job, &job_desc, 0, 0);
```

## 4.7.4 Frame Loopback

In addition to queuing frames for transmission out the network interfaces, the TM also supports the ability to redirect frames back towards the NPS PMU queue.

This loopback ability, from a PMU queue to the TM and back into the PMU, allows the establishment of complex software pipeline architecture, where flow-based bandwidth control and virtual queue services are provided between the different data plane applications, or stages of the software pipeline.

Up to 16 loopback interfaces can be configured. Logical output channels 128 to 143 are reserved for loopback interfaces.

There are two modes for associating a channel with a loopback interface: loopback interface per COS (up to 4 output channels are used by loopback interfaces), and loopback interface per output channel (up to 16 output channels are used and all COSs are routed to the same loopback interface).

Frames sent to the TM which are directed to a packet switch table entry will be looped back into one of the PMU queues on the same side as the TM.

The PMU queue into which looped back frames are queued is determined by the loopback interface configuration: either a pre-selected queue based on the frame's COS, or the queue number indicated in the *dest\_queue* field of the TM information structure.



Run time PMU queue selection is only available for unicast frames.

## 4.7.5 TM IMEM Data Cache

The TM processing may have large latencies and in some cases it may be necessary to limit the number of IMEM buffers pending in the TM. NPS has a mechanism by which an application may limit the number of IMEM buffers used by the traffic forwarded to a specific set of the ports, which are being processed by TM. For each of the external ports that use this mechanism the following should be done:

- At the CP this mechanism should be enabled on each of the required ports.
- At the DP: Whenever forwarding a packet to a port configured with this mechanism, the application needs to decide whether to relocate buffer from IMEM to EMEM to address potential latency issues. An application normally will make the decision according to the:
  - Current number of buffers used by this mechanism (by calling `ezdp_read_tm_imem_buf_ctr`)
  - Expected frame's latency (the TM destination flow congestion level can be retrieved by `ezdp_get_tm_queue_depth`)
  - Frame's priority

If the application decides to use the IMEM buffers for the given packet, the application needs to update the counter by calling `ezdp_inc_tm_imem_buf_ctr`. If the decision is not to use the IMEM buffers, the application needs to ensure that all the buffers of the packet reside in EMEM.

## 4.8 Multicast and Replication

A common requirement of many network processing applications is the necessity to send copies of a frame to multiple destinations.

NPS provides the ability to share frame buffers by multiple frame replicas and supports two methods for multicasting: lightweight multicast (SW based) and heavyweight multicast (TM loopback based).

In addition the NPS provides two features in supports of this requirement: multicast reference counters and TM loopback frame replication.

### 4.8.1 Multicast Reference Counters

Normally, a frame buffer is only referenced by a single frame at a time. This statement ceases to be true, however, when dealing with multicast frames.

Multicast frames are frames that need to be replicated to service a network broadcast or multicast service. Often, the different replicas of the frame have the same data with the exception of a few header fields.

In order to support multicast frames in an efficient manner, without requiring the creation of a separate copy of all the buffers for each frame replica, the NPS supports multicast reference counters.

Each NPS frame buffer has a reference counter associated with it. When frames are transmitted or discarded, the network DMA engine atomically checks and decrements the multicast reference counters of each frame buffer that may be in use by more than one frame. These buffers will only be released when that multicast reference counter indicates that the frame buffer is no longer used by any frame.

As most frames are not multicast frames, the NPS provides the option of marking frames as multicast frames and therefore having their frame buffer multicast reference counters checked and updated via the frame multicast control status.

#### 4.8.1.1 Frame multicast control

Each frame in NPS has a multicast control status associated with it in the frame descriptor *multicast\_control* field. Three such statuses are supported: unicast, multicast and broadcast.

- **Unicast** – The frame is a unicast frame. The frame buffers are private. No multicast reference count check is performed when the frame is transmitted or discarded and all frame buffers will be released unconditionally.
- **Broadcast** – the frame is a broadcast frame. Any of the frame's buffers may be referenced by another frame descriptor. The multicast reference counter will be consulted and updated when a frame buffer is released. Buffers will only be released during transmit or discard if their reference count is zero.
- **Multicast** – the frame is a multicast frame. The frame's first data buffer, and for extended type frames, the frame's linked buffer descriptor buffer are private. All other frame buffers may be referenced by other frame descriptors and multicast reference is implemented for them.

#### 4.8.1.2 Setting frame buffer multicast reference counter


To set the multicast reference counter of a frame data buffer use the *ezdp\_write\_mc\_buf\_counter()* API call:

```
void ezdp_write_mc_buf_counter( ezdp_buffer_desc_t  bd, uint16_t value)
```

The call sets the multicast reference counter of the buffer descriptor *bd* to the designated value.

### 4.8.1.3 Additional multicast reference counter operations

Multicast reference counters can also be read and atomically incremented or decremented using the multicast reference counter API. Wrapper functions are also available for allocating and releasing frame buffers with their multicast reference counter already preset.

 For a full list of the multicast reference counter APIs, including what information is provided by each of them, refer to the “Multicast Reference Counters” section in the “Frame Buffer Management (ezdp\_frame.h)” section in the *EZdp Reference Manual*.

## 4.8.2 Creating a replica specific buffer

In both multicast methods (lightweight and heavyweight), the buffers of a frame are normally shared by all replicas. Whenever some of the buffers hold replica-specific data, the application needs to update the frame accordingly. Typically the differences between the copies are only in the first buffer.

Steps to switch to a private first buffer are:

1. Allocate a new buffer.
2. Copy the data to the new buffer and make the required changes.
3. Free the old buffer (decrement the Multicast counter).
4. In case of extended frame, clone LBD and update first BD.
5. Update first BD in FD.

## 4.8.3 Heavyweight Multicast (HW/TM Multicast)

The heavyweight multicast frame replication feature allows sending a frame to one of the TM loopback interface channels and requesting that a designated number of frame replicas, or copies, be looped back by the TM into the PMU queues.

The replication is logical (i.e. nonphysical) – all the returned frame replicas reference the same frame buffers, which are not copied.

### 4.8.3.1 Requesting replication service

To have the TM replicate a frame, first set the multicast reference counters of all the frame buffers to the number of desired frame copies.

Next, set the job descriptor TM information structure *replication\_count* field to the number of desired copies and set the *multicast\_control* field of the frame descriptor to the value *EZDP\_REPLICA*.

Finally, send the frame for TM processing to one of the loopback interface channels. The TM will queue the requested number of frame replicas into the PMU queue designated by the TM loopback interface.

### 4.8.3.2 Receiving replicated frames

Each replica created by the TM will be received as a new job by one of the NPS threads. Each of these will include a frame descriptor for one of the frame replicas and will carry a job descriptor receive-side information structure of a loopback job.

An abbreviated listing of this data structure is shown below:

```
struct ezdp_job_rx_loopback_info {
    uint16_t replication_id;
};
```

The replication ID field in the loopback job receive-side information contains the replica number of the specific frames. With the exception of the replication ID, all the frame replicas are identical.



To illustrate, when N replicas are requested, N frames will be generated and the replication ID field of each replicate will carry a number from 0 to N-1 denoting the specific replica.

In case there is replica-specific data, it is necessary to switch to private buffer as explained in the [Creating a replica specific buffer](#) section.

## 4.8.4 Lightweight Multicast

The lightweight multicast method can be used to create up to seven replicas of the frame. In this method the SW is responsible for creating a batch of jobs (packed into Job Container) and sending the Job Container to the PMU. (The PMU unpacks the Job Container and creates a Job per replica.)

Steps to be performed to implement this method:

1. Set the Multicast counter to the number of replicas (for all the buffers of the frame).
2. For each replica (up to 7):
  - 2.1. Allocating a new job and add to the container.
  - 2.2. In case of a replica specific data, switch to a private buffer (as explained in the [Creating a replica specific buffer](#) section).  
Alternatively this could be done in context of processing of a specific replica.

## 4.8.5 Multicast Example

For an example of frame multicast, please refer to the [NPS-400 L23QoS reference application](#) available from the EZchip customer web portal and its accompanying application note.

## 4.9 General Purpose Pools

### 4.9.1 Index Pool

NPS supports up to 64 general purpose independent index pools (32 pools per side). Each pool can manage a minimum of 1K indexes and up to  $2^{32} - 1$  indexes.

Configuration of an index pool is achieved by:

**EZapiChannel\_ConfigCmd\_SetIndexPoolParams** (*uint32\_t pool, bool enabled, bool search, uint32\_t startIndex, uint32\_t numIndexes, bool skipPoolInit, , uint32\_t cacheSize*)

**EZapiChannel\_ConfigCmd\_SetIndexPoolParams()** is used to configure up to 32 index pools per side. Each index pool is identified with a *pool ID* (0-31 for side 0 and 32-63 for side 1). Each of these index pools is assigned with number of indexes (*numIndexes*) and a start index (*startIndex*). *numIndexes* + *startIndex* must be lower than  $(2^{32} - 1)$ . Each pool can be initialized as empty by turning on the *skipPoolInit* flag.



Pools can be configured to be used by *search* hash structures. To indicate this, the search flag should be turned on. When the search flag is on, all parameters except *cacheSize* are not applicable. For more info see the [Advanced Search Features](#) section.

Allocating and releasing indexes from the index pools achieved by the following EZdp API. All allocations and releases are atomic.

**uint32\_t ezdp\_alloc\_index**( *uint32\_t pool\_id* )

The *ezdp\_alloc\_index()* function allocates and returns one index from pool *pool\_id*. In case there are no indexes to allocate, EZDP\_NULL\_INDEX will be returned.

**void ezdp\_free\_index**( *uint32\_t pool\_id, uint32\_t free\_index* )

The *ezdp\_free\_index()* function releases index *free\_index* from pool *pool\_id*.

The rate of allocations and releases of indexes can be increased by allocating and releasing indexes in groups of up to 8 indexes. These operations are also atomic.

**uint32\_t ezdp\_alloc\_multi\_index**( *uint32\_t pool\_id, uint32\_t num\_of\_indexes, uint32\_t \_\_cmem \*index\_array\_ptr* )

**void ezdp\_free\_multi\_index**( *uint32\_t pool\_id, uint32\_t num\_of\_indexes, uint32\_t \_\_cmem \*index\_array\_ptr* )

The *ezdp\_alloc\_multi\_index()* and *ezdp\_free\_multi\_index()* functions allocate/release *num\_of\_indexes* index from pool *pool\_id*. Up to 8 indexes can be allocated/released in one operation. The indexes are written/read to/from *index\_array\_ptr* array in CMEM. And the first allocated index is returned. Multi allocation will fail if it is unable to allocate all the required indexes.

EZDP\_NULL\_INDEX is returned in case of allocation failure.



Currently in NPS only 3 indexes can be allocated in the multi-index allocation operation.

Some data plane applications might need to retrieve the current number of free indexes in a pool. This operation is not atomic.

***uint32\_t ezdp\_read\_free\_indexes( uint32\_t pool\_id )***

The *ezdp\_read\_free\_indexes()* function returns the number of indexes that are free and available for allocation in pool *pool\_id*.

## 4.9.2 Memory Pool

A memory pool is essentially an index pool coupled with a chunk of memory (represented by the memory's base address). In the current release there is no EZcp APIs to initialize the memory pool, therefore, in order to configure a memory pool one needs to configure an index pool (see the [Index Pool](#) section) and allocate a memory chunk in IMEM or EMEM (see the [Memory Allocation](#) section).



The size of the allocated memory chunk must be of number of indexes \* size of the element in pool.

A memory pool is referred/accessed in a DP application through a variable of *ezdp\_mem\_pool\_t* type. This type of variable must be used for every memory pool in the DP application. *ezdp\_mem\_pool\_t* has to be initialized with the base address of the memory chunk, the index pool ID to be used, and the size of each memory element in the memory pool. This initialization is done using a dedicated configuration structure.

***void ezdp\_init\_memory\_pool(ezdp\_mem\_pool\_t \*mem\_pool, struct ezdp\_mem\_pool\_config \*config )***

The *ezdp\_init\_memory\_pool()* function initializes the memory pool pointed by *mem\_pool* with the configuration pointed by *config*. The configuration must include the base address of the memory chunk that the memory pool will use, the pool ID of the pool to be used, and the size of each object in the memory pool in bytes.



The base address is supplied as a summarized address with entry size equal to object size.

```

/*! memory pool configuration data structure */
struct ezdp_mem_pool_config
{
    struct ezdp_sum_addr    base_addr;
    /**< The start address of the memory */
    uint16_t                index_pool_id;
    /**< BMU index pool id to be used by memory pool.
     * NOTE: Pool id which should be configured/enabled in NPS */
    uint16_t                obj_size;
    /**< The size of the memory object. */
};

```

After initialization, the memory pool can be used to allocate and release memory objects of the configured size. The memory objects are referred to by a summarized address.

***ezdp\_sum\_addr\_t ezdp\_alloc\_obj(ezdp\_mem\_pool\_t \*mem\_pool)***

The *ezdp\_alloc\_obj()* function allocates and returns one memory object from memory pool *mem\_pool*. In case there are no indexes to allocate, a null summarized address will be returned (may be checked with *ezdp\_is\_null\_sum\_addr()* API). The returned address's entry size is the configured object size.

***void ezdp\_free\_obj(ezdp\_mem\_pool\_t \*mem\_pool, ezdp\_sum\_addr\_t free\_obj )***

The *ezdp\_free\_index()* function releases memory object *free\_obj* from memory pool *mem\_pool*. The supplied address of the object to release must be with entry size equal to the configured object size.

Just like with index pools, some data plane applications might need to retrieve the current number of free memory elements in the memory pool.

***uint32\_t ezdp\_read\_free\_objs(ezdp\_mem\_pool\_t \*mem\_pool)***

The *ezdp\_read\_free\_objs()* function returns the number of memory objects that are free and available for allocation in memory pool *mem\_pool*.

## 4.10 Mathematical Operations

### 4.10.1 Logical, Arithmetical and Bit Manipulation Operations


The NPS is equipped with a large number of arithmetical and logical bit operations that are implemented as single CTOP instructions.

These operations include:

- Arithmetic addition and subtraction operations on selected bit ranges of integers.
- Logical and, not, or, xor and folded xor operations on selected bit ranges of integers.
- Bitwise set, unset, merge, move, split, combine, shift and extraction operations on multiple bits or a set of bits of an integer.
- Special mathematical operations such as counting number of bits, calculating a power of 2, find first one/zero and reflect bits.
- Optimized 8 bit value to 4 bit modulo and divide operations.

Optimized move bits and bitwise set operations are used by the compiler to optimize operation on bits. This behavior is controlled by the `-mbitops` compiler flag.

Other optimized operations on bits, module, div and special mathematic operation can be explicitly used in an application to allow highly efficient implementation of algorithms such as lookup table key computations, load balancing target selection with low instruction and clock to operation ratios.

 For the full details on the mathematical operations API, refer to the “ALU Operations (ezdp\_math.h)” section in the *EZdp Reference Manual*.

### 4.10.2 Hash, CRC and Internet Checksum Computation

NPS provides optimized API routines for calculating hash, CRC and checksum (calculation and update).

#### 4.10.2.1 Hash

NPS provide various hash calculation APIs.

The first API is single cycle CTOP hash calculation. It computes an up to 32 bit hash value on up to 64 bits of data in a single cycle. 16 different hash functions are supported (2 fixed matrixes with 8 permutation flavors).

To perform such calculation, use `ezdp_hash32` and `ezdp_hash64` API calls:

```
uint32_t ezdp_hash32( uint32_t src, uint32_t hash_size, uint32_t permute_id, uint32_t base_matrix )
uint32_t ezdp_hash64(uint32_t src1, uint32_t src2, uint32_t hash_size, uint32_t permute_id, uint32_t
base_matrix )
```

The `ezdp_hash32/64()` computes an up to 32-bit hash value based on 32-bit or 64-bit data provided in the `src` arguments. The hash function is determined by selecting a `base_matrix` and a `permutation` flavor and output hash value width is controlled by `hash_size`.

Another API is activating a hash accelerator to compute a hash value on up to 64B of data. Only one fixed hash flavor is possible in this API and the hash value width is always 32 bits.

To perform such a calculation, use the `ezdp_bulk_hash` API call:

```
uint32_t ezdp_bulk_hash( uint8_t __cmem *data, uint32_t size )
```

The `ezdp_bulk_hash()` invokes hash accelerator on `data` on CMEM with the provided `size` (up to 64B).

In addition, hash value on Destination MAC + Source MAC and Destination IP + Source IP are returned as part of the `ezdp_decode_mac`, `ezdp_decode_ipv4` and `ezdp_decode_ipv6` results.



Additional hash functions exist when activating a security engine, see [Security](#).

#### 4.10.2.2 CRC

NPS supports CRC calculation using the standard polynomial in one cycle. 16-bit and 32-bit CRC calculations are supported.

**`uint32_t ezdp_calc_crc16( uint32_t crc_value, uint8_t input_value, bool input_value_bit_rflt )`**

**`uint32_t ezdp_calc_crc32( uint32_t crc_value, uint8_t input_value, bool input_value_bit_rflt )`**

The `ezdp_calc_crc16/32()` computes a 16/32-bit CRC based on previous (or initial) 16/32-bit `crc_value`. `input_value` is the input byte that may optionally be reflected before CRC calculation using the `input_value_bit_rflt` flag.

For a 16-bit CRC, the polynomial coefficients are 0x1021 and for a 32-bit CRC the polynomial coefficients are 0x04c11db7.

These operations, together with bit manipulation operations (like `ezdp_reflect_bits()`, see [Mathematical Operations](#)) serve as building blocks for complete implementations of CRC algorithms on a buffer.

The following example is an implementation of a flavor of CRC32 calculation over a buffer. This CRC-32 algorithm flavor is reportedly used in PKzip, AUTODIN II, Ethernet, and FDDI. For this flavor, the initial value is 0xffffffff, the input byte are reflected and the output is reflected and XORED with 0xffffffff.

```
uint32_t calcCRC32(uint8_t *data, uint32_t size)
{
    uint32_t crc = 0xffffffff;
    uint32_t i;
    for(i=0;i<size;i++)
    {
        crc = edp_calc_crc32(crc,data[i],true);
    }
    crc = ezdp_reflect_bits(crc, EZDP_REFLECT_RESOLUTION_4_BYTE);
    crc = crc ^ 0xffffffff;    return crc;
}
```

#### 4.10.2.3 Checksum

NPS provides various APIs for efficient checksum calculations or updates.

The first group of functions is checksum calculation of streams of data which invoke the checksum calculator accelerator.

To calculate a checksum for data on CMEM, use the `ezdp_calc_checksum` API call:

**`uint32_t ezdp_calc_checksum( void __cmem *ptr, uint32_t size )`**

The `ezdp_calc_checksum()` invokes a DMA to calculate the checksum of the data in `ptr` (on CMEM) with the provided `size`.

To calculate a checksum for data on IMEM/EMEM, use `ezdp_calc_checksum_ext_addr` API call:

**`uint32_t ezdp_calc_checksum_ext_addr(struct ezdp_ext_addr __cmem *src_ptr, uint32_t size, uint32_t flags )`**

The `ezdp_calc_checksum_ext_addr()` invokes a DMA to calculate the checksum of the data on internal or external memory where the address is provided by `src_ptr` with the provided `size`.

To calculate a checksum for frame data, pointed by a BD, use `ezdp_calc_frame_data_checksum` API call:

```
uint32_t ezdp_calc_frame_data_checksum( struct ezdp_buffer_desc __cmem *bd_ptr, uint32_t
bd_offset, uint32_t size, uint32_t flags )
```

The `ezdp_calc_frame_data_checksum()` invokes a DMA to calculate the checksum over the frame data in `bd_ptr` (on CMEM) from the provided `bd_offset` with the provided `size`.

A second group of checksum API functions provide incremental update (add or subtract) of an already calculated checksum

To add or subtract a value from an already calculated checksum, use `ezdp_add_checksum` and `ezdp_sub_checksum` API calls:

```
uint32_t ezdp_add_checksum( uint32_t checksum_value, uint32_t add_value )
```

```
uint32_t ezdp_sub_checksum( uint32_t checksum_value, uint32_t sub_value )
```

The `ezdp_add/sub_checksum()` are used for incremental updates of checksums by adding or subtracting values (`add_value` or `sub_value`) from a `checksum_value`.

These APIs provide a more efficient way for checksum update, instead calculating the checksum from scratch using the `ezdp_calc_checksum()` API call, single cycle `ezdp_add_checksum()`, `ezdp_sub_checksum()` API call can be used instead.

For example, for an IP header checksum update after reducing the TTL value by 1 (when forwarding through a router), the checksum might be calculated again using `ezdp_calc_checksum()` over the updated IP header (assuming the header is on CMEM) or, alternatively in a more efficient way, by subtracting 0x100 from the checksum in the IP header.

Here is a code implementing an IP header checksum update due to TTL decrement:

```
/* update TTL field */
ipv4_header->ttl--;

/* subtract 0x100 from checksum (TTL affects the MSB of the 16bit checksum, see IP
header format) */
ipv4_header->header_checksum = ezdp_sub_checksum(ipv4_header->header_checksum,
0x100);
```

In addition to the above APIs, NPS calculates the checksum over the entire frame data when receiving packets from the external interfaces. The checksum result is written into `gross_checksum` field in `ezdp_job_rx_info` info struct of the received job (see [Job Information](#)). This information can be used to validate the correctness of the entire payload, for example a TCP checksum.

Here is an example which demonstrates the implementation of validating a TCP checksum using `gross_checksum` and checksum calculation APIs.

```
/* Get gross checksum of frame from RXinfo */
checksum = jd.rx_info.gross_checksum;

/* Subtract L2 checksum from checksum 16bits at a time (can be optimized) */
#define L2_SIZE sizeof(struct l2_hdr)
int32_t i;
for(i=0;i<L2_SIZE-2;i+=2)
{
    checksum = ezdp_sub_checksum(checksum, *(uint16_t*)(buffer+i));
}

/* Subtracting IPv4 checksum from checksum is not necessary, assuming IPv4 checksum
is calculated correctly. Otherwise, a similar code of the above would come here */

/* At this point the checksum variable contains the checksum for TCP header and
payload. Need to add the pseudo header */
#define TCP_PROTOCOL 0x6
checksum = ezdp_add_checksum(checksum, total_length);
```

```
checksum = ezdp_add_checksum(checksum, source_ip);
checksum = ezdp_add_checksum(checksum, destination_ip);
checksum = ezdp_add_checksum(checksum, TCP_PROTOCOL);

/* If the resulting checksum is zero then the TCP checksum is validated */
if(checksum ==0)
{
    // TCP checksum validated
}
else
{
    // TCP segment is corrupted
}
```

In addition, a checksum calculation is returned by most of the DMA operations, for more info (see [DMA and Memory Operations](#)).



In all APIs checksum calculations assume an even (2 byte aligned) pointer, if the pointer is odd, the checksum result should be swapped.



## 4.11 Security

NPS includes dedicated security HW engines accelerating common cryptographic algorithms. The security accelerators address the network application requirement to support reliable communications that guarantee integrity and confidentiality. The NPS includes 12 acceleration security engines per CTOP cluster which provide security acceleration for the CTOP thread running on the cluster.

The table below summaries the HW capabilities for supported ciphers and hash functions.

Standard	Modes	Key Size	Rate per Cluster	Total Rate
<b>Cipher</b>				
AES	ECB/CBC/CFB/ OFB/CTR	128 bit	17.45 Gbps	279.27 Gbps
		192 bit	14.77 Gbps	236.31 Gbps
		256 bit	12.8 Gbps	204.8 Gbps
	GHASH	128/192/256 bit	21.33 Gbps	341.33 Gbps
	GCM	128 bit	17.45 Gbps	279.27 Gbps
		192 bit	14.77 Gbps	236.31 Gbps
		256 bit	12.8 Gbps	204.8 Gbps
	CCM	128 bit	8 Gbps	128 Gbps
		192 bit	8.86 Gbps	109.71 Gbps
		256 bit	6 Gbps	96 Gbps
DES/2DES/3DES	ECB/CBC/ CFB/OFB/CTR	64/128/192 bit	5.65 Gbps	90.35 Gbps
<b>Hash functions</b>				
MD5			21.94 Gbps	351.09 Gbps
SHA1			18.29 Gbps	292.57 Gbps
SHA2	SHA2-224 SHA2-256		22.59 Gbps	361.41 Gbps
	SHA2-384 SHA2-512		18.73 Gbps	299.71 Gbps



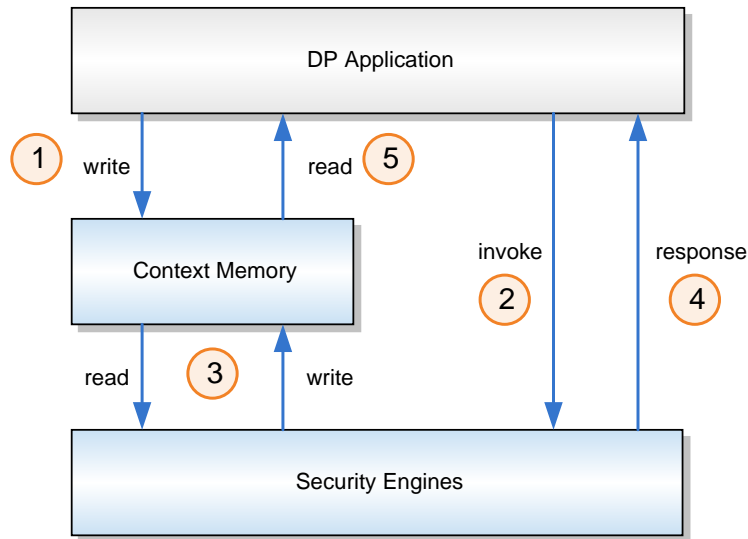
CryptoLib is the component (library) of the EZchip [IPsec sample application](#) that handles the actual encryption/decryption and authentication of the frame according to the IPsec protocol. It utilizes the NPS-400 architecture and accelerators to perform cryptographic tasks efficiently.

### 4.11.1 Architecture

The security architecture is very straightforward and contains security HW engines and security context memory. While the engines are used for security tasks, the context memory stores the state of the task.

The following figure outlines the relationship between the data plane application, security context memory and security engines and typical security task of the data chunk.

**Figure 18. Relationship between NPS security components**



At the beginning of each security task (encryption, decryption or digestion of data), the context memory is initialized by the DP application (marked **1** in the figure). Then the relevant security engine is invoked for each data chunk (marked **2** in the figure). The engine will read the current state of the session from the context memory, process the data and will store the new state to context memory (marked **3** in the figure).

In case of an encryption or decryption operation, the response (cipher text in case of encryption and plain text in case of decryption) is also written to CMEM (marked **4** in the figure). In case of a digest operation, the MAC is written to context memory as part of the session state.

The DP application may read from context memory at any time (not necessarily at the end of the session) and retrieve any information (marked **5** in the figure).

Security tasks are broken into chunks of data. Each chunk must be a multiple of the algorithm's native block size and is limited to up to 128B.

#### 4.11.1.1 Context Memory

As described above, context memory is the memory that stores the state of a security task. This state is initialized by the DP application at the beginning of the session (see section below) and updated by the security engines as data is being processed.

The state of the security task can be read and restored at any time to allow maximum utilization of the security engine for a multi-packet process. The state of a security task depends on the selected algorithm.

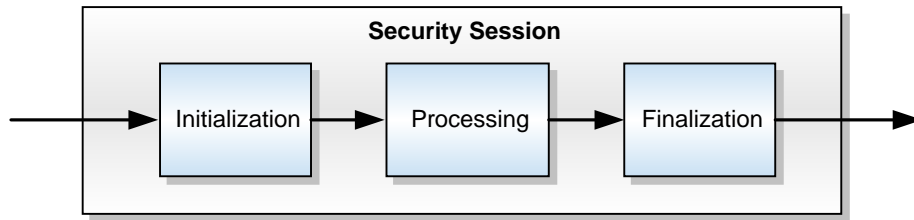
The context memory also stores the final MAC calculated for digest sessions.

Up to 128 security contexts per cluster are supported. Before invoking any security operation a thread must allocate a security context ID (0...127) that will be unique per cluster. It is the responsibility of the SW to allocate context IDs to threads in the cluster level, as well as to release context IDs when a security session is terminated.

## 4.11.2 Launching a Security Task

A security session has three general stages that must be performed by the DP application:

**Figure 19. Security session diagram**



Each stage depends on the desired algorithm to be executed.

Accessing the security task from data plane is done using *struct ezdp\_security\_handle* which contains security context ID (see [Context Memory](#)) and security algorithm type.

Here are the essential parts of the definition of the security handle descriptor as taken from *ezdp\_security\_defs.h* (some non-user serviceable fields have been omitted for clarity):

```

struct ezdp_security_handle
{
    ...
    enum ezdp_sec_alg    alg_type    : EZDP_SECURITY_HANDLE_ALG_TYPE_SIZE;
    ...
    uint8_t              context_id;
    ...
};
  
```

*contextID* is a number between 0 and 127 that indicates the slot in context memory that will be used by the security task.

*alg\_type* is the algorithm of choice (for list of algorithms see *ezdp\_security\_defs.h*).

### 4.11.2.1 Digest Task

#### 4.11.2.1.1 Initialization

Special initialization of a digest task is needed only for HMAC mode. For non HMAC mode, the security context memory is initialized by processing the first data segment.

In HMAC mode, key xor i\_pad should be hashed before hashing the data. This is done using the following API:

```

void ezdp_start_hmac_calculation(ezdp_security_handle_t sec_handle, void __cmem *key_ptr,
    uint32_t size)
  
```

*ezdp\_start\_hmac\_calculation()* initializes MAC in context memory which is pointed by *sec\_handle* with first iteration of HMAC (digest of key xor ipad). The key is taken from *key\_ptr* in CMEM. Key may be of any *size* up to native block size of algorithm.

#### 4.11.2.1.2 Data Processing

Digesting data is accomplished by the following operation that operates on a data of up to 128B. If data is longer than 128B, the DP application should loop over all data.

```

void ezdp_mac_calculation(ezdp_security_handle_t sec_handle, void __cmem *data_ptr, const
    bool init, const bool last, uint32_t size)
  
```

*ezdp\_mac\_calculation()* digests the data pointed by *data\_ptr* in CMEM and updates the session state in context memory. *sec\_handle* indicates the algorithm to use and points to the context memory of the

session. *size* of data to digest and can be up to 128B.

*init* flag indicates that the data to digest is the first in the current session and initial MAC is created.

Notice that if HMAC mode is used then the *init* flag should be false for all calls to

*ezdp\_mac\_calculation()* in the session since the call to *ezdp\_start\_hmac\_calculation()* at initialization stage already created initial MAC in context.

*last* flag indicates that the data to digest is the last in the current session and the relevant security engine will pad the data to a multiple of algorithm native block size (DP application does not need to pad).



The *init* and *last* flags may both true to indicate that the provided data is first and last, i.e. the only data to digest.

#### 4.11.2.1.3 Finalization

Much like the initialization phase, the finalization phase of a digest task is needed only for HMAC mode.

In HMAC mode, key xor o\_pad contains the calculated MAC should be hashed. This is done using the following API:

***void ezdp\_end\_hmac\_calculation(ezdp\_security\_handle\_t sec\_handle, void \_\_cmem \*key\_ptr, uint32\_t size)***

*ezdp\_end\_hmac\_calculation()* finalizes the MAC in context memory which is pointed by *sec\_handle* with last iteration of HMAC. The key is taken from *key\_ptr* in CMEM. Key may be of any *size* up to native block size of algorithm.

After finalization the MAC may be read from context memory with the following operation.

***void ezdp\_read\_security\_mac(ezdp\_security\_handle\_t sec\_handle, void \_\_cmem \*mac\_ptr, uint32\_t mac\_size)***

*ezdp\_read\_security\_mac()* reads the MAC from context memory which is pointed by *sec\_handle* with key into *mac\_ptr* on CMEM.

#### 4.11.2.1.4 MD5 Calculation Example

The following example describes a simple MAC calculation over a buffer in CMEM using the MD5 algorithm. The buffer is provided as a data pointer. The context ID allocated for the task must be provided. The resulting MAC will be read into *result\_mac* pointer in CMEM.

```
void calcMD5digest(uint8_t __cmem *data,
                  uint32_t data_size,
                  uint8_t context_id,
                  uint8_t __cmem *result_mac,
                  uint32_t mac_size)
{
    /* Initialize a security handle */
    struct ezdp_security_handle md5_handle;
    md5_handle.context_id = context_id;
    md5_handle.alg_type = EZDP_MD5_ALG;

    bool first = true;

    /* Digest data in 128B chunks, last chunk is padded by accelerator */
    while(data_size > EZDP_SEC_MAX_BYTES)
    {
        ezdp_mac_calculation(md5_handle.raw_data,
                            data,
                            first,
                            false,
                            EZDP_SEC_MAX_BYTES);
        data_size -= EZDP_SEC_MAX_BYTES;
    }
}
```

```

        data += EZDP_SEC_MAX_BYTES;
        first = false;
    }
    ezdp_mac_calculation(md5_handle.raw_data,
                        data,
                        first,
                        true,
                        data_size);

    /* Read result MAC from context memory to CMEM */
    ezdp_read_security_mac(md5_handle.raw_data,
                          result_mac,
                          mac_size));
}

```

#### 4.11.2.1.5 HMAC-SHA1 Calculation Example

The following example describes a simple MAC calculation over a buffer in CMEM using the SHA1 algorithm with HMAC mode. The buffer is provided as a data pointer. The key for HMAC is provided in *hmac\_key* pointer. The context ID allocated for the task must be provided. The resulting MAC will be read into *result\_mac* pointer in CMEM.

```

void calcSHAdigest(uint8_t __cmem *data,
                  uint32_t data_size,
                  uint8_t __cmem *hmac_key,
                  uint32_t hmac_key_size,
                  uint8_t context_id,
                  uint8_t __cmem *result_mac,
                  uint32_t mac_size)
{
    /* Initialize a security handle */
    struct ezdp_security_handle sha_handle;
    sha_handle.context_id = context_id;
    sha_handle.alg_type = EZDP_SHA_ALG;

    /* Initialize context according to HMAC specifications */
    ezdp_start_hmac_calculation(sha_handle.raw_data,
                               hmac_key,
                               hmac_key_size);

    /* Digest data in 128B chunks, last chunk is padded by accelerator */
    while(data_size > EZDP_SEC_MAX_BYTES)
    {
        ezdp_mac_calculation(sha_handle.raw_data,
                            data_ptr,
                            false,
                            false,
                            EZDP_SEC_MAX_BYTES);
        data_size -= EZDP_SEC_MAX_BYTES;
        data += EZDP_SEC_MAX_BYTES;
    }
    ezdp_mac_calculation(sha_handle.raw_data,
                        data,
                        first,
                        true,
                        data_size);

    /* Finalize MAC according to HMAC specifications */
    ezdp_end_hmac_calculation(sha_handle.raw_data,
                             hmac_key,
                             hmac_key_size);

    /* Read result MAC from context memory to CMEM */
    ezdp_read_security_mac(sha_handle.raw_data,
                          result_mac,

```

```

        mac_size);
    }

```

### 4.11.2.2 Encryption/Decryption

#### 4.11.2.2.1 Initialization

The initialization phase of 3DES or AES based algorithms (encryption or decryption) includes writing the key and initial vector (in modes where needed) to context memory. This is done with the following APIs:

```
void ezdp_write_security_key(ezdp_security_handle_t sec_handle, void __cmem *key_ptr,
uint32_t key_size)
```

```
void ezdp_write_security_initial_vector(ezdp_security_handle_t sec_handle, void __cmem
*iv_ptr, uint32_t iv_size)
```

*ezdp\_write\_security\_key()* and *ezdp\_write\_security\_initial\_vector()* initialize context memory which is pointed by *sec\_handle* with the key and initial vector from CMEM which are pointed by *key\_ptr* and *iv\_ptr* respectively.

In addition, AES based decryption algorithms require key expansion. This should be done using the following API after writing the key to the context:

```
void ezdp_expand_security_key (ezdp_security_handle_t sec_handle)
```

#### 4.11.2.2.2 Processing Data

Encryption and decryption are accomplished through the following two APIs that operate on data that is a multiple of the algorithm native block size and is of up to 128B in size. If data is longer than 128B, the DP application should loop over all data. Padding data that is not a multiple of block size is also the DP application's responsibility.

```
void ezdp_encrypt(ezdp_security_handle_t sec_handle, void __cmem *plain_data_ptr, void
__cmem *encr_data_ptr, uint32_t size)
```

*ezdp\_encrypt()* encrypts the data pointed by *plain\_data\_ptr* in CMEM and writes the result to *encr\_data\_ptr* in CMEM (both may point to same CMEM offset). *sec\_handle* indicates the algorithm to use and points to the context memory of the session. *size* of data to encrypt must be a multiple of the algorithm native block size and not larger than 128B.

```
void ezdp_decrypt(ezdp_security_handle_t sec_handle, void __cmem *encr_data_ptr, void __cmem
*plain_data_ptr, uint32_t size)
```

*ezdp\_decrypt()* decrypts the data pointed by *encr\_data\_ptr* in CMEM and writes the result to *plain\_data\_ptr* in CMEM (both may point to same CMEM offset). *sec\_handle* indicates the algorithm to use and points to the context memory of the session. *size* of data to decrypt must be a multiple of the algorithm native block size and not larger than 128B.

#### 4.11.2.2.3 Finalization

Finalization phase is not required for encryption/decryption tasks.

#### 4.11.2.2.4 3DES-CBC Encryption Example

The following example describes a simple in-place encryption over a buffer in CMEM using the 3DES-CBC algorithm. The plain text is provided as a data pointer in CMEM. The encryption key is provided in *encr\_key* pointer (192 bits). The initial vector is provided in *iv* pointer. The context ID allocated for the task must be provided. The encrypted cipher text will be placed in data pointer, overriding the original plain text data. This function assumes the size of the provided data is a multiple of 3DES block size (8B).

```

void encrypt3DES(uint8_t __cmem *data,
                uint32_t data_size,
                uint8_t __cmem *encr_key,
                uint32_t encr_key_size,
                uint8_t __cmem *iv,
                uint32_t iv_size,
                uint8_t context_id)
{
    /* Initialize a security handle */
    struct ezdp_security_handle tdes_handle;
    tdes_handle.context_id = context_id;
    tdes_handle.alg_type = EZDP_3DES3_CBC_ALG;

    /* Write key to context */
    ezdp_write_security_key(tdes_handle.raw_data,
                          key,
                          key_size);

    /* Write initial vector to context */
    ezdp_write_security_initial_vector(tdes_handle.raw_data,
                                      iv,
                                      iv_size);

    /* Encrypt (in-place) data in 128B chunks */
    while(data_size > EZDP_SEC_MAX_BYTES)
    {
        ezdp_encrypt(3des_handle.raw_data,
                    data,
                    data,
                    EZDP_SEC_MAX_BYTES);
        data_size -= EZDP_SEC_MAX_BYTES;
        data += EZDP_SEC_MAX_BYTES;
    }

    /* Assuming residual data is a multiple of block size */
    ezdp_encrypt(3des_handle.raw_data,
                data,
                data,
                data_size);
}

```

#### 4.11.2.2.5 AES-CBC-256 Encryption Example

The following example describes a simple in-place decryption over a buffer in CMEM using AES-CBC algorithm with a 256-bit key. The plain text is provided as data pointer in CMEM. The encryption key is provided in *encr\_key* pointer. The initial vector is provided in *iv* pointer. The context ID allocated for the task must be provided. The encrypted cipher text will be placed in data pointer, overriding the original plain text data. This function assumes the size of the provided data is a multiple of AES block size (16B).

```

void decryptAES_CBC_256(uint8_t __cmem *data,
                       uint32_t data_size,
                       uint8_t __cmem *encr_key,
                       uint32_t encr_key_size,
                       uint8_t __cmem *iv,
                       uint32_t iv_size,
                       uint8_t context_id)
{
    /* Initialize a security handle */
    struct ezdp_security_handle aes_handle;
    tdes_handle.context_id = context_id;
    tdes_handle.alg_type = EZDP_AES_CBC_256_ALG;

    /* Write key to context */
    ezdp_write_security_key(aes_handle.raw_data,
                          key,

```

```

        key_size);

    /* Write initial vector to context */
    ezdp_write_security_initial_vector(aes_handle.raw_data,
                                      iv,
                                      iv_size);

    /* Expand key */
    ezdp_expand_security_key(aes_handle.raw_data);

    /* Decrypt (in-place) data in 128B chunks */
    while(data_size > EZDP_SEC_MAX_BYTES)
    {
        ezdp_decrypt(aes_handle.raw_data,
                    data,
                    data,
                    EZDP_SEC_MAX_BYTES);
        data_size -= EZDP_SEC_MAX_BYTES;
        data += EZDP_SEC_MAX_BYTES;
    }

    /* Assuming residual data is a multiple of block size */
    ezdp_encrypt(aes_handle.raw_data,
                data,
                data,
                data_size);
}

```

### 4.11.2.3 GCM

#### 4.11.2.3.1 Initialization

In the initialization phase, the security key is copied to context memory with *ezdp\_write\_security\_key()* just as described above ([4.11.2.2.1](#), initialization of encryption/decryption task).

The initial vector, however, is handled differently. If the initial vector size is 96 bits then it needs to be padded with 31b'0 || 1 (31 bits of zeros and a bit of 1) before writing it to context memory using *ezdp\_write\_security\_initial\_vector()* as described above (section [4.11.2.2.1](#), initialization of encryption/decryption task).

However, if the initial vector size is not 96 bits then it needs to be padded as follows:

1. Pad IV with zeroes to a size that is a multiple of 128 bits.
2. Pad with 64 bits of zeros.
3. Pad with size of IV in bits represented in 64 bits.

The result is fed in chunks of up to 128B (and multiples of 16B) to the following operation that will initialize context state with the initial vector (J0) according to GCM specifications:

```
void ezdp_generate_security_initial_vector(ezdp_security_handle_t sec_handle, void __cmem
*data_ptr, const bool init, uint32_t size)
```

*ezdp\_generate\_security\_initial\_vector()* initializes the initial vector in context memory which is pointed by *sec\_handle* according to GCM specifications (AES encrypt 128b'0 with the key in context and use the result as key for GHASH digest of the provided *data*). *init* flag indicates that the provided *data* is the first to digest.

In addition, the additional authentication data (AAD) needs to be digested with *ezdp\_mac\_calculation()* API as described in digest section above (section [4.11.2.1.2](#), processing data of digest task).



#### 4.11.2.3.2 Processing Data

The processing phase is accomplished in the same way described in the encryption/decryption section above (section [4.11.2.2.2](#)) with two important differences. The first is that the MAC in context memory is updated after each call to *ezdp\_encrypt()* or *ezdp\_decrypt()*. The second difference is that padding of the data to a multiple of block size is not needed.

#### 4.11.2.3.3 Finalization

In the finalization phase, the initial vector needs to be regenerated in context memory as described [above](#), in initialization phase.

Then the DP application is required to prepare 128 bits in CMEM that describe the length of the digested data as follows. The first 64 bits describe the length of the additional authentication data and the last 64 bits describe the length of the encrypted data. These 128 bits are fed into the following operation:

***void ezdp\_end\_gcm\_calculation(ezdp\_security\_handle\_t sec\_handle, void \_\_cmem \*data\_ptr, uint32\_t size)***

*ezdp\_end\_gcm\_calculation()* finalize the MAC in context memory which is pointed by *sec\_handle* according to GCM specifications (GHASH the provided data and encrypt MAC). The data is pointed by *data\_ptr*.

After that, the MAC may be read with *ezdp\_read\_security\_mac()* as described above (section [4.11.2.1.3](#), finalization of digest task).

#### 4.11.2.3.4 AES-GCM-128 Encryption and Digest Example

The following example describes a simple MAC calculation with in-place encryption over a buffer in CMEM using the AES-GCM algorithm with a 128-bit key, according to the following scheme:

The additional authentication data is provided in *aad* pointer in CMEM. The plain text is provided as *data* pointer in CMEM. The encryption key is provided in *encr\_key* pointer. The initial vector is provided in *iv* pointer. The context ID allocated for the task must be provided. The encrypted cipher text will be placed in *data* pointer, overriding the original plain text data and the resulting MAC will be written to *result\_mac* pointer in CMEM.

A 16B work area is required in CMEM.

```
void encryptGCMdigest(uint8_t __cmem *aad,
                     uint32_t aad_size,
                     uint8_t __cmem *data,
                     uint32_t data_size,
                     uint8_t __cmem *encr_key,
                     uint32_t encr_key_size,
                     uint8_t __cmem *iv,
                     uint32_t iv_size,
                     uint8_t context_id,
                     uint8_t __cmem *result_mac,
                     uint32_t mac_size,
                     uint8_t __cmem *work_area)
{
    /* Initialize a security handle */
    struct ezdp_security_handle gcm_handle;
    gcm_handle.context_id = context_id;
    gcm_handle.alg_type = EZDP_AES_GCM_128_ALG;

    /* Initialization phase */
    gcm_initialize(sdata,
                  encr_key,
                  encr_key_size,
                  iv,
                  iv_size,
```

```

        aad,
        aad_size,
        work_area);

    /* Processing data phase */
    gcm_process(gcm_handle.raw_data,
        data,
        data_size);

    /* Finalization phase */
    gcm_finalize(gcm_handle.raw_data,
        iv,
        iv_size,
        data_size,
        aad_size,
        work_area);

    /* Read result MAC from context memory to CMEM */
    ezdp_read_security_mac(gcm_handle.raw_data,
        result_mac,
        mac_size);
}

void gcm_initialize(ezdp_security_handle_t gcm_handle,
    uint8_t __cmem *encr_key,
    uint32_t encr_key_size,
    uint8_t __cmem *iv,
    uint32_t iv_size,
    uint8_t __cmem *aad,
    uint32_t aad_size,
    uint8_t __cmem *work_area)
{
    /* Copy key to context memory */
    ezdp_write_security_key(gcm_handle,
        encr_key,
        encr_key_size);

    /* Generate J0 and write it as initial vector to security context */
    gcm_generate_j0(iv, work_area);

    /* Assume J0 is 16 bytes */
    ezdp_write_security_initial_vector(gcm_handle,
        work_area,
        16);

    /* Digest additional authentication data (assuming one chunk) */
    ezdp_mac_calculation(gcm_handle,
        aad,
        true,
        false,
        aad_size);
}

void gcm_process(ezdp_security_handle_t gcm_handle,
    uint8_t __cmem *data,
    uint32_t data_size)
{
    /* Encrypt (in-place) data in 128B chunks */
    while(data_size > EZDP_SEC_MAX_BYTES)
    {
        ezdp_encrypt(gcm_handle.raw_data,
            data,
            data,
            EZDP_SEC_MAX_BYTES);
        data_size -= EZDP_SEC_MAX_BYTES;
        data += EZDP_SEC_MAX_BYTES;
    }
    ezdp_encrypt(gcm_handle.raw_data,

```

```

        data,
        data,
        data_size);
}

void gcm_finalize(ezdp_security_handle_t gcm_handle,
                uint8_t __cmem *iv,
                uint32_t iv_size,
                uint32_t data_size,
                uint32_t aad_size,
                uint8_t __cmem *work_area)
{
    /* Generate J0 and write it as initial vector to security context */
    gcm_generate_j0(in, work_area);

    /* Assume J0 is 16 bytes */
    ezdp_write_security_initial_vector(gcm_handle,
                                      work_area,
                                      16);

    /* Finalize GCM MAC calculation. */
    uint64_t *gcm_fin = (uint64_t*)work_area;
    gcm_fin[0] = aad_size*8; // size of AAD in bits
    gcm_fin[1] = data_size*8; // size of data in bits
    ezdp_end_gcm_mac_calculation(gcm_handle.raw_data,
                                gcm_fin,
                                2*sizeof(uint64_t));
}

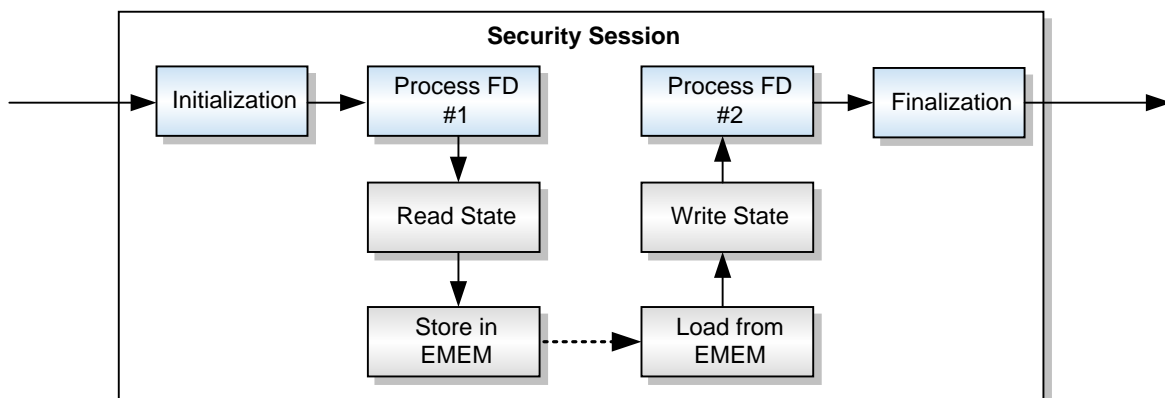
void gcm_generate_j0(uint8_t *iv, uint32_t *j0)
{
    /* Assuming iv_size of 96 bits (12B) */
    ezdp_memcpy(j0, iv, 12);
    /* Pad IV and copy to context memory */
    j0[3] = 0x00000001;
}

```

#### 4.11.2.4 Security Context State

Some DP applications might need to store and restore the session state so that execution can be resumed from the same point at a later time.

For example, when a DP application needs to perform a security operation over data that spans across FDs it is not recommended to keep the context ID until the next FD will be received. Instead, the session state should be stored (perhaps in EMEM) and restored when the next FD is received.



The following APIs allow such a load and store of a session state from context memory:

```
void ezdp_read_security_state(ezdp_security_handle_t sec_handle, void __cmem *sec_state_ptr,
    uint32_t sec_state_size)

void ezdp_write_security_state(ezdp_security_handle_t sec_handle, void __cmem
    *sec_state_ptr, uint32_t sec_state_size)
```

`ezdp_read/write_security_state()` will load and store a security session state from/to the context memory pointed by `sec_handle`. The state is loaded/stored to/from CMEM at `sec_state_ptr`.



The size of the state is algorithm dependent and defined in `ezdp_sec_state_size` enum, for more information refer to the `ezdp_security_defs.h` file.

## 4.12 FCU – Budgeting - TBD

## 4.13 IPC – TBD

## 4.14 PCI

### 4.14.1 NPS PCI Overview

The NPS-400 integrates two PCI Express end-point controllers – west side and east side (E0 and E1). Each controller supports four Physical Functions (PFs) with 128 Virtual Functions (VFs).

One of the PFs (PF0) on the west side controller is hardcoded for NPS SoC configuration. EZchip provides an NPS driver for NPS configuration. E0 PF0 functionality is hardcoded and cannot be changed by the NPS developer. For details refer to the *EZcp Reference Manual*.

The rest of the Physical Functions (E0 PF1...PF3 and E1 PF0...PF3) can be used for implementation of PCI functionality as the customer requires.

### 4.14.2 PCI Related Terminology

In this section of the document the following terms are used:

Host:	The computer connected to the NPS over PCI. Host is the master on this PCI connection. NPS is one of multiple PCI devices connected to the host.
PNI:	PCI Network Interface. EZchip implemented interface (API) used for data exchange between the host and NPS.
PNI protocol:	EZchip issued protocol that lies at the root of PNI.
PNI backend:	Code running on NPS intended to support PNI protocol.
PNI frontend:	Code running on the host intended to support PNI protocol.
DPDK:	Data Plane Development Kit. DPDK is a set of host based data plane libraries and network interface controller drivers for fast packet processing. EZchip intends NPS to be one of the platforms supported by DPDK.
BAR:	Base Address Registers

### 4.14.3 NPS PCI Programming Approaches

Several approaches may be used for the NPS PCI programming. The approaches are listed below with the top one being the most recommended and simpler to use and the bottom one providing the most control but also requiring the most *knowledge* of the low level implementation. Customers can implement any PCI functionality according to their needs.

EZchip provides a ready-to-use highly efficient solution based on PNI protocol. This solution can be used for network packet exchange between the NPS and the host over PCI.

#### PNI Application Approach:

The application approach is the recommended approach for most use cases. In it, the PNI backend runs on dedicated threads. The DP application communicates (i.e. receive and redirect) using dedicated PMU queues. The DP application in this approach does not need to know the details of the PNI backend and is not responsible for its *proper* performance. Overall that trades some core utilization for easier overall architecture and a much shorter development cycle.

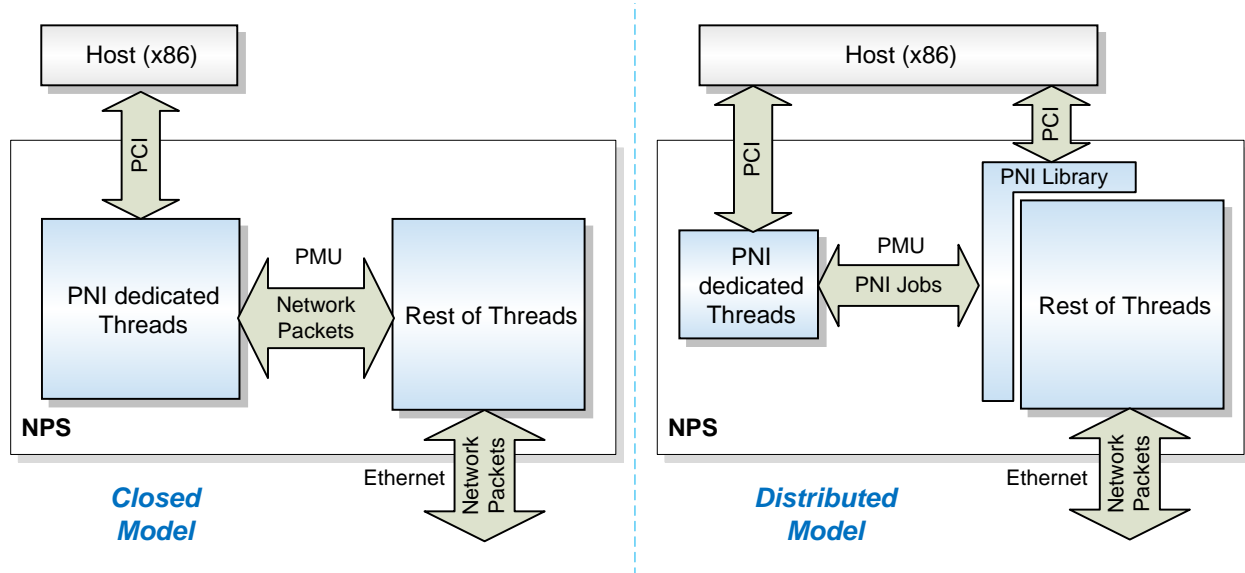
We term such a design a “closed model” because the PNI backend runs independently and does not intersect with the customer’s application.

#### PNI Library Approach:

The library approach is similar to the application approach described above except that in it the application is now responsible for the proper resource allocation and handling of the PNI backend. This promises more efficient PNI resource management at the cost of extra complexity and the need to really understand the PNI backend.

We term such a design a “distributed model” because the PNI backend code is mixed with the user DP application.

**Figure 20. PNI Application Approach (“closed model”) vs. PNI Library Approach (“distributed model”)**



#### Intermediate Level Approach:

EZchip provides an intermediate level API for PCI programming as a part of Data Path library. (See “PCI Interface Operations” chapter in *EZdp Reference Manual*.) This library wraps the low level API with clear functions. This way the developer can have ready to use functions for DMA access to host memory, BARs implementation, sending interrupts to host, etc. You can implement your own PCI related algorithm using this library.

For better understanding of this API, refer to the *EZdp Reference Manual* and analyze `ezdp_pci.h` file. Also you can find simple examples of using this API as a part of the NPS software development kit.

Customers should go this way only if there is a need to implement unique functionality that is not supported by the PNI protocol.

#### Low Level Approach:

The NPS software developer can implement PCI functionality on a low level, using direct hardware access. This approach gives complete control, but it is the hardest to implement and is not recommended.

Refer to the “PCI Express Controller” section in the *NPS-400 Architectural Specifications* for details on the low level access.

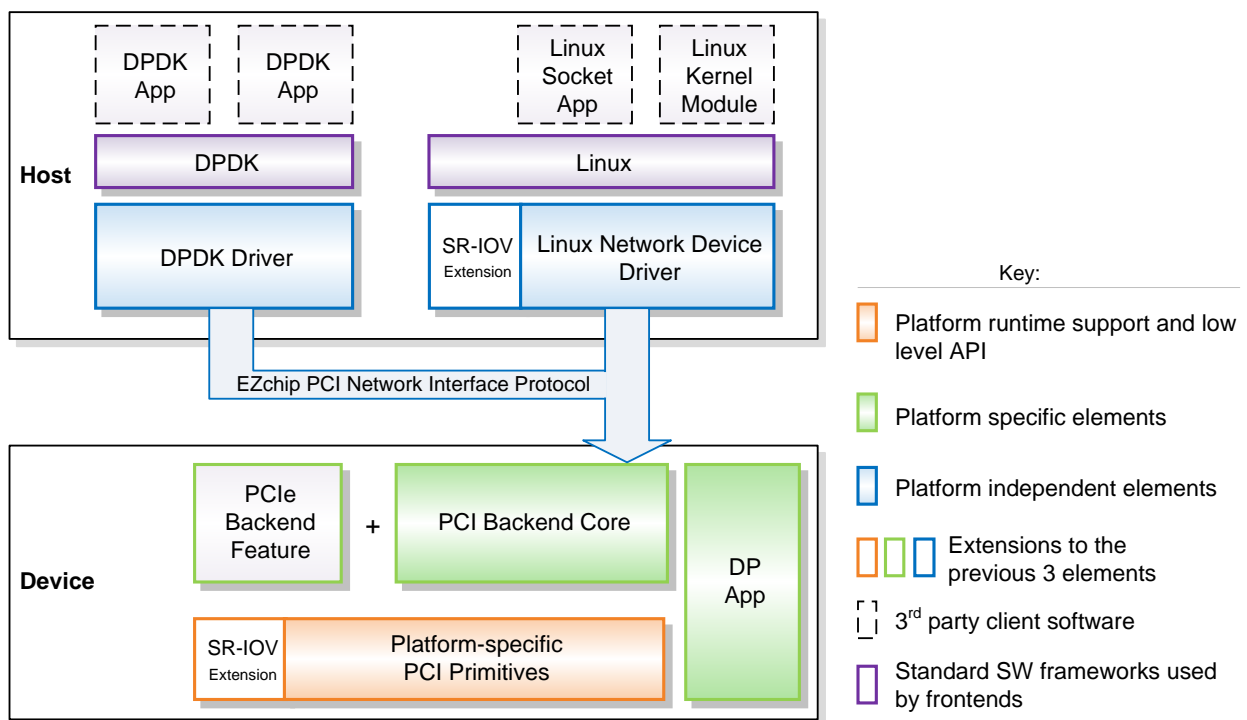
We expect taht customers will never go with this approach.

### 4.14.4 High level SW Architecture

The following diagram depicts the software system architecture as relevant to the PNI protocol based solution. It is important to note that the customer receives a package, including PNI backend, demo DP application and two independent solutions for PNI frontend (Linux Network Device driver) and DPDK driver.

Since the PNI protocol is a standard for EZchip PCI Network Interface, the same host drivers will work with other modern EZchip processors, such as the TILE-Mx.

**Figure 21. NPS software system architecture as relevant to PNI solution**



### 4.14.5 PNI Application Overview

EZchip provides a PNI demo package. This package includes a host PCI driver, which presents NPS as a standard Ethernet controller. For the NPS side, this package provides a PNI library for support of the host driver, the PNI application implements a closed model and a demo application, implements NIC functionality. The application is compatible with DPDK.

Using this package, you can build an application that will send/receive network packets to/from PCI, send notifications to and receive commands from the host. This package saves the developer from implementing PCI related features, and allows focusing on data processing only, according to product demands.

The host drivers support standard host software (Linux kernel, DPDK). The host driver and NPS PNI library provide the highest possible performance for data processing. They support Single Root Input-Output Virtualization (SR-IOV) as well. It is highly recommended that 3<sup>rd</sup> party host applications use EZchip's host driver rather than developing their own.

### 4.14.6 PNI Application Integration

EZchip recommends that our customers run the provided demo in order to familiarize themselves with the PCI implementation. This demo shows how to turn NPS into a regular NIC. Linux 'sees' the NPS as an Ethernet device and can send/receive packets over it.

The next step is to understand the configuration process. Configuration here means the resources allocated for the application. All resources can be divided into "internal" and "external" parts. Internal resources are used inside the PNI backend. Customers should be familiar with them in order to prevent interference from other applications. These (i.e. the internal configuration) include CPU clusters, IMEM blocks, EMEM area, PMU queues and timers. External resources include PMU queues used for data exchange with PNI application.

After this, customers can design and implement their applications. The closed model should be used by default and programmers should consider alternatives only if this closed mode leads to a lack of resources.

### 4.14.7 Multiple PCI Device Implementations

The PNI demo application is based on one endpoint E0, one physical function PF1 and one virtual function VF0.

In addition, customers can add their own functionality using free Physical Functions. This functionality can be written using the intermediate level approach. This way, the customer's code can be completely independent from the PNI application with a single exception. If this functionality uses the IO BAR, it should share the PCI Message Queue with the PNI application.

## 4.15 Lock Mechanism – TBD

## 4.16 Data Structures – TBD

### 4.16.1 Queue – TBD

### 4.16.2 List – TBD

## 4.17 Advanced Statistic Counter Features

### 4.17.1 On-demand Statistic Counters

In addition to regular statistic counters covered in the [Counter Acceleration](#) section, the on-demand (OD) statistic block can provide acceleration for additional services that require coherent access (atomic read-modify-write) to memory by multiple tasks simultaneously without blocking the data-plane code.

On-demand counters may reside either in EMEM (up to 512M counters) or in a dedicated internal SRAM (up to 8K counters). The OD block makes use of the layer 2 cache for optimizing performance when the same counters are used occasionally. There is also a programmatic method for loading counters to the cache when the program expects to be using a counter later on (e.g. *ezdp\_prefetch\_single\_ctr* (*ezdp\_sum\_addr\_t addr*)).

#### 4.17.1.1 Bitwise Counters

Bitwise counters utilize 64 bits in each counter line, providing atomic access to groups of 1, 2, 4, 8 or 16 bits at a time.

##### 4.17.1.1.1 Implementing semaphores

In their simplest form, bitwise counters may be used for implementing semaphores, where each bit is a separate semaphore that can be “grabbed” or “released”. Grabbing a semaphore is implemented using the *ezdp\_read\_and\_set\_bits\_bitwise\_ctr()* API on the corresponding bit, setting it to “1” and looping until the returned value is “0”. Releasing a semaphore is implemented using the *ezdp\_clear\_bits\_bitwise\_ctr()* API by clearing the corresponding bit back to “0”.



In order to implement a counting semaphore, a regular (long) counter can be used with the *ezdp\_read\_and\_cond\_dec\_single\_ctr()* API.

##### 4.17.1.1.2 Implementing a state-machine

A state machine can be implemented using the *ezdp\_read\_and\_cond\_write\_bits\_bitwise\_ctr()* API on bitwise counters in groups of 1, 2 or 4 bits to implement a machine with up to 16 states. The previous state should be provided in the *cmp\_value* operand, and the new state in the *value* operand. The *success\_ind* returned bit should be checked to verify successful state transfer.

In case where more than 16 states are required, another possible method of implementing a state-machine could be use of an atomic operation, not via the statistics block, such as the *ezdp\_atomic\_cmpxchg32\_sum\_addr()* API.

#### 4.17.1.2 Counters Message Queue

The on-demand message queue provides a convenient method for the data plane to receive messages with the counter value while resetting the counters. The software can then avoid having to sequentially scan long ranges of counters, a tedious task, especially if the counters are sporadically updated. There are eight message queues, one per each on-demand engine. A message can be generated in one of two cases:

- Counter increment caused it to exceed the per-counter configured threshold, and message generation is enabled on the counter.
- A periodic scan machine has visited the counter, and the counter value exceeded the machine’s configured threshold (either main threshold in a main scan, or intermediate threshold in an intermediate scan).



The above two cases also apply to double counters, in which case a separate threshold can be configured for the byte count and for the event count. All thresholds are specified as the threshold bit to cross, practically allowing values which are powers of two.

If message queues are to be read from the data plane, a message queue descriptor needs to be initialized in a similar manner to initializing a search structure descriptor, using the `ezdp_init_ctr_msg_queue_desc()` API.

If messages are expected to be generated at a high rate, a dedicated thread could be created for monitoring and reading the message queues. In this case, the queue descriptor can be initialized only on the CTOP running this task. For lower rates, message handling can be triggered by a periodic PMU timer event, in order to allow better utilization of the CTOP engines. In this case, all CTOPs of the application need to have the queue descriptor initialized.

Reading messages from the message queue is performed with the `ezdp_read_ctr_msg()` API, which returns `struct ezdp_ctr_msg`, containing:

- Message Type (`ezdp_ctr_msg_type`): Periodic, Counter Threshold, Null
- Counter Type (`ezdp_ctr_type`): Single, Dual
- Counter Address (`ezdp_sum_addr`)
- Counter value

On sending a message, the counter itself is reset and the application may need to store the logical aggregated value, or pass it on to the control plane.



Messages are distributed to the eight message queues according to the on-demand engine which manages the specific counter, regardless of the application setting. In case a per-application message queue is required, then the OD engines need to be allocated per application.

Each of the eight OD engines has 16 general purpose machines which can be configured to operate as a scan machine. Each periodic scan machine can be configured with a high threshold that is used in the main cycles, a low threshold that is used in the intermediate cycles, and with the number of intermediate cycles between each two main cycles.

#### 4.17.1.3 Shadow Counters with Synchronized Swapping

A group containing long and/or double counters can be configured to have a shadow group as well. In practice, these are two groups that can each in turn be mapped to the same address range seen by the data plane. The control plane can swap the active set of counters that the data plane is using, so that it can read and reset the idle set without interference from the data plane. The swapping is completely transparent to the data plane processing code.



Swapping between the active and shadow groups by the HW is synchronized in time, and not necessarily performed on a frame boundary. If the frame processing code updates several counters sequentially and it is required for all counters to be of the same set, then the swapping should be implemented by other means in SW.

#### 4.17.1.4 Token Bucket Counters

Token buckets in their simplest form serve as traffic rate limiters. Each TB can be configured with a committed information rate (CIR), which is the rate in which the bucket is filled with credits, and committed burst size (CBS), which is the size of the bucket. The code of the frame being processed would perform the `ezdp_read_tb_ctr()` operation on the corresponding TB counter, specifying the frame length as *value*. The returned result would indicate the TB bucket status and color, where normally, for a red result the frame should be discarded (assuming credits have not been consumed from the bucket), while for a green result the frame is forwarded.

Dual bucket TBs (three color marker) also have a peak bucket which can be configured with its own rate and size. In case the commit bucket is empty, but the peak bucket has credits, a “yellow” result is returned, in which case the data plane SW may mark the frame or give it higher precedence to be dropped.

The software may adjust the frame length given as *value* to `ezdp_read_tb_ctr()` in order to compensate for the CRC that has been stripped, external header that will be added/removed, packet encapsulation, etc. A constant value can be used instead in order to limit packets per second, rather than bytes per second.

A TB counter may be configured to be “color-aware”. In this case the “pre\_color” operand should also be provided to the `ezdp_read_tb_ctr()` API, and may have an affect over which buckets’ credits will be consumed from and on the returned color.

The `ezdp_check_tb_ctr()` API may be used in order to read the TB state without updating it. The `ezdp_inc_tb_ctr()` API can be used to force updating of a TB regardless of the bucket states.

In order for token buckets to operate correctly, an OD general purpose machine must be configured to operate as a refresh machine (*bTokenBucket* set to True in `EZapiStat_GPMachineParams`) and scan the TB range.

#### 4.17.1.5 Hierarchical Token Bucket Counters

TBD

#### 4.17.1.6 Watchdog Counters

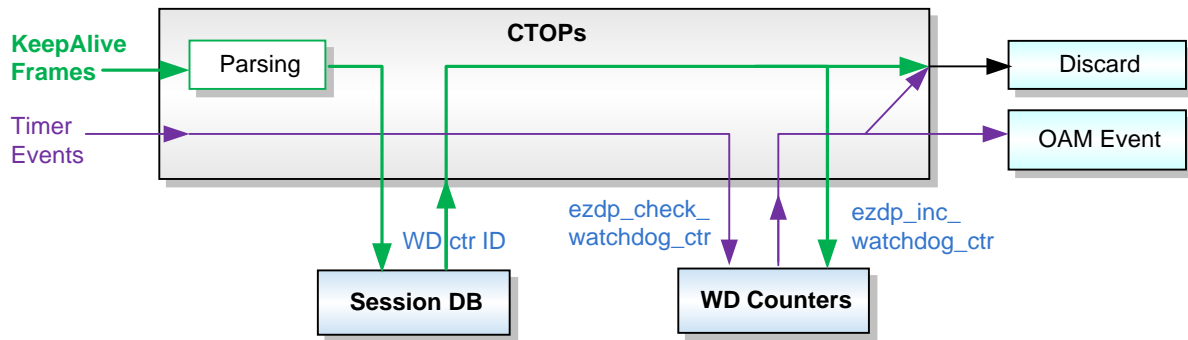
Watchdog counters are used to monitor rates of events, such as the arrival of incoming keep-alive OAM frames indicating that a link is in order. When the monitored events do not occur at a satisfactory rate, a watchdog event is generated that can trigger, for example a switch-over event.

A watchdog (WD) counter can be configured with a profile that indicates when an event should be triggered. For example, a profile may contain the number of events that should be expected in each window and number of sequential non-conforming windows to generate an event.

In a standard OAM implementation, keep-alive frames are parsed by the data-plane code. If they are classified to be destined to a local end point, the WD counter index is obtained from a session DB lookup table, and the `ezdp_inc_watchdog_ctr()` operation is performed on the WD counter.

The WD counter is advanced using the `ezdp_check_watchdog_ctr()` operation, which also receives indication from the WD counter as to whether an OAM event should be triggered. Typically, this will be performed from a thread triggered by a PMU timer event, set according to the required WD window interval.

### Figure 22. Watchdog counter operation



struct `ezdp_watchdog_ctr_check_result` contains the result indicating if an alert should be triggered, and the alert type: Min Alert, in case there were insufficient events, or Max Alert in case there were too many events.

Both `ezdp_inc_watchdog_ctr()` and `ezdp_check_watchdog_ctr()` operations can run only on WD counters of live sessions that have been started. Starting a WD counter is performed with the `ezdp_start_watchdog_ctr()` API. Once an alert has been detected, the WD counter stops and needs to be restarted by SW for continued monitoring.

Checking the status of a WD counter can be done with `ezdp_read_watchdog_ctr_cfg()` API. The returned valid bit indicates whether the counter has been started. The SW must be aware of the WD sub type (accumulative window, sliding window) or detect it, in order to know the appropriate location of the valid bit (`ezdp_watchdog_ctr_cfg.accumulative_window.valid` or `ezdp_watchdog_ctr_cfg.sliding_window.valid`).

### 4.17.2 Posted Statistic Counters

For basic operation with posted counters, see the [Posted Counters](#) section.

#### 4.17.2.1 Shadow Counters with Synchronized Swapping

A group of posted counters can be configured to have a shadow group, in a similar manner to the on-demand shadow groups. Swapping between the two sets is performed by the control plane and is transparent to the data plane which can continue operating on the counters in a normal fashion. Updates to two counters by a dual counter update command are guaranteed to be performed on the same set.

#### 4.17.2.2 *Reporting and counters message queue*

Posted counters are optimized to update operations only, yet reading of a posted counter is allowed using reporting and a special message queue in which the counter value is written.

`ezdp_report_posted_ctr` and `ezdp_dual_report_posted_ctr` APIs are used to generate a message with the counter value to a message queue.

► *Note a dual counter report will generate two counter messages.*

In regular way (like read the counter value) but using report and reading the counter value from message queue.

ezdp\_report\_posted\_ctr  
ezdp\_dual\_report\_posted\_ctr

## Additional

ezdp\_report and clear\_posted\_ctr

```
ezdp_dual_report_and_clear_posted_ctr  
ezdp_reset_posted_ctr  
ezdp_dual_reset_posted_ctr  
  
ezdp_read_posted_ctr_msg  
ezdp_init_posted_ctr_msg_queue_desc
```

APIs support resetting, incrementing, decrementing, setting and reading each of the counters.

## 4.18 Controlling Hardware Thread Scheduling

As we have previously described in the [Multi-threading](#) section, the CTOP core supports a mechanism of hardware multi-threading that is used to hide memory access and accelerator processing latencies providing high execution unit utilization.

At the most basic level, the CTOP multi-threading feature is managed automatically by a combination of hardware mechanisms and SDK implementation and the programmer need not be concerned with its operation.

An advanced programmer however, may choose to exercise fine control of the CTOP thread scheduling as a way to lower frame processing latencies by parallelization of several independent operations.

### 4.18.1 Asynchronous Operations

All the API functions described thus far work in a synchronous manner; a function call is made and when the function returns all the outputs are already available.

Many of these APIs utilize hardware acceleration engines outside the core, where the underlying programming model hidden by the high level API is asynchronous in nature and compromised of two stages:

- **Invocation** – the function sets up parameters needed for performing the operation in the format expected by the specific hardware engine and the hardware engine is invoked via a specialized CTOP instruction.
- **Synchronization** – the function then requests the CTOP core Multi-Threading Management (MTM) unit to context switch the current hardware thread off of the execution unit. When the hardware thread is next scheduled into the execution unit, we are guaranteed that the requested operation has completed.

In order to expose this lower level asynchronous programming model, these API calls have asynchronous versions, in addition to the synchronous versions discussed so far.

The asynchronous versions of API calls are similar to their synchronous counterpart taking the same parameters as their synchronous counterparts with two exceptions:

Their name carries the *\_async* suffix and the return value is void.

As an example, the *ezdp\_copy\_data()* API call discussed in previous chapters which invokes a DMA copy operation of a buffer from a frame buffer in IMEM or EMEM to a buffer in local core CMEM memory, has an asynchronous version named *ezdp\_copy\_data\_async()*. Whereas the synchronous *ezdp\_copy\_data()* call will initiate the DMA transaction and schedule out the hardware thread until such time as the DMA transaction has finished before returning control to your program, the *ezdp\_copy\_data\_async()* variant will initiate the DMA copy and return immediately.

### 4.18.2 Synchronization and Scheduling

Since the asynchronous API starts the requested operation and returns immediately before it is finished, an explicit synchronization call is required to guarantee the availability of results of the requested operations. Two synchronization and scheduling API call variants are provided.

#### 4.18.2.1 Schedule out and synchronize on all pending transactions

**void ezdp\_sync(void);**

The *ezdp\_sync()* API call instructs the core to schedule out, if possible, the running hardware thread, freeing the execution unit for other threads to make progress.

The hardware thread will be made eligible again for scheduling and finally return to the execution unit only after **all** pending transactions of the thread are complete.

From the programmer's viewpoint, after a call to `ezdp_sync()` all operations that were initiated prior to the `ezdp_sync()` call are guaranteed to have completed.

#### 4.18.2.2 Schedule out and synchronize on read transactions

**void ezdp\_rsync(void);**

The `ezdp_rsync()` API call instructs the core to schedule out, if possible, the running hardware thread, freeing the execution unit for other threads to make progress.

The hardware thread will be made eligible again for scheduling and finally return to the execution unit only after all pending read transactions of the thread are complete.

Read transactions are those transactions that have a return result that the core needs in order to make progress.

From the programmer's viewpoint, after a call to `ezdp_rsync()` all read operations that were initiated prior to the `ezdp_sync()` call are guaranteed to have completed.

### 4.18.3 Using Asynchronous Operations to Achieve Parallelism

As several concurrent asynchronous operations can be started at once and as synchronization works on all pending transactions of the requested types together, careful use of the asynchronous APIs can be used to run several operations in parallel.

The following code examples show how a double CMEM buffer is used to first copy via DMA the first frame buffer, then invoke a MAC decoder to check the frame header while in parallel using DMA to copy the second frame buffer:

```
/* Load the first buffer to CMEM */
ezdp_load_frame_data(double_buffer[0], &first_bd, job_desc.fd.header_offset, \
    FRAME_HEADER_LENGTH, 0);

/* Now the first frame buffer is available in CMEM */

/* Start a MAC decode async */
ezdp_decode_mac_async(double_buffer[0], FRAME_HEADER_LENGTH, &decode_result);

/* In parallel, load the 2nd buffer to CMEM */
ezdp_load_frame_data_async(&double_buffer[1], &second_bd, 0, FRAME_LENGTH, 0);

/* Synchronize on the completion of both transactions */
ezdp_sync();

/* Check the result of the MAC decoder */
if (decode_result.error_codes.decode_error) {
    print_warning("Decode MAC failed - job discarded \n");
    goto L_DISCARD;
}
```

## 5. Sample Reference Applications and Available Libraries

### 5.1 Background

The NPS software environment includes a rich set of sample applications and libraries. In most cases these are provided with code and should serve as a reference for how to use key features efficiently. The following sub sections briefly describe each of the libraries and sample applications.

It should be noted that EZchip is investing heavily in the NPS software stack and as such new libraries and applications are constantly added. For the latest on this please refer to our website (<http://www.ezchip.com>).

### 5.2 Software Libraries

#### 5.2.1 Stateful Flow Table (SFT)

The SFT is a highly-optimized hardware-accelerated library designed to offload the mapping of packets to stateful contexts at wire speed by an EZchip software component running on the NPS rather than by the client service.

The importance of this is enhanced further as the need to co-locate multiple such services is growing. In many places in the network today stateful security features, such as intrusion prevention system (IPS), FW, and DLP, are co-located with services providing stateful traffic management, L7 visibility, application-aware load balancing and optimization. The EZchip approach uses the wire speed SFT, and DPI engine, as infrastructure that is designed to serve multiple co-located services.

The SFT library delivers the following:

- HW optimized lookup utilizing the NPS hardware acceleration to provide wire packet to flow mapping. The SFT is designed so that a single lookup provides multiple contexts (e.g. identity, session) for clients that are multi-context aware (e.g. NGFW and DLP).
- Context lifetime management offloading in favor of stateful clients. The SFT statefully follows the lifetime of its flows and related contexts, this part of the functionality includes aging. Stateful clients often allocate state (i.e. memory) per flow or user. This memory must be de-allocated once the flow is terminated or the user has no more open flows. Registered clients would be notified by the SFT when a flow terminates or when a user does not own any more flows, so that clients can release relevant client-specific memory, thus saving the need to actively follow the context life time.
- Flow and packet state information for clients through SFT APIs.
- Context aware fast path actions so that a flow, or any other supported context, may be dropped, bypassed, decrypted, encrypted and/or shaped by the NPS without the need to send the following packets to the client service.

##### 5.2.1.1 Library Overview

The NPS SFT is a stateful software component provided by EZchip in favor of stateful clients (e.g. FW). The SFT is optimized hardware accelerated code that utilizes the NPS hardware engines and multi-core, multi-threaded, run-to-completion architecture to provide wire-speed stateful awareness. Among the supported statefulness are flows, protocol sessions and user awareness.

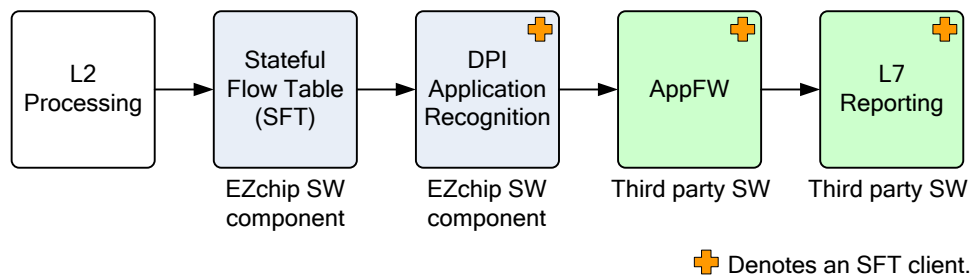
An SFT client is a stateful service which uses the SFT as infrastructure. The SFT is a client agnostic software component that is designed to support multiple such clients at the same time. For example, a vendor's solution may include the functionalities such as DPI, some stateful security features, and an applicative load balancing service. All of these require flow statefulness which is provided in a unified manner to all once they are implemented as SFT clients.

### 5.2.1.2 Library Description

The NPS has 256 cores each running 16 threads. The packet processing at each has a run-to-completion execution model that processes one packet per thread at a time. *Current flow* in this specification refers to the flow to which the current thread's packet belongs. The NPS can process 4096 such "current flows" at a time. Clients using the SFT library do not need to be bothered by this inherent parallelism. All APIs and data structures relating the current flow may be used without locking or similar mechanisms.

The figure below is an example of a feature path implemented in the NPS. The example includes multiple SFT clients. The intention here is to demonstrate where the SFT is located and to discuss below how it is used by clients.

**Figure 23. Sample NPS feature path**



The SFT receives the packets after they have completed their L2 and L3 processing. When receiving a packet, the SFT constructs a lookup key with which it attempts to identify an already opened flow entry to which the packet belongs. If such an entry exists (i.e. the common case), the SFT L4 flow state machine will be updated. If it does not exist, a new flow entry will be created.

Next the packet enters the DPI application recognition (AR) client. It uses the SFT APIs to check if the AppId in the flow entry is final and does not require more processing. When this is the case the DPI AR will be skipped, otherwise the DPI client will use an SFT API to access its client-specific memory which may hold a regular expression (regex) intermediate state from which the DPI should continue processing. When the DPI is done with the packet, it may update the flow AppId.

Next the packet arrives at the FW which may decide to drop the flow based on the provisioned policy and the AppId for the current flow received directly from the SFT. In some cases the FW will reach a conclusion about the flow and may decide to drop it. Alternatively, it may perform additional stateful security functionality and would use the SFT client object set by it before.

Finally the packet may reach an L7 reporting client whose purpose may be to produce a NetFlow message with flow counters and the L7 AppId. This would be required only when the flow terminates. This client would update the counter for every packet that arrives and would send a NetFlow report only when the SFT has identified an end of flow event and has notified its callback.

Readers should note that the demonstrated clients do not require any lookups beyond the initial one performed by the SFT. In addition, the example described depicts SFT use by multiple clients per packet.



### 5.2.1.3 What is an SFT Flow?

A flow is defined as a bi-directional 6-tuple entity with the following fields:

- Source IP Address
- Source Port
- Destination IP Address
- Destination Port
- Protocol Number
- Virtual routing and forwarding (VRF) number / VLAN ID

#### Flow creation

Upon packet arrival the SFT constructs the lookup key and issues a HW accelerated lookup attempting to identify the suitable flow entry in the flow table. If none is identified then a new flow entry is created and the packet is marked accordingly as First In Flow (FIF). Flow entries are opened for any L4 protocol (i.e. not just TCP) over either IPv4 or IPv6. ARP flows are also supported.

In the general case, the SFT would *see* flows from their beginning; however this is not always the case. Flow entries are opened even when flows are not seen from their initial packet.

#### Flow termination

Flows terminate when termination is detected by the SFT L4 state machine, the entry is aged or when the relevant API is called. An EOF callback is called to notify an SFT client service of a flow termination. After the clients release their flow resources, the SFT releases the flow entry.

### 5.2.1.4 Library Feature List

The SFT library supports the following main features:

- **L4 state machine** – The SFT L4 is stateful. For TCP, it identifies properly established flows, terminated flows and the first payload bearing packet. These indications are part of the flow status as well as part of the packet metadata. An API is provided to allow clients to use a different state machine instead of the default one provided as part of the SFT.
- **Fragmentation and OOO handling** – Fragmentation and Out Of Order handling are implemented by the SFT.
- **Client agnostic** – SFT is designed to support multiple clients.
- **Manages flow lifetime for its clients** – The SFT notifies registered clients of flow termination events, thus offloading the host and ensuring system consistency.
- **Manages and retrieves memory for its clients** – The SFT allows each client to store a client object single value that would be part of the flow entry. Clients can directly access the client object through the SFT APIs.
- **Eliminates clients' flow base locking** – NPS is massively parallel. The NPS stateful dispatcher, using a hardware+software architecture, guarantees that packets of the same flow will not be processed at the same time, thus ensuring that clients do not need to use locking when accessing flow memory.
- **6-tuple flow** – The SFT defines a flow as a bi-directional 6-tuple entity with the following fields: source IP address, source port, destination IP address, destination port, protocol number and virtual routing and forwarding (VRF) number / VLAN ID
- **Bi-directionality and NAT support** – An SFT flow is a bi-directional entity. Since the other side may in some cases be NAT-ed, NAT translation is enabled.
- **Hardware accelerated hash and lookup**

- **Packet direction** – The SFT indicates whether the packet is the flow's initiator and if it is the First In Flow (FIF).
- **Tunnel stripping** – Much of the traffic may be tunneled which limits the ability to apply effective flow based stateful policies. The SFT strips some of the tunnel types. In these cases, the flow is determined by the internal tuple.
- **IPv4 and IPv6 support** – A flow is created for any L4 protocol over either IPv4 or IPv6.
- **Denial of service** – The SFT performs DOS identification and DOS handling.
- **Shortage handling** – The SFT identifies a shortage (i.e. memory shortage in the flow table), signals it, uses a more aggressive aging mechanism, and maps the traffic to a default shortage flow entry.
- **Aging process** – The aging process is important in order to deal with shortage of the SFT. Per flow, the aging time is typically set by the DPI client and the TCP State Machine. At times of shortage, the aging process should get a higher priority and aging should be more aggressive. The aging process of the SFT uses client notification callbacks prior to flow release.
- **Status byte** – Captures the flow's status.
- **Fast path actions** – Several flow based actions are offered by the SFT for the client services to use: Bypass Flow, Drop Flow, Shape Flow, Encrypt/Decrypt Flow and Report Flow.
- **Debugging** – SFT debug capabilities include counters at the table level to augment the table level status.

### 5.2.1.5 Sample Code

The example below demonstrates the use of the SFT library:

```
#include <stdio.h>
#include <ezdp.h>
#include <ezsft.h>

ezsft_t          cur_sft __cmem_var;
struct ezsft_conf config __cmem_var;
struct ezsft_flow cur_flow __cmem_var;
ezsft_general_working_area_t sft_general_wa __cmem_var;
ezsft_working_area_t sft_wa __cmem_var;
char             client_obj[256] __cmem_var;

/* client enumeration is defined in ezsft_defs.h
enum CLIENT_ENUMERATION
{
    DPI_AR = 0,
    TCP_STACK = 1
};
*/
void init()
{
    ezsft_init(&cur_sft,&config,&sft_wa);
}
struct ezpacket* process_packet(struct ezdp_job_desc *job_desc,
                                struct ezpacket __cmem *packet)
{
    ezpacket_init (packet);
    //l2 and l3 handling
    if (l2_process(job_desc, packet) == NULL)
        return NULL;
    if (l3_process(job_desc, packet) == NULL)
        return NULL;
    do {
        packet = ezsft_next(&cur_sft,packet,&cur_flow,&sft_wa);
        if (packet != NULL)
        {
            dpi_func(packet,DPI_AR);
        }
    } while (packet != NULL);
}
```

```

        tcp_stack_func(packet, TCP_STACK);
    }
} while (packet != NULL);
}

```

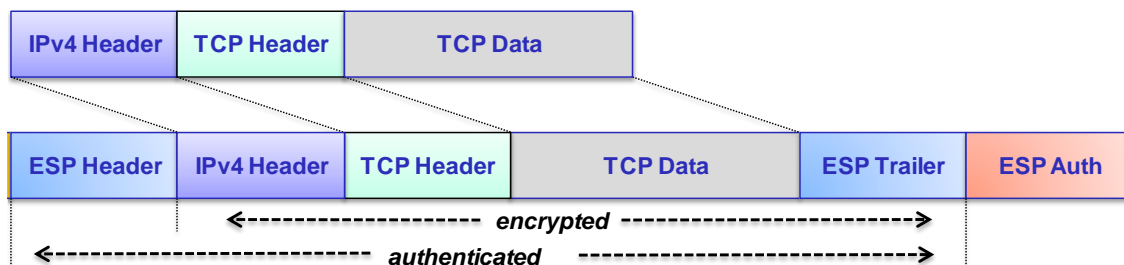
 For additional information refer to the *NPS-400 Stateful Flow Table Application Note*.

## 5.2.2 Crypto Library

CryptoLib is the component (library) of the EZchip [IPsec sample application](#) that handles the actual encryption/ decryption and authentication of the frame according to the IPsec protocol. It utilizes the NPS-400 architecture and accelerators to perform cryptographic tasks efficiently.

The figure below illustrates the changes to a frame undergoing IPsec encapsulation with encryption and authentication.


**Figure 24. IPsec frame encapsulation, encryption and authentication**



This implies two main functionalities of CryptoLib:

- Encapsulation of IPv4 frame with ESP header and trailer (PKCS#7 padding), MAC calculation, encryption of input frame data and appending of the MAC to the end of the frame.
- Decryption of the frame data, MAC calculation and comparing it to the MAC in the frame, decapsulation of the ESP header and MAC. Note that the Trailer is not removed from frame (because size cannot be predicted before calling a CryptoLib API).

The library offers several CryptoLib API routines.

 For additional information refer to the *NPS-400 IPsec Application* document.

## 5.3 Sample Applications

EZchip provides a rich set of software middleware and reference applications. Middleware is comprised of reusable building blocks, modules or stacks that build on the basic data and control plane API to provide higher level functionality to facilitate the fast and simple creation of application with high level of functionality.

The following sections describe several of the reference applications that are offered.

### 5.3.1 CESR Reference Application

The CESR reference application is a full featured Carrier Ethernet Switch Router implemented using the NPS data plane APIs.

Application highlights:

- Performs L2 switching and L3 forwarding.
- Recognizes IEEE 802.1Q VLAN, IEEE 802.1ad QinQ and MPLS tagging.
- Implements Ethernet OAM and IEEE1588v2 Synchronous Ethernet timers.
- Supports IEEE 802.3ad Link Aggregation Groups.
- Incorporates flexible multi-stage frame classification and access control lists.
- Features class of service based traffic shaping.
- Provides performance monitoring, statistics and port mirroring services.
- Scales up to millions of subscribers, 400 Gbps network traffic and 600 Mpps at 64-byte packet size.

#### 5.3.1.1 Application Overview

EZchip provides a reference Carrier Ethernet Switch-Router (CESR) application for EZchip's NPS-400 network processor. The NPS CESR application is designed to support a large scale of services and flows with the ability to aggregate various services on the same device, reducing the number of devices used and simplifying the network design.

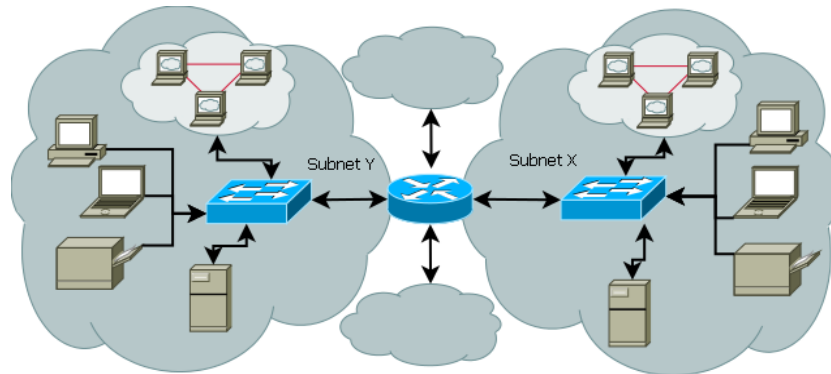
The NPS CESR application supports a vast list of L2 and L3 protocols and is designed with performance efficiency to maximize the bandwidth supported by the NPS-400 device.

#### 5.3.1.2 Application Description

The NPS CESR application provides the following services:

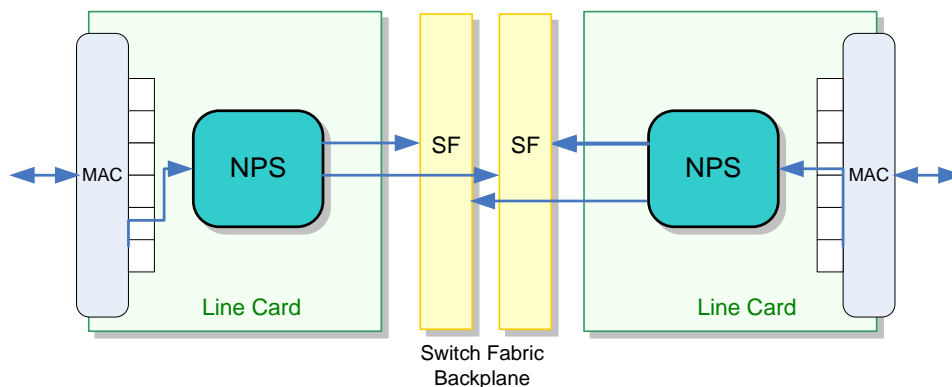
- E-LINE – Point to point service (P2P)
- E-LAN – L2 switching service
- IP routing – L3 routing service

The user can configure a concatenation of two different services on a single NPS device. For example, a service flow that passes through an L2 switch and L3 router on a single box. [Figure 25](#) illustrates a simple network topology with two switches and one router that connect between four different subnets. The NPS can emulate all three devices' functionality in a single device running the CESR application. In this example, the user can create a P2P or E-LAN service between two computers on the same LAN or on different LANs. Another possible configuration for this topology is a switch and router service between computers on one subnet and computers on others subnets. The user can create a full path between end points from two different subnets by configuring a switch & router & switch service flow.

**Figure 25. CESR reference application - network topology**

The CESR application supports additional features such as service level agreement enforcement (SLA), P2P protection, load balancing, statistics collection, security capabilities and additional features that are explained in later sections.

The CESR application is designed to operate on multiple line cards within a chassis-based system, with the line cards connected through a switching fabric. [Figure 26](#) illustrates the generic build of a chassis that this application is intended to run on.

**Figure 26. CESR reference application - chassis module**

The CESR application supports interconnection of all port combinations:


- External interface to switch fabric
- Switch fabric to external interface
- External interface to external interface
- Switch fabric to switch fabric

### 5.3.1.3 Application Feature List

The NPS CESR application supports the following main features:

- **VLAN Processing** – The CESR application can process both tagged and untagged frames. Untagged frames can be tagged with the default VLAN+ configured on the IVIF or be discarded in the classification stage. The classification process can handle multiple tags in a single frame.
- **E-LINE** – Establishing a P2P connection between two end points.
- **E-LAN** – Using an external forwarding database (FDB) to support switching of frames. The application supports the learning of new MAC addresses, flooding of unknown frames, aging mechanism and MAC address filtering. MSTP is also supported and BPDU frames are recognized and sent to the host.
- **IPv4/IPv6 Routing** – Basic IPv4/IPv6 routing using a routing database (RIB). The ARP processing is assumed to be handled by the host and there is no processing of IPv4 frame options.

- **MPLS Encapsulation** – The application provides the ability to encapsulate frames entering an MPLS cloud with an MPLS header. The application supports up to three MPLS labels on a single frame. A frame arriving with more than three labels is forwarded to the Host CPU for further processing.
- **Link Aggregation** – Every physical interface in the NPS device is attached to a LAG interface with up to 8 members per LAG. The LAG functionality supports bandwidth load balancing and protection mechanism. The implementation of LAG relies on the Traffic Manager to select which port of a bundle to send the packet.
- **Security** – ACL tables for MAC and IP addresses prevent pre-configured traffic streams from being forwarded. Another source for filtering MAC addresses is the forwarding database. Denial of Service attacks are prevented by using shapers/pacers on ports/queues.
- **Mirroring** – The ability to mirror a traffic stream is enabled on the following application entities: physical interface, LAG, IVIF, ISP, and ESP.
- **Statistics** – Counters are available on several of the application entities.
- **Debugging** – The application supports a logger mechanism for offline/online debugging. Additional counters are maintained for tracking frame discard.
- **Control Packet Processing** – L2 control frames and L3 control frames can be tunneled, discarded or sent to the host according to the configuration of a specific service.
- **Multicast** – IGMPv2, IGMPv3, and MLD.
- **Advanced QoS** – Implemented using Token Bucket (TB) policers for an IVIF, ISP or service.
- **Protection** – Protection of P2P flows.
- **OAM** – OAM support according to Y.1731: continuity, loss and delay measurements on a service point.
- **IEEE 1588** – Precision Time Protocol (PTP) as described in the IEEE 1588 standard.

 For additional information refer to the *NPS-400 CESR Application* document.

### 5.3.2 DPI based Application Recognition

Deep Packet Inspection (aka DPI) is a broad term that relates to a large set of technologies and algorithms which are used in inspecting traffic at layers 4 and above. Overwhelmingly DPI refers to the stateful inspection of layer 7 messages. DPI engines support a wide range of DPI capabilities as used by the industry today. The provided library includes hundreds of AR signatures.


The DPI-based Application Recognition application for NPS provides up to 800 Gbps application recognition. A DPI engine is loaded with up to 1500 compiled signatures based on a set of primitives. Signatures are arranged based on cost, strength and technology so that fidelity is optimized. Packets are processed by the engine after they have been mapped to flows by the [SFT](#). Flows may be used without locking or similar mechanisms, DPI does not need to be bothered by parallelism. SFT support has already ordered streams so DPI classification can concentrate on layer 7 inspection.

A DPI engine identifies the application and sets the correct Flow AppID based on the loaded AR signatures, sets the flow aging time to suit the application type and bypasses the flow to mark that no further DPI processing is required. Application IDs for the classified flows are stored in the SFT.

Application highlights:

- DPI engine optimized for massively parallel execution.
- Cross packet inspection and field extraction.
- Optimized DPI engine agnostic of loaded signatures.
- Stateful HTTP parser supporting metadata extraction.
- Built in support for multi-flow session application recognition (e.g. SIP & FTP)

- Scales to millions of flows with performance up to 800Gbps when integrated into a CESR router.

 For additional information refer to the “Deep Packet Inspection (DPI)” chapter of the *NPS-400 Architectural Specifications* document.

### 5.3.3 Lawful Interception and Content Filtering

The Lawful Interception and Pattern Matching based Content Filtering reference application for NPS provides hardware-accelerated, flow-based, cross-packet, Bloom filter-based pattern matching based content filtering. A PCRE compiler identifies key sub strings which are loaded into the Bloom filter. Suspected signatures are then inspected, and residual inspection is offloaded to a host.

Application highlights:

- Hardware accelerated, optimized, Bloom-filter pattern matching based content filtering.
- PCRE compiler.
- Flow based, stateful, cross packet inspection.
- Solution built to be extended through the use of an external DPI engine.
- The application scales to millions of concurrent TCP connections for total of 400 Gbps network traffic.


 For additional information refer to the “Bloom Filter based Content Filtering” section in Deep Packet Inspection (DPI) chapter of the *NPS-400 Architectural Specifications* document.

### 5.3.4 IPsec Sample Application

The IPsec sample application is designed to demonstrate the SW and HW abilities of the NPS-400 that enable high-speed and high-performance for IPsec traffic.

Application highlights:

- Security protocols – The IPsec application supports the following crypto and authentication combinations:
  - AES-CBC-128 + HMAC-SHA1-96
  - AES-CBC-256 + HMAC-SHA256-128
  - AES-128 GCM-128
  - 3DES-CBC + MD5
- Mode of operation – The IPsec application supports only tunnel mode, but it could easily be expanded to support the transport mode as well.
- Anti-replay attack – The mechanism of anti-replay attacks employed in the application is defined in IPsec RFCs ("IP Authentication Header" [[RFC4302](#)] and "IP Encapsulating Security Payload (ESP)" [[RFC4303](#)]).
- Packet order – The IPsec application maintains the network order of the IPsec packets per interface.
- 200Gbps for average 410 byte packet.

 For additional information refer to the *NPS-400 IPsec Application* document.

### 5.3.5 L23QoS Sample Application

L23QoS sample application is a simple Layer 2 switching and Layer 3 routing application for NPS-400 network processors. In the sample, there are two possible scenarios depending upon the type of incoming frame:

- L2 Switching – The L2 switching algorithm is applied to frames that are not destined to the router's MAC address. Various VLANs are assumed in the sample. Frames with an unknown destination address or a broadcast DA will be broadcast to all the ports in the VLAN. This algorithm is designed according to IEEE 802.1Q-1988.
- L3 Routing – The L3 routing algorithm is applied to IP frames that are destined to the router's MAC address.

This sample handles Ethernet II frames with and without tags as well as IP frames at 600 Gbps. It implements MAC learning for all L2 frames. Checksum, TTL and IPv4 validation are performed for L3 frames.

The L23QoS application includes several sample frame files that may be used with the supplied projects. In addition, you may create your own frame files to match additional NPS-400 configurations.

#### Layer 2 Switching

The L2 switching algorithm is applied to frames that are not destined to the router's MAC address. Various VLANs are assumed in the sample. For cases where the destination is known (Destination Address was matched in the MAC table (DA hash), the frame is forwarded according to the output port obtained from the MAC table. The VLAN table is also used as a validation that the input and output ports participate in the VLAN group.

Frames with an unknown destination address or a broadcast DA will be broadcasted to all the ports in the VLAN group, excluding the input port. This algorithm is designed according to IEEE 802.1Q-1988. In this case, the VLAN table provides all destination ports.


In cases of L2 switching, the packet content is not modified.

#### Layer 3 Routing

The L3 routing algorithm is applied to IP frames that are destined to the router's MAC address. A lookup of the Destination Address (DIP) is performed in the IP table (DIP lpm) in order to determine the outgoing interface and frame modifications to be carried out.

In the L3 header, the TTL field of the IP header is decremented by one, and the checksum field is recalculated.

In the L2 header, the new DA and VLAN tags are replaced with values taken from the IP forwarding table (DIP lpm), whereas the new SA is taken from the output port configuration table.

 For additional information refer to the *NPS-400 L23QoS Application* document.

### 5.3.6 TCP Stack

The internet protocol suite commonly known as TCP/IP is the networking model and a set of communications protocols used for the Internet and similar networks. TCP/IP provides end-to-end connectivity specifying how data should be formatted, addressed, transmitted, routed and received at the destination. This functionality has been organized into abstraction layers which are used to sort all related protocols according to the scope of networking involved.

EZchip provides a limited TCP/IP stack for the NPS-400 that is designed to be completely modular. The core stack is a TCP over IPv4 implementation, and the user can choose to add other protocols, such as UDP and IPv6. This implementation does not support higher nor lower level protocols (ARP, HTTP, etc.).



The TCP/IP stack is comprised of several main functional modules. The following list provides a high level overview these modules, their names and functions.

### **Stateful Classification (SC)**

The Stateful Classification module, henceforth referred to as SC in this document, provides lockless classification of incoming packets into sessions. This SC module ensures that each session will be handled by at most one thread at a time. A session is defined by its 5-tuple (local IP address, remote IP address, local port, remote port, virtual network identifier).

The SC module maintains a hash entry in a Stateful hash for each session, each contains a task queue. Each task represents single quanta of work: a frame to process (RX or TX), an expired timer event or close session event.

Tasks received by a thread are executed only if there is no any other thread which handles tasks for the same session. If there is such a thread (leader), additional tasks will be added to the session's task queue and will be processed later on by the leader in a FIFO order.

The SC module is also responsible for maintaining ordering between different tasks belonging to the same session.

The number of entries in the Stateful hash is configurable and should be equal to the total number of TCP sessions.

### **IPv4**

The IPv4 module implements IPv4 protocol according to RFC 791 and includes functionality such as validation and IP-reassembly (excluding fragments retransmission).

### **TCP**

The TCP module implements TCP protocol according to RFC 793. Missing functionality includes option handling and flags handling.

## 5.4 Smaller Reference Applications

Both the reference application listed below and the sample applications listed above should be viewed as references to proper and efficient NPS programming. The applications listed in this section are smaller more pinpointed in their scope.

### 5.4.1 PCI Endpoint Stack Reference Application

The PCI Endpoint Stack provides a framework for efficient communication of the NPS data plane with host-based control or data plane applications over the PCI interface.

Application highlights:

- Support for generic software data and frame data transport.
- Support for software defined flexible DMA buffer structure for any application.
- Support for software defined flexible configuration space for any application.
- Supports multiple controllers, physical functions and SR-IOV virtual functions.
- Supports a flexible number of queues per application, instances and more.
- Linux network device driver included.
- DPDK network device driver included.

## A. Acronyms and Abbreviations

This section contains an alphabetical list of the acronyms, abbreviations and terms frequently used in this document. Terms **highlighted** are of particular relevance to the NPS-400.

ACK	ACKnowledge message	DDR3 SDRAM	Double Data Rate Type Three Synchronous Dynamic Random Access Memory
ACL	Access Control List		
AES	Advanced Encryption Standard		
Aging	On-chip mechanism to delete old entries from hash tables	DDR4 SDRAM	Double Data Rate Type Four Synchronous Dynamic Random Access Memory
AH	Authentication Header protocol		
ALU	Arithmetic Logic Unit; for computational functions	DFA	Deterministic Finite Automaton
ARP	Address Resolution Protocol	DMA	Direct Memory Access
ASID	Address Space Identifier	DPI	Deep Packet Inspection
ATM	Asynchronous Transfer Mode	DSLAM	Digital Subscriber Line Access Multiplexers
<b>BD</b>	Buffer Descriptor; data structure describing a single frame buffer	EBP	Explicit Burst Protection
BDR	Buffer Descriptor Ring	EBS	Excess Burst Size
BLT	Buffer Link Table	ECC	Error-Correcting Code memory
<b>BMU</b>	Buffer Manager Unit; manages pools of frame buffers and of jobs, provides resource budget management	EDT	Expandable Direct Table
BSD	Berkley Standard Distribution	EF	Expedited Forwarding
<b>BST</b>   BIST	Built in Self Test	EFA	Ethernet Fabric Adaptor
BT	Burst Tolerance	EIR	Excess Information Rate
BW	Bandwidth	ELF	Executable and Linkable Format
CAM	Content Addressable Memory	<b>EMEM</b>	NPS External Memory
CAUI	100 GbE (10 x 10.3125Gbps) Attachment Unit Interface (electrical interface) (C Roman numeral for 100)	Entry	User-defined data in search structure, i.e. key, result and other associated data
CBR	Constant Bit Rate	EP	Endpoint; PCIe
CBS	Committed Burst Size for srTCM	EPC	Evolved Packet Core; 4G mobile
CDPI	Coarse DPI	ESP	Encapsulation Security Payload protocol
CDR	Clock Data Recovery	ETS	Enhanced Transmission Selection
CESR	Carrier Ethernet Switches and Routers	<b>EZcesr</b>	EZchip Carrier Ethernet switching and routing application
CIR	Committed Information Rate	<b>EZcp</b>	EZchip Control Plane SDK; APIs for configuration and management of the NPS device
<b>CIU</b>	Cluster Interface Unit	<b>EZdk</b>	EZchip SDK is a comprehensive set of design and runtime tools developing both data-plane and control-plane applications for NPS
<b>CLDMA</b>	Cluster DMA; provides data transaction services to NPC threads	<b>EZdp</b>	EZchip Data Plane SDK; API library for data plane services
<b>CMEM</b>	Core's Local Memory; part of CTOP	<b>EZdpi</b>	EZchip deep packet inspection solution
CMT	Channel Mapping Table	<b>EZetcamConfig</b>	EZchip utility for use with external TCAM; supplied with EZdk
CN	Congestion Notification	<b>EZide</b>	EZchip Eclipse™-based Integrated Development Environment
COS   CoS	Class of Service	<b>EZ-ISA</b>	EZchip Instruction Set Architecture
CPPI	100 Gigabit Parallel Physical Interface	<b>EZldk</b>	EZchip Linux® development kit
<b>CTOP</b>	EZchip's C-programmable Task Optimized Processor; internal a multi-threaded processor core	<b>EZnet</b>	NPS cross chip fabric to interconnect several internal blocks including CTOPs, DMAs and PMUs
CTT	Connection Translation Table	<b>EZof</b>	EZchip OpenFlow
DA	Destination Address		
DCB	Data Center Bridging		
DDP	Direct Data Placement		

<b>EZpci</b>	EZchip high bandwidth PCIe interconnect	<b>IMEM</b>	NPS Internal Memory
<b>EZregex</b>	EZchip PCRE compiler	Interlaken	An interconnect protocol optimized for high bandwidth and reliable packet transfers
<b>EZsec</b>	EZchip IPsec encryption/decryption and authentication	Interlaken LA	ILKN-LA Interlaken Look-Aside, protocol suitable for short, transaction-related transfers
<b>EZsim</b>	EZchip NPS Simulator; instruction accurate simulation of CTOPs	IPG	Inter Packet Gap; also IFG
<b>EZtap</b>	EZchip utility to create Linux® TAP virtual network interfaces	IPS	Intrusion Prevention Systems
<b>EZtcp</b>	EZchip TCP/IP protocol stack	IPsec	Internet Protocol Security; protocol suite for securing IP communications
<b>FC</b>	Flow Control	ISA	Instruction Set Architecture
<b>FD</b>	Frame Descriptor	ISR	Interrupt Service Routines
<b>FDPI</b>	Fine DPI	ISSU	In Service Software Update
<b>FEC</b>	Forward Error Correction	ITT	Initiator Transfer Tag
<b>FIC</b>	Fabric Interface Controller	IV	Initialization Vector
<b>FID</b>	Flow ID for NPS Traffic Manager	<b>JD</b>	Job Descriptor; data structure tracking the state of all jobs being processed; holds information about the frame which is associated with the job
<b>FLT</b>	Frame Link Table	<b>JID   Job ID</b>	Job Identifier; token by which one references a single job
<b>FMT</b>	Fixed Mapping Table; for implementing virtual memory	<b>Job</b>	Concurrent tasks on multiple processors in the system
<b>FPX</b>	Floating Point Acceleration; in CTOP	<b>JSM</b>	Job Status Manager; in MTM
<b>FQ</b>	Fair Queuing	<b>JTLB</b>	Joint Translation Lookaside Buffers
<b>GbE   GE</b>	Gigabit Ethernet	<b>Key</b>	Used to perform a lookup in a search (data) structure.
<b>Gbps</b>	Gigabits per second	<b>LAG</b>	Link Aggregation Group
<b>GCI</b>	GigaChip Interface; high bandwidth, low latency memory interface for external memory	<b>LBD</b>	Linked Buffer Descriptor; an array of buffers descriptors
<b>GIM</b>	NPS General Interrupt Controller block	<b>LCMEM</b>	Local Code Memory
<b>HBA</b>	Host Bus Adapter	<b>LDMEM</b>	Local Data Memory
<b>HEC</b>	Header Error Checks	<b>LKP</b>	Look-up in a search (data) structure
<b>HMC</b>	Hybrid Memory Cube; high performance memory	<b>Logical ID</b>	Encodes information of the kind of processing required by the software for this particular job; assigned to each frame at its point of origin
<b>Host</b>	Refers to an external control CPU	<b>LLP</b>	Linear Line Protection
<b>HT</b>	HyperTransport	<b>LPM</b>	Longest Prefix Match
<b>HW</b>	Hardware	<b>LRAM</b>	Local RAM
<b>IC</b>	Interrupt Controller	<b>LRU</b>	Least Recently Used
<b>ICMEM</b>	Internal Code Memory	<b>MAC</b>	Message Authentication Code or Media Access Control
<b>ICMEMG</b>	Internal Code Memory Global	<b>MC</b>	Memory Controller or multicast
<b>ICMP</b>	Internet Control Message Protocol	<b>MCN</b>	Memory Core Network
<b>ICU</b>	Input Classification Unit; performs initial packet classification	<b>MDDR</b>	Modified Deficit Round Robin
<b>ICV</b>	Integrity Check Value	<b>MDN</b>	Memory DMA Network
<b>IDE</b>	Integrated Development Environment	<b>MEF</b>	Metro Ethernet Forum's Traffic Management Specification
<b>IDMEM</b>	Internal Data Memory	<b>MMU</b>	Memory Management Unit; provides virtual memory addressing, memory protection and cache control
<b>IDMEMG</b>	Internal Data Memory Global	<b>MPLS</b>	Multiprotocol Label Switching
<b>IDMEMGIO</b>	Internal Data Memory Global I/O		
<b>IDS</b>	Intrusion Detection Systems		
<b>IE   IFE</b>	Interface Engine; internal blocks		
<b>IFG</b>	Inter Frame Gap; also IPG		
<b>IFU</b>	Interface Units, specifically RxIFU and TxIFU		
<b>IGMP</b>	Internet Gateway Message Protocol		
<b>ILKN</b>	Interlaken interface		
<b>ILKN-LA</b>	Interlaken Look-Aside interface		
<b>ILP</b>	Instruction Level Parallelism		

MPLS-TP	Multiprotocol Label Switching Transport Profile	<b>Protocol decoder</b>	Internal hardware engines perform protocol classification, validation and decoding on a set of known protocols
Mpps	Million packets per second	<b>PSID</b>	Packet Switch ID for NPS Traffic Manager
<b>MSID</b>	Memory Space Identifier	<b>PSOFS</b>	PSID offset
MSN	Messaging Network	PTE	Page Table Entry
MSS	Maximum Segment Size	PTP	Precision Time Protocol; may refer to the IEEE 1588 protocol
<b>MSU</b>	Messaging & Service Unit; internal block	PUL	Power Up Library
<b>MTM</b>	Multi Thread Manager; CTOP HW threads scheduler	QCN	Quantized Congestion Notification
MTP	Multiple-Fibre Pull-off	QDR	Quad Data Rate
MTU	Maximum Transmission Unit; TCP/IP	QoS	Quality of Service
<b>NDMA</b>	Network DMA, specifically RxNDMA and TxNDMA	QSGMII	Quad Serial Gigabit Media Independent Interface
NFA	Nondeterministic Finite Automaton	RD	Raw Data
<b>NP</b>	EZchip Network Processor	RDMA	Remote Direct Memory Access
<b>NPC</b>	Network Processor Cluster; clusters with each consisting of an array of CTOPs with local memory, L2 cache and a data bus interface unit	RE2	Regular Expression Library
<b>NPS</b>	EZchip NP for Smart networks; EZchip product family	Regex	Regular Expression
NPU	Network Processor Unit; such as EZchip's NP and NPS product lines	Result	Key-associated data returned from the match in a search (data) structure.
NRT	Non Real Time traffic	RISC	Reduced Instruction Set Computing
OAM	Operations, Administration and Maintenance	RR	Round Robin
OOB	Out-of-band	RT	Real Time
<b>OQ</b>	Output Queue; internal TM block	RTC	Real Time Clock
OTN	Optical Transport Network	<b>RxIFU</b>	Receive Interface Units; through which frames flow in and out of the system
PADDR	Physical address	<b>RxNDMA</b>	Receive Network DMA; engines which acquire frame buffers from BMU buffer pools and move frame data from the RxIFU to the NPS memory subsystem
PBD	Protection by Duplication	SA	Source Address
PBS	Peak Burst Size	SAR	Segmentation and Reassembly
PCie	PCI Express; Peripheral Component Interconnect Express	<b>SCM</b>	Statistics Counter Manager; maintains posted and on-demand statistical counters in EMEM
PCS	Physical Coding Sublayer	SCR	Sustained Cell Rate
PCR	Peak Cell Rate	<b>SD</b>	Search Descriptor; handler to a specific search structure data element
PCRE	Perl Compatible Regular Expression	SDK	Software Development Kit
<b>PD</b>	Packet Descriptor	SDN	Software Defined Networking
PDU	Protocol Data Unit	SDRAM	Synchronous Dynamic Random Access Memory
PFC	Priority Flow Control	<b>SDRAM Controller</b>	Memory controllers for DDR3/DDR4 SDRAM devices
Phase	May be used refer to hardware revisions of the EZchip device	<b>SDT</b>	Structure Definition Table; internal table defining the data structures
PHB	Per Hop Behavior	Search Structure	Data structure for lookups (e.g. table, hash)
PIB	Policy Information Base	SECDED	Single Error Correction Double Error Detection
PIR	Peak Information Rate	SerDes	Serializer/Deserializer
PMA	Physical Medium Attachment	SLA	Service Level Agreement
PMD	Physical Medium Dependent	SMP	Symmetric Multi-Processing Linux®
<b>PMU</b>	Processor Management Unit; orchestrates the assignment of jobs for processing		
PPS	Packets per second		
<b>PQ</b>	Physical Queue; internally managed		
PRBS	Pseudorandom Binary Sequence		

SoC	System on a Chip
SRAM	Static Random-Access Memory
srTCM	Single Rate Three Color Meter
SSL	Secure Sockets Layer
STAT	Statistics counters manager
Structure	Used for both NPS search structures (i.e. lookup tables) as well as C code structures (i.e. group of parameters).
SW	Software
TCAM	Ternary Content Addressable Memory
TLB	Translation Lookaside Buffer; for implementing virtual memory
TLP	Thread Level Parallelism
TM	Traffic Manager; internal blocks
TOS	Type of Service
trTCM	Two Rate Three Color Meter
TTL	Time To Live
TxIFU	Transmit Interface Units; through which frames flow in and out of the system
TxNDMA	Transmit Network DMA; engines move frame data from the NPS memory subsystem to the TxIFU and release buffer resources to the BMU.
UC	Unicast
UDM	Unified Data Memory; CTOP internal block
UTF-8	UCS (Universal Character Set) Transformation Format, 8-bit
VC	Virtual Channel
VOQ	Virtual Output Queuing
VPN	Virtual Private Network
VRF	Virtual Router Function
WFQ	Weighted Fair Queuing
WRED	Weighted Random Early Discard
WRR	Weighted Round Robin
XLAUI	40 GbE (4 x 10.3125-Gbps) Attachment Unit Interface (electrical interface) (XL Roman numeral for 40)
XLPPi	40 Gigabit Parallel Physical Interface
XSMI	10GE Serial Management Interface
ZOL	Zero Overhead Linux